

**M** MANNING

# ASP.NET Core IN ACTION

SECOND EDITION

Andrew Lock



**Middleware placed before the routing middleware cannot tell which endpoint will be executed.**

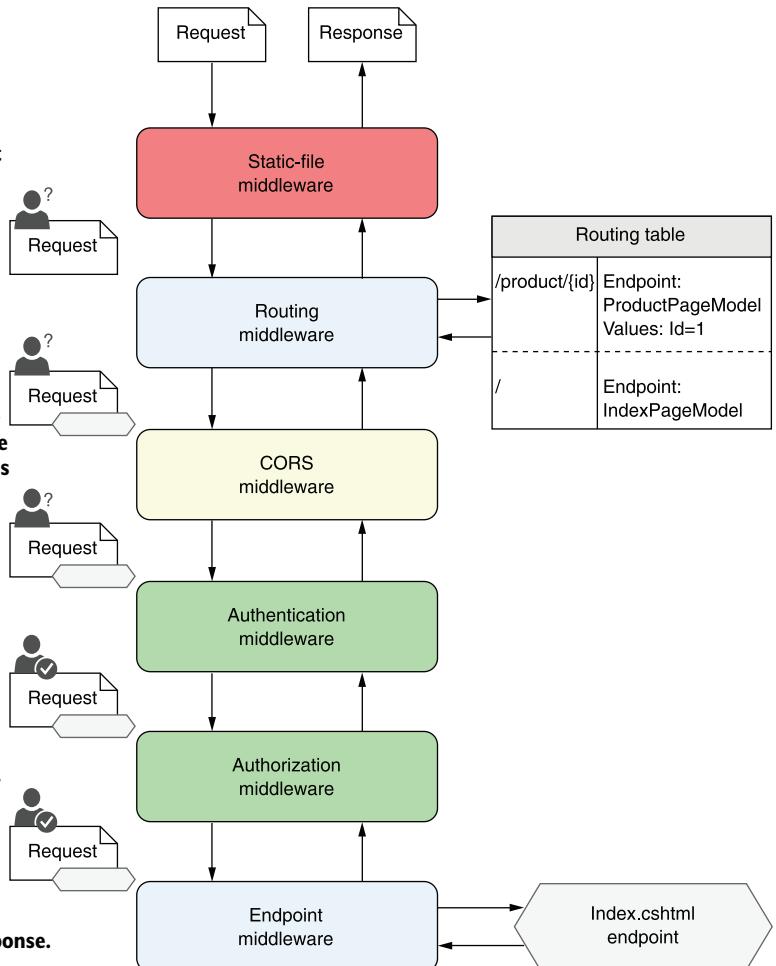
**The routing middleware selects an endpoint based on the request URL and application's route templates.**

**The CorsMiddleware is placed after the routing middleware so it can determine which endpoint was selected and access metadata about the endpoint.**

**Middleware placed before the AuthenticationMiddleware will not see the request as authenticated. Middleware placed afterwards can access the ClaimsPrincipal.**

**The AuthorizationMiddleware must be placed after authentication so the user is known. It must be placed before the endpoint middleware.**

**The endpoint middleware executes the selected endpoint and returns the response.**



*ASP.NET Core in Action, Second Edition*



# *ASP.NET Core in Action*

SECOND EDITION

ANDREW LOCK



MANNING  
SHELTER ISLAND

For online information and ordering of this and other Manning books, please visit [www.manning.com](http://www.manning.com). The publisher offers discounts on this book when ordered in quantity. For more information, please contact

Special Sales Department  
Manning Publications Co.  
20 Baldwin Road  
PO Box 761  
Shelter Island, NY 11964  
Email: [orders@manning.com](mailto:orders@manning.com)

©2021 by Manning Publications Co. All rights reserved.

No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by means electronic, mechanical, photocopying, or otherwise, without prior written permission of the publisher.

Many of the designations used by manufacturers and sellers to distinguish their products are claimed as trademarks. Where those designations appear in the book, and Manning Publications was aware of a trademark claim, the designations have been printed in initial caps or all caps.

⊗ Recognizing the importance of preserving what has been written, it is Manning's policy to have the books we publish printed on acid-free paper, and we exert our best efforts to that end. Recognizing also our responsibility to conserve the resources of our planet, Manning books are printed on paper that is at least 15 percent recycled and processed without the use of elemental chlorine.

 Manning Publications Co.  
20 Baldwin Road  
PO Box 761  
Shelter Island, NY 11964

Development editor: Marina Michaels  
Technical development editor: Mark Elston  
Review editor: Mihaela Batinic  
Production editor: Deirdre S. Hiam  
Copy editor: Andy Carroll  
Proofreader: Jason Everett  
Technical proofreader: Tanya Wilke  
Typesetter: Dennis Dalinnik  
Cover designer: Marija Tudor

ISBN: 9781617298301  
Printed in the United States of America

# *brief contents*

---

## **PART 1 GETTING STARTED WITH ASP.NET CORE .....**1

- 1 ■ Getting started with ASP.NET Core 3**
- 2 ■ Your first application 25**
- 3 ■ Handling requests with the middleware pipeline 58**
- 4 ■ Creating a website with Razor Pages 91**
- 5 ■ Mapping URLs to Razor Pages using routing 122**
- 6 ■ The binding model: Retrieving and validating user input 156**
- 7 ■ Rendering HTML using Razor views 188**
- 8 ■ Building forms with Tag Helpers 223**
- 9 ■ Creating a Web API for mobile and client applications using MVC 255**

## **PART 2 BUILDING COMPLETE APPLICATIONS .....289**

- 10 ■ Service configuration with dependency injection 291**
- 11 ■ Configuring an ASP.NET Core application 330**
- 12 ■ Saving data with Entity Framework Core 364**

13	■ The MVC and Razor Pages filter pipeline	398
14	■ Authentication: Adding users to your application with Identity	436
15	■ Authorization: Securing your application	470
16	■ Publishing and deploying your application	503
<b>PART 3 EXTENDING YOUR APPLICATIONS .....</b>		<b>537</b>
17	■ Monitoring and troubleshooting errors with logging	539
18	■ Improving your application's security	572
19	■ Building custom components	609
20	■ Building custom MVC and Razor Pages components	640
21	■ Calling remote APIs with IHttpClientFactory	667
22	■ Building background tasks and services	689
23	■ Testing your application	714

# *contents*

---

<i>preface</i>	xix
<i>acknowledgments</i>	xxi
<i>about this book</i>	xxiii
<i>about the author</i>	xxviii
<i>about the cover illustration</i>	xxix

## PART 1 GETTING STARTED WITH ASP.NET CORE .....1

### **1** *Getting started with ASP.NET Core* 3

#### 1.1 An introduction to ASP.NET Core 4

*Using a web framework* 5 ▪ *What is ASP.NET Core?* 7

#### 1.2 When to choose ASP.NET Core 9

*What type of applications can you build?* 9 ▪ *If you're new to .NET development* 12 ▪ *If you're a .NET Framework developer creating a new application* 14 ▪ *Converting an existing ASP.NET application to ASP.NET Core* 18

#### 1.3 How does ASP.NET Core work? 19

*How does an HTTP web request work?* 19 ▪ *How does ASP.NET Core process a request?* 21

#### 1.4 What you will learn in this book 23

- 2 Your first application 25**
- 2.1 A brief overview of an ASP.NET Core application 26
  - 2.2 Creating your first ASP.NET Core application 29
    - Using a template to get started* 29 ▪ *Building the application* 32
  - 2.3 Running the web application 34
  - 2.4 Understanding the project layout 35
  - 2.5 The .csproj project file: Defining your dependencies 37
  - 2.6 The Program class: Building a web host 39
  - 2.7 The Startup class: Configuring your application 42
    - Adding and configuring services* 44 ▪ *Defining how requests are handled with middleware* 45
  - 2.8 Generating responses with Razor Pages 50
    - Generating HTML with Razor Pages* 51 ▪ *Handling request logic with PageModels and handlers* 53
- 3 Handling requests with the middleware pipeline 58**
- 3.1 What is middleware? 60
  - 3.2 Combining middleware in a pipeline 64
    - Simple pipeline scenario 1: A holding page* 65 ▪ *Simple pipeline scenario 2: Handling static files* 68 ▪ *Simple pipeline scenario 3: A Razor Pages application* 72
  - 3.3 Handling errors using middleware 77
    - Viewing exceptions in development: DeveloperExceptionPage* 78
    - Handling exceptions in production: ExceptionHandlerMiddleware* 80
    - Handling other errors: StatusCodePagesMiddleware* 84
    - Error handling middleware and Web APIs* 89
- 4 Creating a website with Razor Pages 91**
- 4.1 An introduction to Razor Pages 93
    - Exploring a typical Razor Page* 93 ▪ *The MVC design pattern* 94 ▪ *Applying the MVC design pattern to Razor Pages* 97 ▪ *Adding Razor Pages to your application* 104
  - 4.2 Razor Pages vs. MVC in ASP.NET Core 107
    - MVC controllers in ASP.NET Core* 108 ▪ *The benefits of Razor Pages* 110 ▪ *When to choose MVC controllers over Razor Pages* 112
  - 4.3 Razor Pages and page handlers 113
    - Accepting parameters to page handlers* 115 ▪ *Returning responses with ActionResults* 117

- 5 Mapping URLs to Razor Pages using routing 122**
- 5.1 What is routing? 123
  - 5.2 Routing in ASP.NET Core 126
    - Using endpoint routing in ASP.NET Core 127 ▪ Convention-based routing vs. attribute routing 130 ▪ Routing to Razor Pages 132*
  - 5.3 Customizing Razor Page route templates 134
    - Adding a segment to a Razor Page route template 136*
    - Replacing a Razor Page route template completely 137*
  - 5.4 Exploring the route template syntax 138
    - Using optional and default values 138 ▪ Adding additional constraints to route parameters 139 ▪ Matching arbitrary URLs with the catch-all parameter 142*
  - 5.5 Generating URLs from route parameters 143
    - Generating URLs for a Razor Page 144 ▪ Generating URLs for an MVC controller 145 ▪ Generating URLs with ActionResults 146*
    - Generating URLs from other parts of your application 147*
  - 5.6 Selecting a page handler to invoke 149
  - 5.7 Customizing conventions with Razor Pages 151
- 6 The binding model: Retrieving and validating user input 156**
- 6.1 Understanding the models in Razor Pages and MVC 157
  - 6.2 From request to model: Making the request useful 160
    - Binding simple types 164 ▪ Binding complex types 168*
    - Choosing a binding source 172*
  - 6.3 Handling user input with model validation 174
    - The need for validation 174 ▪ Using DataAnnotations attributes for validation 176 ▪ Validating on the server for safety 178*
    - Validating on the client for user experience 182*
  - 6.4 Organizing your binding models in Razor Pages 184
- 7 Rendering HTML using Razor views 188**
- 7.1 Views: Rendering the user interface 189
  - 7.2 Creating Razor views 193
    - Razor views and code-behind 194 ▪ Introducing Razor templates 195 ▪ Passing data to views 197*

- 7.3 Creating dynamic web pages with Razor 199
  - Using C# in Razor templates* 200 ▪ *Adding loops and conditionals to Razor templates* 201 ▪ *Rendering HTML with Raw* 204
- 7.4 Layouts, partial views, and \_ViewStart 206
  - Using layouts for shared markup* 207 ▪ *Overriding parent layouts using sections* 209 ▪ *Using partial views to encapsulate markup* 211 ▪ *Running code on every view with \_ViewStart and \_ViewImports* 213
- 7.5 Selecting a view from an MVC controller 215

## 8 *Building forms with Tag Helpers* 223

- 8.1 Catering to editors with Tag Helpers 225
- 8.2 Creating forms using Tag Helpers 228
  - The Form Tag Helper* 233 ▪ *The Label Tag Helper* 235
  - The Input and Textarea Tag Helpers* 236 ▪ *The Select Tag Helper* 240 ▪ *The Validation Message and Validation Summary Tag Helpers* 245
- 8.3 Generating links with the Anchor Tag Helper 248
- 8.4 Cache-busting with the Append Version Tag Helper 250
- 8.5 Using conditional markup with the Environment Tag Helper 251

## 9 *Creating a Web API for mobile and client applications using MVC* 255

- 9.1 What is a Web API and when should you use one? 256
- 9.2 Creating your first Web API project 259
- 9.3 Applying the MVC design pattern to a Web API 266
- 9.4 Attribute routing: Linking action methods to URLs 270
  - Combining route attributes to keep your route templates DRY* 272
  - Using token replacement to reduce duplication in attribute routing* 274 ▪ *Handling HTTP verbs with attribute routing* 274
- 9.5 Using common conventions with the [ApiController] attribute 276
- 9.6 Generating a response from a model 280
  - Customizing the default formatters: Adding XML support* 282
  - Choosing a response format with content negotiation* 284

**PART 2 BUILDING COMPLETE APPLICATIONS .....289****10 Service configuration with dependency injection 291**

## 10.1 Introduction to dependency injection 292

*Understanding the benefits of dependency injection 293**Creating loosely coupled code 298 ▀ Dependency injection in ASP.NET Core 300*

## 10.2 Using the dependency injection container 302

*Adding ASP.NET Core framework services to the container 302**Registering your own services with the container 304 ▀ Registering services using objects and lambdas 307 ▀ Registering a service in the container multiple times 311 ▀ Injecting services into action methods, page handlers, and views 315*

## 10.3 Understanding lifetimes: When are services created? 318

*Transient: Everyone is unique 321 ▀ Scoped: Let's stick together 322 ▀ Singleton: There can be only one 323**Keeping an eye out for captured dependencies 324***11 Configuring an ASP.NET Core application 330**

## 11.1 Introducing the ASP.NET Core configuration model 331

## 11.2 Configuring your application with CreateDefaultBuilder 333

## 11.3 Building a configuration object for your app 335

*Adding a configuration provider in Program.cs 338**Using multiple providers to override configuration values 340**Storing configuration secrets safely 342 ▀ Reloading configuration values when they change 346*

## 11.4 Using strongly typed settings with the options pattern 347

*Introducing the IOptions interface 349 ▀ Reloading strongly typed options with IOptionsSnapshot 350 ▀ Designing your options classes for automatic binding 351 ▀ Binding strongly typed settings without the IOptions interface 353*

## 11.5 Configuring an application for multiple environments 354

*Identifying the hosting environment 355 ▀ Loading environment-specific configuration files 356 ▀ Setting the hosting environment 358*

<b>12</b>	<b>Saving data with Entity Framework Core 364</b>
12.1	Introducing Entity Framework Core 366 <i>What is EF Core? 366 ▪ Why use an object-relational mapper? 367 ▪ When should you choose EF Core? 369 Mapping a database to your application code 370</i>
12.2	Adding EF Core to an application 372 <i>Choosing a database provider and installing EF Core 373 Building a data model 375 ▪ Registering a data context 377</i>
12.3	Managing changes with migrations 378 <i>Creating your first migration 379 ▪ Adding a second migration 382</i>
12.4	Querying data from and saving data to the database 384 <i>Creating a record 385 ▪ Loading a list of records 387 Loading a single record 389 ▪ Updating a model with changes 390</i>
12.5	Using EF Core in production applications 394
<b>13</b>	<b>The MVC and Razor Pages filter pipeline 398</b>
13.1	Understanding filters and when to use them 399 <i>The MVC filter pipeline 401 ▪ The Razor Pages filter pipeline 403 ▪ Filters or middleware: Which should you choose? 404 ▪ Creating a simple filter 406 ▪ Adding filters to your actions, controllers, Razor Pages, and globally 408 Understanding the order of filter execution 411</i>
13.2	Creating custom filters for your application 413 <i>Authorization filters: Protecting your APIs 415 ▪ Resource filters: Short-circuiting your action methods 416 ▪ Action filters: Customizing model binding and action results 419 ▪ Exception filters: Custom exception handling for your action methods 423 Result filters: Customizing action results before they execute 425 Page filters: Customizing model binding for Razor Pages 427</i>
13.3	Understanding pipeline short-circuiting 429
13.4	Using dependency injection with filter attributes 431
<b>14</b>	<b>Authentication: Adding users to your application with Identity 436</b>
14.1	Introducing authentication and authorization 437 <i>Understanding users and claims in ASP.NET Core 438 Authentication in ASP.NET Core: Services and middleware 439 Authentication for APIs and distributed applications 442</i>

14.2	What is ASP.NET Core Identity? 446
14.3	Creating a project that uses ASP.NET Core Identity 448 <i>Creating the project from a template 448 □ Exploring the template in Solution Explorer 450 □ The ASP.NET Core Identity data model 453 □ Interacting with ASP.NET Core Identity 455</i>
14.4	Adding ASP.NET Core Identity to an existing project 458 <i>Configuring the ASP.NET Core Identity services and middleware 459 □ Updating the EF Core data model to support Identity 460 □ Updating the Razor views to link to the Identity UI 461</i>
14.5	Customizing a page in ASP.NET Core Identity's default UI 463
14.6	Managing users: Adding custom data to users 466

## 15 Authorization: Securing your application 470

15.1	Introduction to authorization 471
15.2	Authorization in ASP.NET Core 474 <i>Preventing anonymous users from accessing your application 476 □ Handling unauthorized requests 478</i>
15.3	Using policies for claims-based authorization 481
15.4	Creating custom policies for authorization 484 <i>Requirements and handlers: The building blocks of a policy 484 □ Creating a policy with a custom requirement and handler 486</i>
15.5	Controlling access with resource-based authorization 492 <i>Manually authorizing requests with IAuthorizationService 493 □ Creating a resource-based AuthorizationHandler 495</i>
15.6	Hiding elements in Razor templates from unauthorized users 498

## 16 Publishing and deploying your application 503

16.1	Understanding the ASP.NET Core hosting model 504 <i>Running vs. publishing an ASP.NET Core app 506 □ Choosing a deployment method for your application 510</i>
16.2	Publishing your app to IIS 511 <i>Configuring IIS for ASP.NET Core 512 □ Preparing and publishing your application to IIS 514</i>

16.3	Hosting an application on Linux	517
	<i>Running an ASP.NET Core app behind a reverse proxy on Linux</i>	517
	<i>Preparing your app for deployment to Linux</i>	520
16.4	Configuring the URLs for your application	522
16.5	Optimizing your client-side assets using BundlerMinifier	525
	<i>Speeding up an app using bundling and minification</i>	527
	<i>Adding BundlerMinifier to your application</i>	529
	<i>Using minified files in production with the Environment Tag Helper</i>	532
	<i>Serving common files from a CDN</i>	533

## PART 3 EXTENDING YOUR APPLICATIONS ..... 537

### **17 Monitoring and troubleshooting errors with logging 539**

17.1	Using logging effectively in a production app	541
	<i>Highlighting problems using custom log messages</i>	542
	<i>The ASP.NET Core logging abstractions</i>	543
17.2	Adding log messages to your application	545
	<i>Log level: How important is the log message?</i>	547
	<i>Which component created the log</i>	549
	<i>Formatting messages and capturing parameter values</i>	550
17.3	Controlling where logs are written using logging providers	552
	<i>Adding a new logging provider to your application</i>	553
	<i>Replacing the default ILoggerFactory with Serilog</i>	556
17.4	Changing log verbosity with filtering	559
17.5	Structured logging: Creating searchable, useful logs	564
	<i>Adding a structured logging provider to your app</i>	565
	<i>Using scopes to add additional properties to your logs</i>	568

### **18 Improving your application's security 572**

18.1	Adding HTTPS to an application	573
	<i>Using the ASP.NET Core HTTPS development certificates</i>	576
	<i>Configuring Kestrel with a production HTTPS certificate</i>	579
	<i>Enforcing HTTPS for your whole app</i>	580
18.2	Defending against cross-site scripting (XSS) attacks	584
18.3	Protecting from cross-site request forgery (CSRF) attacks	588

18.4	Calling your web APIs from other domains using CORS	593
	<i>Understanding CORS and how it works</i>	594
	<i>Adding a global CORS policy to your whole app</i>	596
	<i>Adding CORS to specific Web API actions with EnableCorsAttribute</i>	598
	<i>Configuring CORS policies</i>	599
18.5	Exploring other attack vectors	601
	<i>Detecting and avoiding open redirect attacks</i>	601
	<i>Avoiding SQL injection attacks with EF Core and parameterization</i>	603
	<i>Preventing insecure direct object references</i>	605
	<i>Protecting your users' passwords and data</i>	605

## 19 *Building custom components* 609

19.1	Customizing your middleware pipeline	610
	<i>Creating simple endpoints with the Run extension</i>	611
	<i>Branching middleware pipelines with the Map extension</i>	612
	<i>Adding to the pipeline with the Use extension</i>	616
	<i>Building a custom middleware component</i>	619
19.2	Creating custom endpoints with endpoint routing	621
	<i>Creating a custom endpoint routing component</i>	622
	<i>Creating simple endpoints with MapGet and WriteJsonAsync</i>	625
	<i>Applying authorization to endpoints</i>	627
19.3	Handling complex configuration requirements	629
	<i>Partially building configuration to configure additional providers</i>	630
	<i>Using services to configure IOptions with IConfigureOptions</i>	632
19.4	Using a third-party dependency injection container	635

## 20 *Building custom MVC and Razor Pages components* 640

20.1	Creating a custom Razor Tag Helper	641
	<i>Printing environment information with a custom Tag Helper</i>	642
	<i>Creating a custom Tag Helper to conditionally hide elements</i>	645
	<i>Creating a Tag Helper to convert Markdown to HTML</i>	647
20.2	View components: Adding logic to partial views	649
20.3	Building a custom validation attribute	654
20.4	Replacing the validation framework with FluentValidation	659
	<i>Comparing FluentValidation to DataAnnotations attributes</i>	660
	<i>Adding FluentValidation to your application</i>	663

## 21 *Calling remote APIs with IHttpClientFactory* 667

- 21.1 Calling HTTP APIs: The problem with HttpClient 668
- 21.2 Creating HttpClients with IHttpClientFactory 674
  - Using IHttpClientFactory to manage HttpClientHandler lifetime* 674 ▀ *Configuring named clients at registration time* 677 ▀ *Using typed clients to encapsulate HTTP calls* 679
- 21.3 Handling transient HTTP errors with Polly 681
- 21.4 Creating a custom HttpResponseMessageHandler 684

## 22 *Building background tasks and services* 689

- 22.1 Running background tasks with IHostedService 690
  - Running background tasks on a timer* 691 ▀ *Using scoped services in background tasks* 694
- 22.2 Creating headless worker services using IHost 696
  - Creating a worker service from a template* 698 ▀ *Running worker services in production* 700
- 22.3 Coordinating background tasks using Quartz.NET 703
  - Installing Quartz.NET in an ASP.NET Core application* 704
  - Configuring a job to run on a schedule with Quartz.NET* 706
  - Using clustering to add redundancy to your background tasks* 708

## 23 *Testing your application* 714

- 23.1 An introduction to testing in ASP.NET Core 716
- 23.2 Unit testing with xUnit 717
  - Creating your first test project* 718 ▀ *Running tests with dotnet test* 720 ▀ *Referencing your app from your test project* 721
  - Adding Fact and Theory unit tests* 724 ▀ *Testing failure conditions* 727
- 23.3 Unit testing custom middleware 728
- 23.4 Unit testing API controllers 731
- 23.5 Integration testing: Testing your whole app
  - in-memory* 734
  - Creating a TestServer using the Test Host package* 735 ▀ *Testing your application with WebApplicationFactory* 738 ▀ *Replacing dependencies in WebApplicationFactory* 740 ▀ *Reducing duplication by creating a custom WebApplicationFactory* 742

23.6 Isolating the database with an in-memory  
EF Core provider 744

*appendix A* Preparing your development environment 751

*appendix B* Understanding the .NET ecosystem 758

*appendix C* Useful references 776

*index* 783



# ***preface***

---

ASP.NET Core 5.0 was released in 2020, more than four years after the release of ASP.NET Core 1.0. But ASP.NET also has a long history prior to ASP.NET Core. That history provided the basis and impetus for the development of ASP.NET Core.

Microsoft released the first version of ASP.NET in 2002 as part of the original .NET Framework 1.0. Since then, it's been through multiple iterations, with each version bringing added features and extensibility. However, each iteration has been built on top of the .NET Framework, and so comes preinstalled in all versions of Windows.

This brings mixed blessings—on the one hand, the ASP.NET 4.x framework today is a reliable, battle-tested platform for building modern applications on Windows. On the other hand, it is also limited by this reliance—changes to the underlying .NET Framework are far-reaching and so consequently slow to roll out, and it fundamentally excludes the many developers building and deploying to Linux or macOS.

When I first began looking into ASP.NET Core, I was one of those developers. A Windows user at heart, I was issued a Mac by my employer and so was stuck working in a virtual machine all day. ASP.NET Core promised to change all that, allowing me to develop natively on both my Windows machine and my Mac.

I was relatively late to the party in many respects, only taking an active interest just before the time of the RC2 release of ASP.NET Core. By that point there had already been eight beta releases, many of which contained significant breaking changes. By not diving in fully until RC2, I was spared the pain of dodgy tooling and changing APIs.

What I saw at that point really impressed me. ASP.NET Core let developers leverage their existing knowledge of the .NET framework, and of ASP.NET MVC applications

in particular, while baking in current best practices like dependency injection, strongly typed configuration, and logging. On top of that, you could build and deploy cross-platform. I was sold.

This book came about largely due to my approach to learning about ASP.NET Core. Rather than simply reading documentation and blog posts, I decided to try something new and start writing about what I learned. Each week I would dedicate some time to exploring a new aspect of ASP.NET Core, and I'd write a blog post about it. When the possibility of writing a book came about, I jumped at the chance—another excuse to dive further into the framework!

Since I started this book, a lot has changed, both with the book and ASP.NET Core. The first major release of the framework in June 2016 still had many rough edges, in particular around the tooling experience. With the release of .NET 5.0 in November 2020, ASP.NET Core has really come into its own, with the APIs and tooling reaching mature levels. This book targets the .NET 5.0 release of ASP.NET Core, but as long as you're using at least .NET Core 3.1, you will be able to follow along without any issues.

This book covers everything you need to get started with ASP.NET Core, whether you're completely new to web development or you're an existing ASP.NET developer. It focuses very much on the framework itself, so I don't go into details about client-side frameworks such as Angular and React or technologies like Docker. I also don't cover all the new features in .NET 5.0 such as Blazor and gRPC. Instead, I provide links where you can find more information.

We'll focus on building server-rendered applications using Razor Pages and Web APIs using MVC controllers. You'll learn the fundamentals of ASP.NET Core, such as middleware, dependency injection, and configuration, and how to customize each for your requirements. You'll learn how to add authentication and authorization to your apps, how to improve their security, and how to deploy and monitor them. Finally, you'll learn how to test your applications using both unit tests and integration tests.

Personally, I find it a joy working with ASP.NET Core apps compared to apps using the previous version of ASP.NET, and I hope that passion comes through in this book!

## *acknowledgments*

---

While there is only one name on the cover of this book, a plethora of people contributed to both its writing and production. In this section I'd like to thank everyone who encouraged me, contributed, and put up with me for the past year.

Firstly, and most importantly, I'd like to thank my girlfriend, Becky. Your continual support and encouragement mean the world to me and have kept me going through such a busy time. You've taken the brunt of my stress and pressure, and I'm eternally grateful. I love you always.

I'd also like to thank my whole family for their support. In particular my parents, Jan and Bob, for putting up with my ranting, and my sister, Amanda, for all your upbeat chats.

On a professional level, I'd like to thank Manning for giving me this opportunity. Brian Sawyer "discovered" me in the first version of this book and encouraged me to tackle the second version. Marina Michaels served as my development editor for the second time running, and again proved alternately meticulous, critical, encouraging, and enthusiastic. The book is undoubtedly better thanks to your involvement. Thanks also to Deirdre Hiam, my project editor; Andy Carroll, my copyeditor; Jason Everett, my proofreader; and Mihaela Batinic, my reviewing editor.

My thanks also go to Mark Elston and Tanya Wilke who served as technical development editor and technical proofer, respectively. Mark provided invaluable feedback, highlighting my incorrect assumptions and technical biases that come from working with a framework I know so well. Tanya Wilke verified that the code I wrote actually ran and made sense, working through the chapters with formidable efficiency.

To everyone at Manning who helped get this book published and marketed, a heartfelt thanks. I'd also like to thank all the MEAP reviewers for their comments, which helped improve the book in innumerable ways.

I would have never been in a position to write this book if it weren't for the excellent content produced by the .NET community and those I follow on Twitter. In particular, thanks to Jon Galloway for regularly featuring my blog on the ASP.NET community standup.

Finally, thanks to all those friends who encouraged and supported me, and showed interest generally. We may not have been able to meet up as much as we'd like, but I look forward to getting together for a drink as soon as it's possible.

To all the reviewers: Al Pezewski, Ben McNamara, Daniel Vásquez, Filip Wojcieszyn, Foster Haines, Gustavo Filipe Ramos Gomes, Jean-François Morin, Joel Kotarski, John Guthrie, Juan Luis Barreda, Luis Moux, Mike Erickson, Raushan Jha, Rob Ruetsch, Ron Lease, Ruben Vandeginste, Sau Fai Fong, Steve Love, Tanya Wilke, Vincent Delcoigne, and Willis G. Hampton, your suggestions helped make this a better book.

# *about this book*

---

This book is about the ASP.NET Core framework: what it is, and how you can use it to build web applications. While some of this content is already available online, it's scattered around the internet in disparate documents and blog posts. This book guides you through building your first applications, introducing additional complexity as you cement previous concepts.

I present each topic using relatively small examples, rather than building on a single example application through the book. There are merits to both approaches, but I wanted to ensure the focus remained on the specific topics being taught, without the mental overhead of navigating an increasingly large project.

By the end of the book, you should have a solid understanding of how to build apps with ASP.NET Core, the framework's strengths and weaknesses, and how to leverage its features to build apps securely. While I don't spend a lot of time on application architecture, I do make sure to point out best practices, especially where I only superficially cover architecture in the name of brevity.

## ***Who should read this book***

This book is for C# developers who are interested in learning a cross-platform web framework. It doesn't assume you have any experience building web applications—you may be a mobile or desktop developer, for example—though previous experience with ASP.NET or another web framework is undoubtedly beneficial.

Other than a working knowledge of C# and .NET, I assume some knowledge of common object-oriented practices and a basic understanding of relational databases

in general. I assume a passing familiarity with HTML and CSS and of JavaScript's place as a client-side scripting language. You don't need to know any JavaScript or CSS frameworks for this book, though ASP.NET Core works well with both if that is your forte.

Web frameworks naturally touch on a wide range of topics, from the database and network to visual design and client-side scripting. I provide as much context as possible, and I include links to sites and books where you can learn more.

### ***How this book is organized: A roadmap***

This book is divided into 3 parts, 23 chapters, and 3 appendixes. Ideally, you will read the book cover to cover and then use it as a reference, but I realize that won't suit everyone. While I use small sample apps to demonstrate a topic, some chapters build on the work of previous ones, so the content will make more sense when read sequentially.

I strongly suggest reading the chapters in part 1 in sequence, as each chapter builds on topics introduced in the previous chapters. Part 2 is also best read sequentially, though most of the chapters are independent if you wish to jump around. You can read the chapters in part 3 out of order, though I recommend only doing so after you have covered parts 1 and 2.

Part 1 provides a general introduction to ASP.NET Core and the overall architecture of a typical web application. Once we have covered the basics, we'll dive into the Razor Pages framework, which makes up the bulk of most ASP.NET Core server-rendered web applications, and the underlying Model-View-Controller (MVC) architecture.

- Chapter 1 introduces ASP.NET Core and its place in the web development landscape. It discusses when you should and shouldn't use ASP.NET Core, the basics of web requests in ASP.NET Core, and the options available for a development environment.
- Chapter 2 walks through all the components of a basic ASP.NET Core application, discussing their roles and how they combine to generate a response to a web request.
- Chapter 3 describes the middleware pipeline, which is the main application pipeline in ASP.NET Core. This defines how incoming requests are processed and how a response should be generated.
- Chapter 4 shows how to use Razor Pages to build page-based websites. Razor Pages are the recommended way to build server-rendered applications in ASP.NET Core, and they are designed for page-based applications.
- Chapter 5 describes the Razor Pages routing system. Routing is the process of mapping incoming request URLs to a specific class and method, which then executes to generate a response.
- Chapter 6 looks at model-binding—the process of mapping form data and URL parameters passed in a request to concrete C# objects.

- Chapter 7 shows how to generate HTML web pages using the Razor template language.
- Chapter 8 builds on chapter 7 by introducing Tag Helpers, which can greatly reduce the amount of code required to build forms and web pages.
- Chapter 9 describes how to use MVC controllers to build APIs that can be called by client-side apps.

Part 2 covers important topics for building full-featured web applications, once you have understood the basics.

- Chapter 10 describes how to use ASP.NET Core's built-in dependency injection container to configure your application's services.
- Chapter 11 discusses how to read settings and secrets in ASP.NET Core, and how to map these to strongly typed objects.
- Chapter 12 introduces Entity Framework Core for saving data into a relational database.
- Chapter 13 builds on the topics in part 1 by introducing the MVC and Razor Pages filter pipeline.
- Chapter 14 describes how to add user profiles and authentication to your application using ASP.NET Core Identity.
- Chapter 15 builds on the previous chapter by introducing authorization for users, so you can restrict which pages a signed-in user can access.
- Chapter 16 looks at how to publish your app, how to configure your app for a production environment, and how to optimize your client-side assets.

The chapters that make up part 3 cover important, cross-cutting aspects of ASP.NET Core development.

- Chapter 17 shows how to configure logging in your application, and how to write log messages to multiple locations.
- Chapter 18 explores some of the security considerations you should be aware of when developing your application, including how to configure your application for HTTPS.
- Chapter 19 describes how to build and use a variety of custom components, such as custom middleware, and how to handle complex configuration requirements.
- Chapter 20 expands on chapter 19 and shows how to build custom Razor Page components, such as custom Tag Helpers and custom validation attributes.
- Chapter 21 discusses the new `IHttpClientFactory` and how to use it to create `HttpClient` instances for calling remote APIs.
- Chapter 22 explores the generic `IHost` abstraction, which you can use to create Windows Services and Linux daemons. You'll also learn to run tasks in the background of your applications.
- Chapter 23 shows how to test an ASP.NET Core application with the xUnit testing framework. It covers both unit tests and integration tests using the Test Host.

The three appendixes provide supplementary information.

- Appendix A describes how to configure your development environment, whether you're on Windows, Linux, or macOS.
- Appendix B provides some background to .NET 5.0, .NET Core, and .NET Standard, discusses how they fit in the .NET landscape, and explains what they mean for your apps.
- Appendix C contains a number of links that I have found useful in learning about ASP.NET Core.

### About the code

This book contains many examples of source code both in numbered listings and in line with normal text. In both cases, source code is formatted in a fixed-width font like this to separate it from ordinary text. Sometimes code is also **in bold** to highlight code that has changed from previous steps in the chapter, such as when a new feature adds to an existing line of code.

In many cases, the original source code has been reformatted; we've added line breaks and reworked indentation to accommodate the available page space in the book. Additionally, comments in the source code have often been removed from the listings when the code is described in the text. Code annotations accompany many of the listings, highlighting important concepts.

Source code is provided for all chapters except chapter 1. You can view the source code for each chapter in my GitHub repository at <https://github.com/andrewlock/asp-dot-net-core-in-action-2e>. A zip file containing all the source code is also available from the publisher's website at [www.manning.com/books/asp-net-core-in-action-second-edition](http://www.manning.com/books/asp-net-core-in-action-second-edition).

All the code examples in this book use .NET 5.0 and were built using both Visual Studio and Visual Studio Code. To build and run the examples, you will need to install the .NET SDK, as described in appendix A.

### liveBook discussion forum

Purchase of *ASP.NET Core in Action, Second Edition*, includes free access to a private web forum run by Manning Publications where you can make comments about the book, ask technical questions, and receive help from the author and from other users. To access the forum, go to <https://livebook.manning.com/book/asp-net-core-in-action-second-edition/discussion>. You can also learn more about Manning's forums and the rules of conduct at <https://livebook.manning.com/#!/discussion>.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the author can take place. It is not a commitment to any specific amount of participation on the part of the author, whose contribution to the forum remains voluntary (and unpaid). We

suggest you try asking the author some challenging questions lest his interest stray! The forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

## *about the author*

---



**ANDREW LOCK** graduated with an Engineering degree from Cambridge University, specializing in Software Engineering, and went on to obtain a PhD in Digital Image Processing. He has been developing professionally using .NET for the last 10 years, using a wide range of technologies including WinForms, ASP.NET WebForms, ASP.NET MVC, and ASP.NET Webpages. He has been building and maintaining applications with ASP.NET Core since the release of 1.0 in 2016. Andrew has a very active blog at <https://andrewlock.net> dedicated to ASP.NET Core. It is frequently featured in the community spotlight by the ASP.NET team at Microsoft, on the .NET blog, and in the weekly community standups. Andrew has been a Microsoft Valued Professional (MVP) since 2017.

## *about the cover illustration*

---

The caption for the illustration on the cover of *ASP.NET Core in Action, Second Edition* is “The Captain Pasha. Kapudan pasha, admiral of the Turkish navy.” The Kapudan Pasha was the highest military rank of the Ottoman Navy from 1567 until 1867 when the post was abolished and replaced with the Naval Minister. The illustration is taken from a collection of costumes of the Ottoman Empire published on January 1, 1802, by William Miller of Old Bond Street, London. The title page is missing from the collection and we have been unable to track it down to date. The book’s table of contents identifies the figures in both English and French, and each illustration bears the names of two artists who worked on it, both of whom would no doubt be surprised to find their art gracing the front cover of a computer programming book ... two hundred years later. The collection was purchased by a Manning editor at an antiquarian flea market in the “Garage” on West 26th Street in Manhattan. The seller was an American based in Ankara, Turkey, and the transaction took place just as he was packing up his stand for the day. The Manning editor didn’t have on his person the substantial amount of cash that was required for the purchase, and a credit card and check were both politely turned down. With the seller flying back to Ankara that evening, the situation was getting hopeless. What was the solution? It turned out to be nothing more than an old-fashioned verbal agreement sealed with a handshake. The seller simply proposed that the money be transferred to him by wire, and the editor walked out with the bank information on a piece of paper and the portfolio of images under his arm. Needless to say, we transferred the funds the next day, and we remain grateful and impressed by this unknown person’s trust in one of us. We at Manning

celebrate the inventiveness, the initiative, and, yes, the fun of the computer business with book covers based on the rich diversity of regional life of two centuries ago, brought back to life by the pictures from this collection.

## *Part 1*

# *Getting started with ASP.NET Core*

**W**eb 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, and we'll 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'll 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'll focus on Razor Pages, which is the main approach to generating responses in ASP.NET Core apps. In chapters 4 through 6 we'll examine the behavior of the Razor Pages framework itself, routing, and model binding. In chapters 7 and 8 we'll look at how you can 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'll explore specific features of ASP.NET Core that let you build Web APIs and see 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'll 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.

# *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 and .NET 5.0
- 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 .NET 5.0, and on the advantages ASP.NET Core can offer 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 5.0, 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

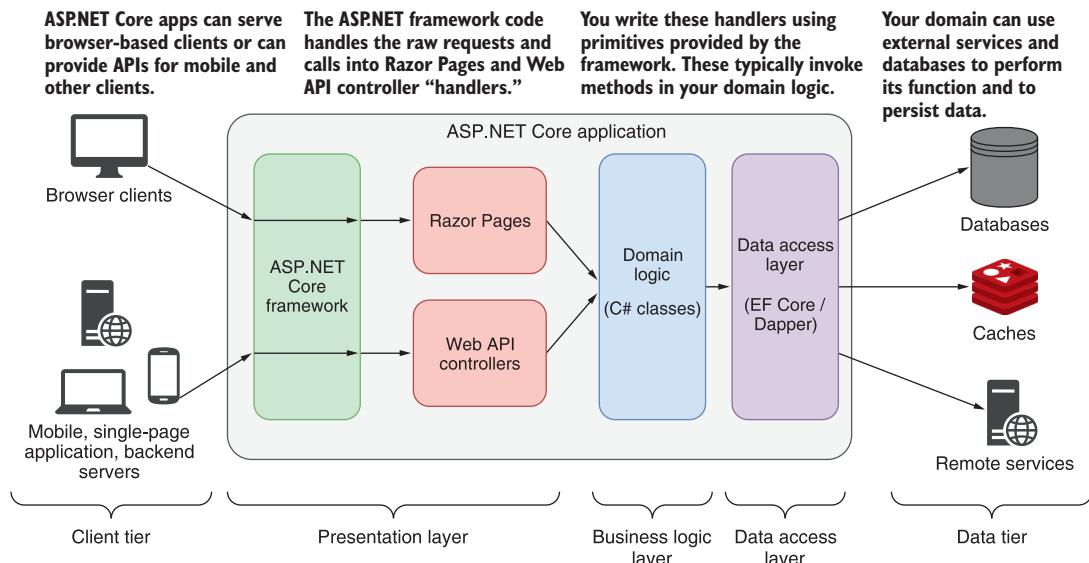
ASP.NET Core is a cross-platform, open source, web application framework that you can use to quickly build dynamic, server-side rendered applications. You can also use ASP.NET Core to create HTTP APIs that can be consumed by mobile applications, by browser-based single-page applications such as Angular and React, or by other backend applications.

ASP.NET Core provides structure, helper functions, and a framework for building applications, which saves you having to write a lot of this code yourself. The ASP.NET Core framework code then calls into your “handlers” which, in turn, call methods in your application’s business logic, as shown in figure 1.1. This business logic is the core of your application. You can interact with other services here, such as databases or remote APIs, but your business logic does not typically depend *directly* on ASP.NET Core.

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.



**Figure 1.1** A typical ASP.NET Core application consists of several layers. The ASP.NET Core framework code handles requests from a client, dealing with the complex networking code. The framework then calls into handlers (Razor Pages and Web API controllers) that you write using primitives provided by the framework. Finally, these handlers call into your application’s domain logic, which are typically C# classes and objects without any ASP.NET Core-specific dependencies.

### 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 desktop or mobile development frameworks can help you build 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 for building 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* may display different data depending on the current logged-in user, or it could display content submitted by users. Without a dynamic framework, it wouldn't be possible to log in to websites or to display any sort of personalized data on a page. In short, websites like Amazon, eBay, and Stack Overflow (seen in figure 1.2) wouldn't be possible.

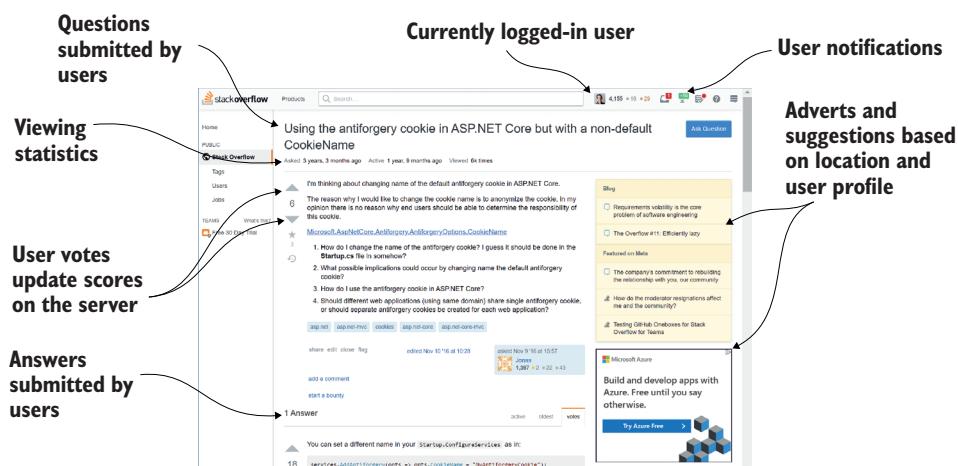


Figure 1.2 The Stack Overflow website (<https://stackoverflow.com>) is built using ASP.NET and is almost entirely dynamic content.

## The benefits and limitations of ASP.NET

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 have a sense of familiarity.

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 it is well known by virtually all Windows web developers.

On the other hand, this reliance is limiting—changes to the underlying System.Web.dll file 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.

More recently, Microsoft has declared .NET Framework to be “done.” It won’t be removed or replaced, but it also won’t receive 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, and thus ASP.NET Core was born.

### 1.1.2 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:

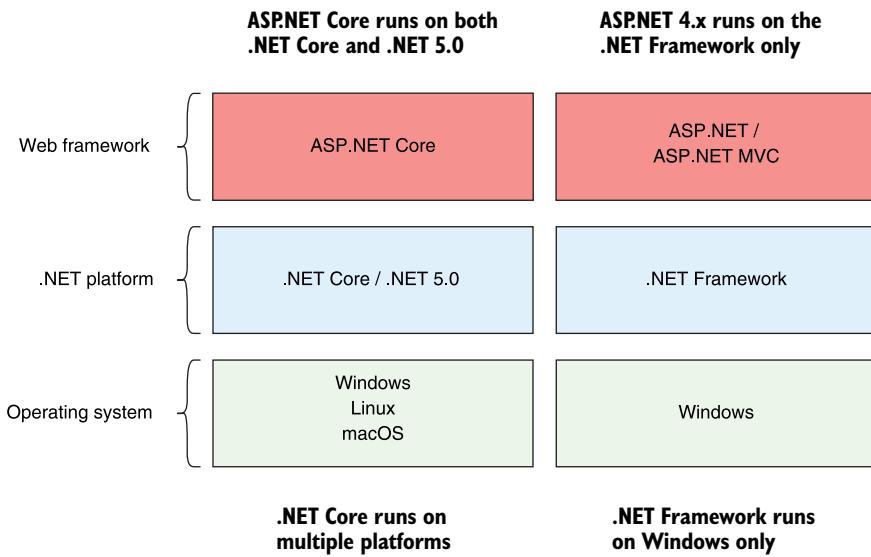
- To be run and developed cross-platform
- To have a modular architecture for easier maintenance
- To be developed completely as open source software
- To be applicable to current trends in web development, such as client-side applications and deployment to cloud environments

To achieve all these goals, Microsoft needed a platform that could provide underlying libraries for creating basic objects such as lists and dictionaries, and for 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 5.0), as shown in figure 1.3.

**DEFINITION** .NET 5.0 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 5.0 are often used interchangeably, but for consistency with Microsoft’s language, I use the term .NET 5.0 to refer to the latest version of .NET Core, and .NET Core when referring to previous versions.

.NET Core (and its successor, .NET 5.0) employs many of the same APIs as .NET Framework, but it’s more modular and only implements a subset of the features .NET Framework does, 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 5.0 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 1.4.

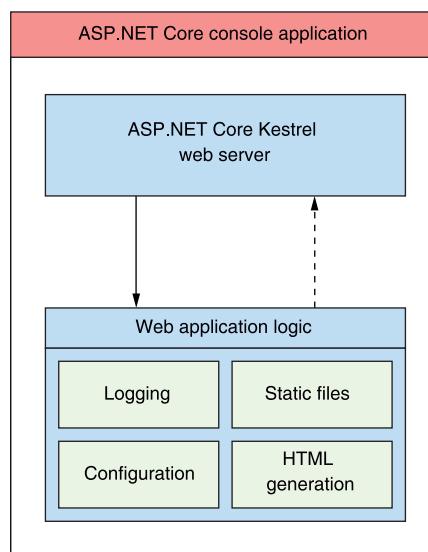


**Figure 1.3** The relationship between ASP.NET Core, ASP.NET, .NET Core/.NET 5.0, and .NET Framework. ASP.NET Core runs on .NET Core and .NET 5.0, so it can run cross-platform. Conversely, ASP.NET runs on .NET Framework only, so it is tied to the Windows OS.

**You write a .NET 5.0 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.**



**Figure 1.4** The ASP.NET Core application model. The .NET 5.0 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 5.0 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 there, 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 are covered by 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<sup>1</sup> has been redeveloped as Orchard Core<sup>2</sup> to run on ASP.NET Core. In contrast, the cloudscribe<sup>3</sup> CMS project (figure 1.5) was written specifically for ASP.NET Core from its inception.

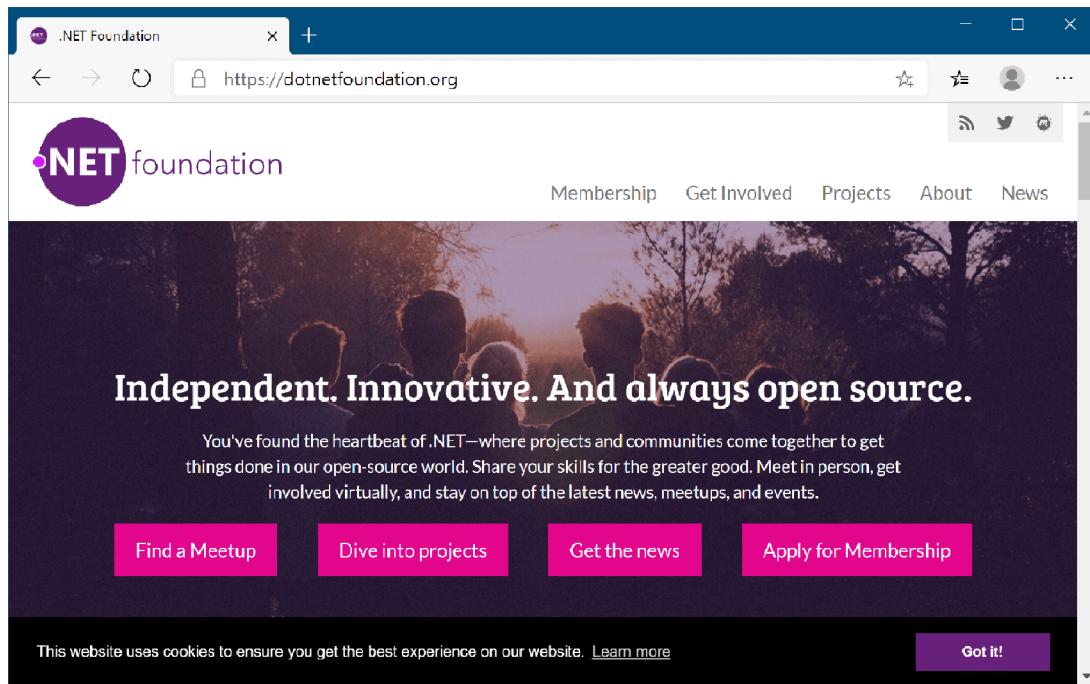


Figure 1.5 The .NET Foundation website (<https://dotnetfoundation.org/>) is built using the cloudscribe CMS and ASP.NET Core.

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.

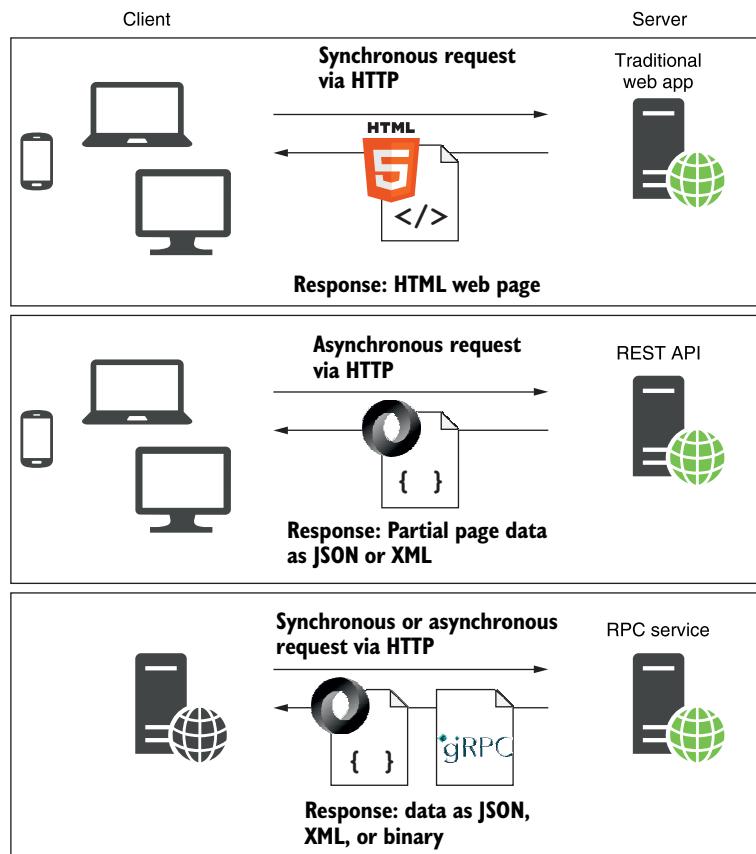
**DEFINITION** REST stands for *representational state transfer*. RESTful applications typically use lightweight and stateless HTTP calls to read, post (create/update), and delete data.

<sup>1</sup> The Orchard project source code is at <https://github.com/OrchardCMS>.

<sup>2</sup> Orchard Core ([www.orchardcore.net](http://www.orchardcore.net)). Source code at <https://github.com/OrchardCMS/OrchardCore>.

<sup>3</sup> The cloudscribe project ([www.cloudscribe.com](http://www.cloudscribe.com)). Source code at <https://github.com/cloudscribe>.

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 1.6. 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.



**Figure 1.6** ASP.NET Core can act as the server-side application for a variety of 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.

**NOTE** In this book I focus on building traditional, page-based server-side-rendered web applications and RESTful web APIs. I also show how to create “headless” worker services in chapter 22.

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 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.<sup>4</sup> And as all 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/.NET 5.0 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 5.0, you'll be getting a dependable platform as well as a full-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

---

<sup>4</sup> View the support policy at <https://dotnet.microsoft.com/platform/support/policy/dotnet-core>.

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, code-named Roslyn.<sup>5</sup>

**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 5.0 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, CentOS, 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 5.0, and Docker for Windows, that's all changing. A lightweight ASP.NET Core application running on the cross-platform .NET 5.0 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 5.0 runtime installed, 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

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

your production Linux machines. This compile-once, run-anywhere promise has finally been realized with ASP.NET Core and .NET Core/.NET 5.0.

### 1.2.3 **If you're a .NET Framework developer creating a new application**

If you're a .NET developer, 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 5.0, that is no longer a problem; Microsoft now explicitly advises that all new .NET applications should use .NET 5.0. Microsoft has pledged to provide bug and security fixes for the older ASP.NET framework, but it won't provide 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, the ability to build and deploy 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 5.0, makes all this possible.

.NET Core and .NET 5.0 are 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 that makes these APIs available in .NET 5.0.<sup>6</sup> They're only available when running .NET 5.0 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 that 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

---

<sup>6</sup> The Windows Compatibility Pack is designed to help port code from .NET Framework to .NET Core/.NET 5.0. See <https://docs.microsoft.com/dotnet/core/porting/windows-compat-pack>.

and handle requests directly, as shown in figure 1.7. This hosting model is conceptually much simpler and allows you to test and debug your applications from the command

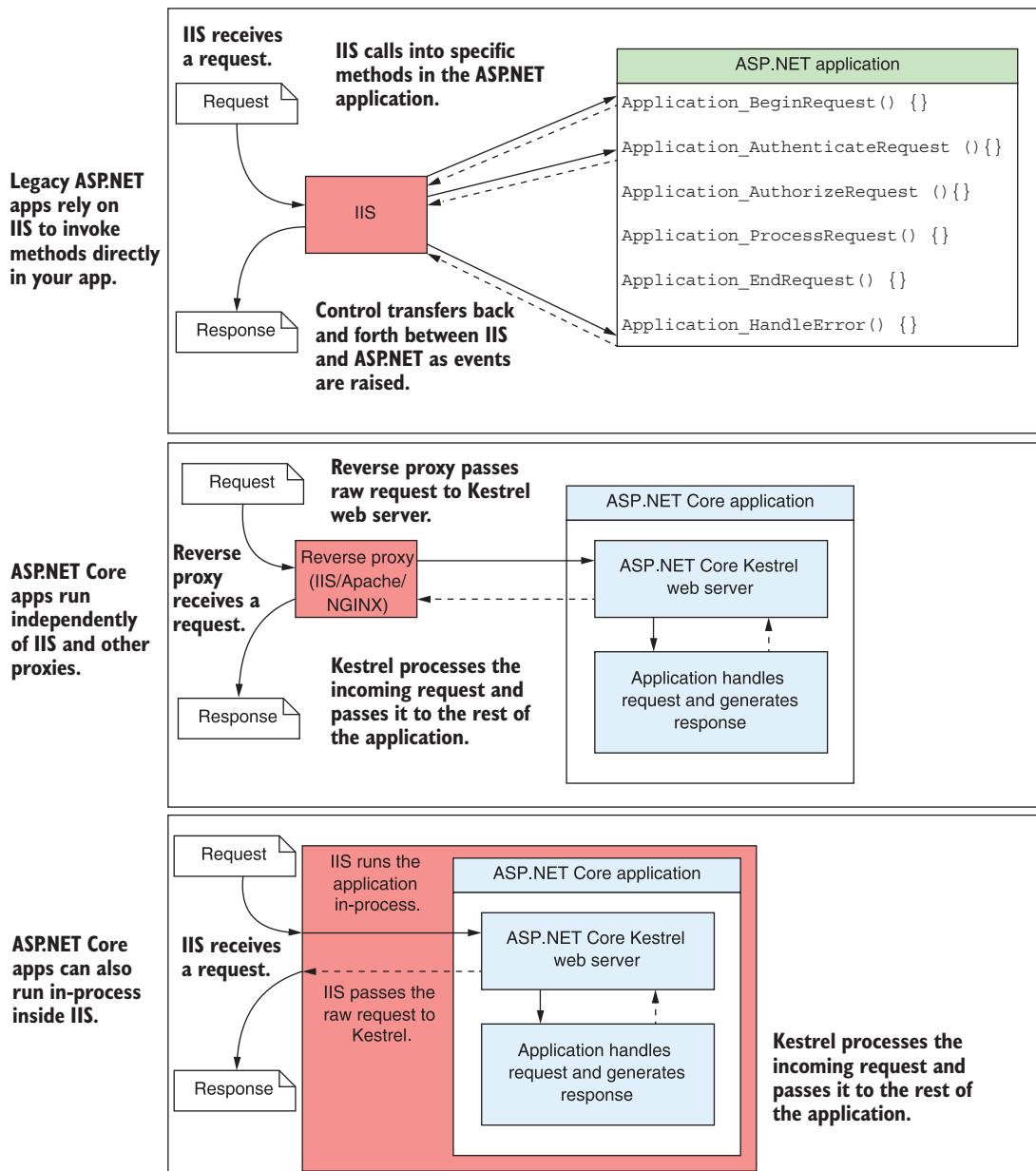


Figure 1.7 The difference between hosting models in ASP.NET (top) and ASP.NET Core (bottom). With the previous version of ASP.NET, IIS is tightly coupled with the application. The hosting model in ASP.NET Core is simpler; IIS hands off the request to a self-hosted web server in the ASP.NET Core application and receives the response, but has no deeper knowledge of the application.

line, though it doesn't necessarily remove the need to run IIS (or equivalent) in production, as you'll see in section 1.3.

**NOTE** You can also optionally run ASP.NET Core *inside* of IIS, as shown in figure 1.7, which can have performance benefits over the reverse-proxy version. This is primarily a deployment detail and doesn't change the way you build ASP.NET Core applications.

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 a 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 has focused on making the Kestrel HTTP server as fast as possible. TechEmpower ([www.techempower.com/benchmarks](http://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!<sup>7</sup>

### 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.

<sup>7</sup> 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.

Many of the performance improvements made to Kestrel did not come from the ASP.NET team members themselves, but from contributors to the open source project on GitHub.<sup>8</sup> 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 5.0, 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.<sup>9</sup> 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 5.0 platform.

**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 approach to releases, 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 with 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, there’s no significant reason not to use it. The largest obstacle you’re likely to come across is wanting 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

---

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

<sup>9</sup> The release schedule for .NET 5.0 and beyond: <https://devblogs.microsoft.com/dotnet/introducing-net-5/>.

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. 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, 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](http://www.asp.net/web-api) website.<sup>10</sup>

Windows Communication Foundation (WCF) is only partially supported in ASP.NET Core.<sup>11</sup> 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 you don't have a hard requirement on WCF itself, consider using gRPC instead. gRPC is a modern RPC framework with many concepts similar to WCF, and it's supported by ASP.NET Core out of the box.<sup>12</sup>

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

---

<sup>10</sup> An alternative approach would be to consider converting your application to Blazor using the community-driven effort to create Blazor versions of common WebForms components: <https://github.com/FritzAndFriends/BlazorWebFormsComponents>.

<sup>11</sup> You can find the client libraries for using WCF with .NET Core at <https://github.com/dotnet/wcf>.

<sup>12</sup> 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/>.

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, 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, moving to ASP.NET Core may well be a good choice.

## 1.3 How does ASP.NET Core work?

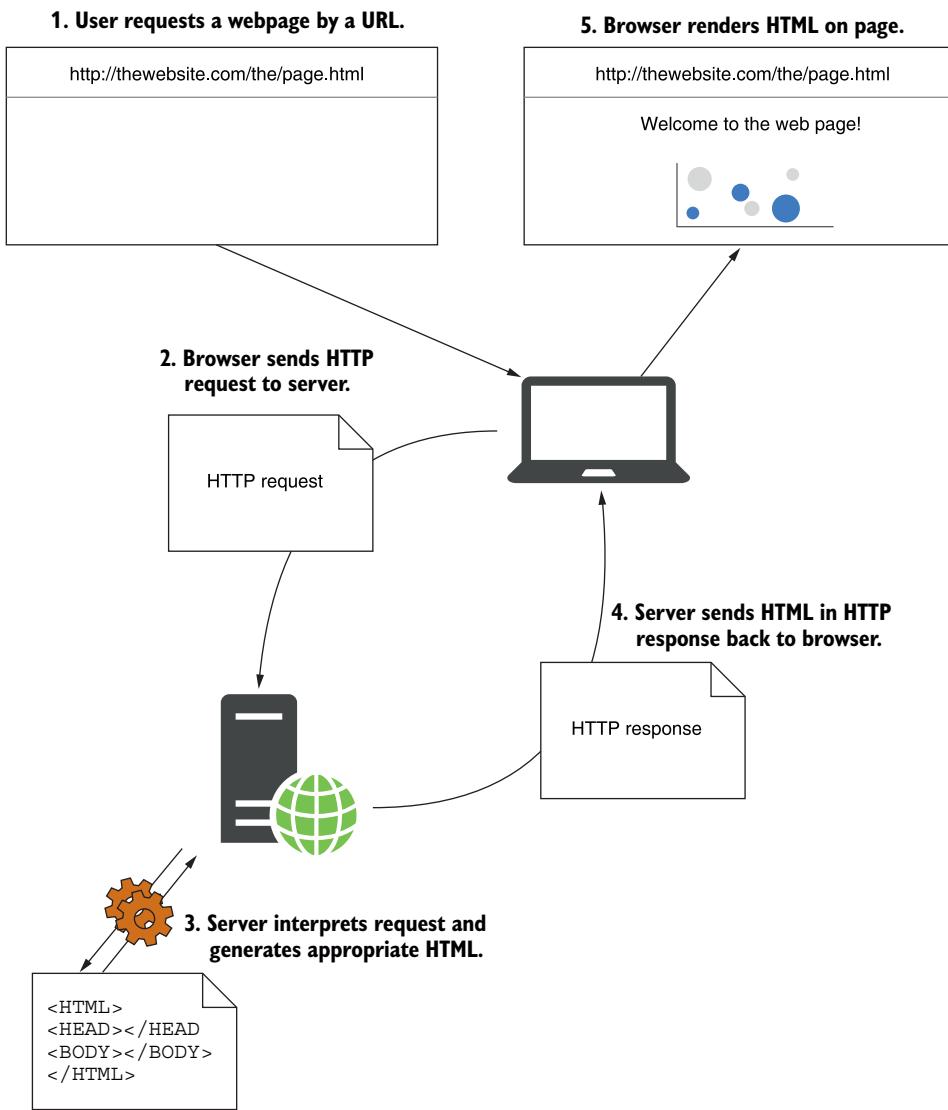
By now, you should have a good idea of what ASP.NET Core is and the sort of applications you should use it for. In this section, you'll see how an application built with ASP.NET Core works, from the user requesting a URL to a page being displayed in the browser. To get there, first you'll see how an 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 1.8.

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 the 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](http://microsoft.com), [www.google.co.uk](http://www.google.co.uk), and [facebook.com](http://facebook.com).



**Figure 1.8 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 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.

### A brief primer on HTTP

Hypertext Transfer Protocol (HTTP) is the application-level protocol that powers the web. It is a stateless request-response protocol, whereby a client machine sends a request to a server, which sends a response in turn.

Every HTTP request consists of a *verb* indicating the “type” of the request and a *path* indicating the resource to interact with. They typically also include *headers*, which are key-value pairs, and in some cases a *body*, such as the contents of a form, when sending data to the server.

An HTTP response contains a *status code*, indicating whether the request was successful, and optionally *headers* and a *body*.

For a more detailed look at the HTTP protocol itself, as well as more examples, see section 1.3 (“A quick introduction to HTTP”) in *Go Web Programming* by Sau Sheong Chang (Manning, 2016), <https://livebook.manning.com/book/go-web-programming/chapter-1/point-9018-55-145-1>.

Once the server receives the request, it will check that the request makes sense, and if it does, it 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 home page of a web app, 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.

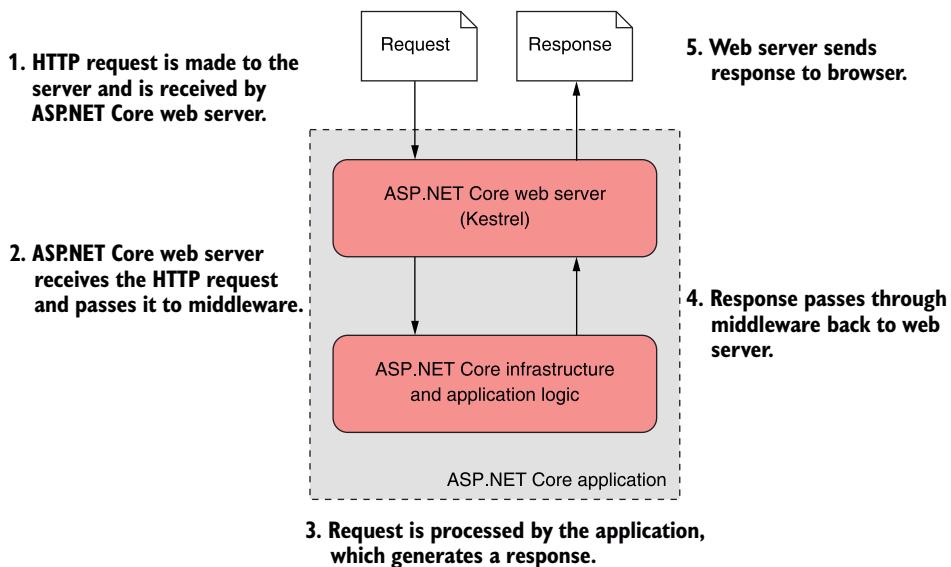
Pretty much all interactions that take place on the internet are a facade over this same basic process. A basic web page may only require a few simple requests to fully render, whereas a modern, large web page may take hundreds. At the time of writing, the Amazon.com homepage ([www.amazon.com](http://www.amazon.com)),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

Now that 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 that 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 1.9.



**Figure 1.9** 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 1.8—the user sent an HTTP request and received an HTTP response. All the differences are server-side, within your 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 remain a simple HTTP server not 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; the 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 will take 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. Appendix A includes details on setting up your editor or IDE and installing the .NET 5.0 SDK. Even though the examples in this book show Windows tools, everything you see can be achieved equally well on Linux or Mac platforms.

**TIP** You can install .NET 5.0 from <https://dotnet.microsoft.com/download>.

Appendix A contains further details on how to configure your development environment for working with ASP.NET Core and .NET 5.0.

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.

## 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 5.0 platform. You can access Windows-specific features such as the Windows Registry by using the Windows Compatibility Pack.
- .NET 5.0 is the next version of .NET Core after .NET Core 3.1.
- Fetching a web page involves sending an HTTP request and receiving an HTTP response.
- ASP.NET Core allows you to dynamically build 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.

# 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 you can use to start building applications.

**TIP** See appendix A for guidance on installing the .NET 5.0 SDK and choosing an editor/IDE.

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 you how to 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 you 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, you'll have similar templates available. I use Visual Studio 2019 and ASP.NET Core 5.0 with .NET 5.0 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 will be 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 you'll 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're for and how they're used. You'll see how to define the middleware pipeline for your application and how you can customize it.

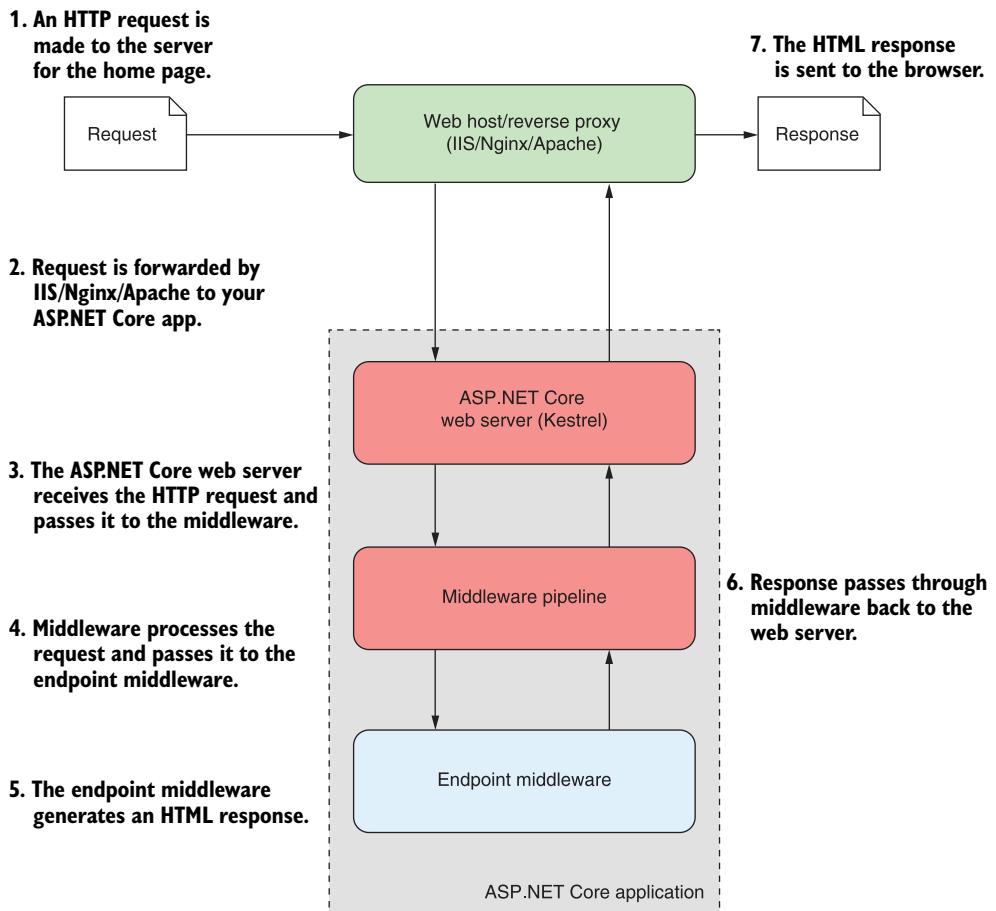
Finally, you'll see how the app generates HTML in response to a request, looking at each of the components that make up the Razor Pages endpoint. You'll see how it controls what code is run in response to a request and how to define the HTML that should be returned for a particular request.

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, 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; then, later, you could add dynamic features to your app. Figure 2.1 shows how the application would handle a request for a page in your application.



**Figure 2.1** An overview of an ASP.NET Core application. The ASP.NET Core application receives an incoming HTTP request from the browser. Every request passes to the middleware pipeline, which potentially modifies it and 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.<sup>1</sup>

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 process 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, 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.

---

<sup>1</sup> 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, and 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 5.0 with .NET 5.0.

Getting an application up and running typically involves 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.

Visual Studio and the .NET CLI include many ASP.NET Core templates for building different types of applications. For example,

- *Razor Pages web application*—Razor Pages applications generate HTML on the server and are designed to be viewed by users in a web browser directly.
- *MVC (Model-View-Controller) application*—MVC applications are similar to Razor Pages apps in that they generate HTML on the server and are designed to be viewed by users directly in a web browser. They use traditional MVC controllers instead of Razor Pages.
- *Web API application*—Web API applications return data in a format that can be consumed by single-page applications (SPAs) and APIs. They are typically used in conjunction with client-side applications like Angular and React.js or mobile applications.

We will look at each of these application types in this book, but in this chapter we will focus on the Razor Pages template.

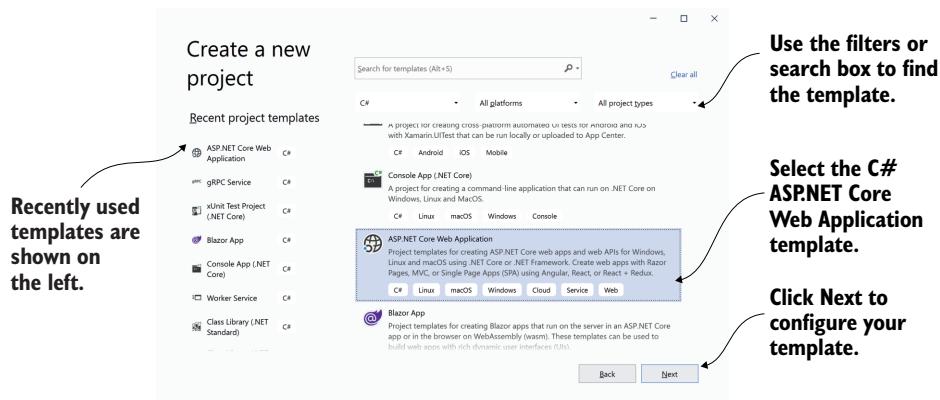
### 2.2.1 **Using a template to get started**

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

**TIP** In .NET, a *project* is a unit of deployment, which will be compiled into a .dll file or an executable, for example. Each separate app is a separate project. Multiple projects can be built and developed at once in a *solution*.

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.



**Figure 2.2** The new project dialog box. Select the C# ASP.NET Core Web Application template from the list on the right 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 solution name, and click Create, as shown in figure 2.3. For example, use WebApplication1 as both the project and solution name.
- 4 On the following screen (figure 2.4):
  - Ensure .NET Core is selected.
  - Select ASP.NET Core 5.0. If this option isn't available, ensure you have .NET 5.0 installed. See appendix A for details on configuring your environment.
  - Select ASP.NET Core Web App to create a Razor Pages web application.
  - 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.

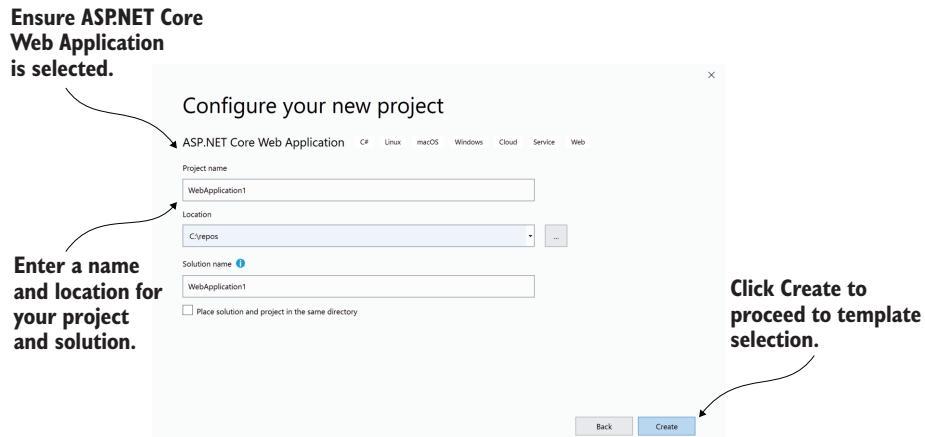


Figure 2.3 The Configure Your New Project dialog box. To create a new .NET 5.0 application, select ASP.NET Core Web Application from the template screen. On the following screen, enter a project name, location, and solution name, and click Create.

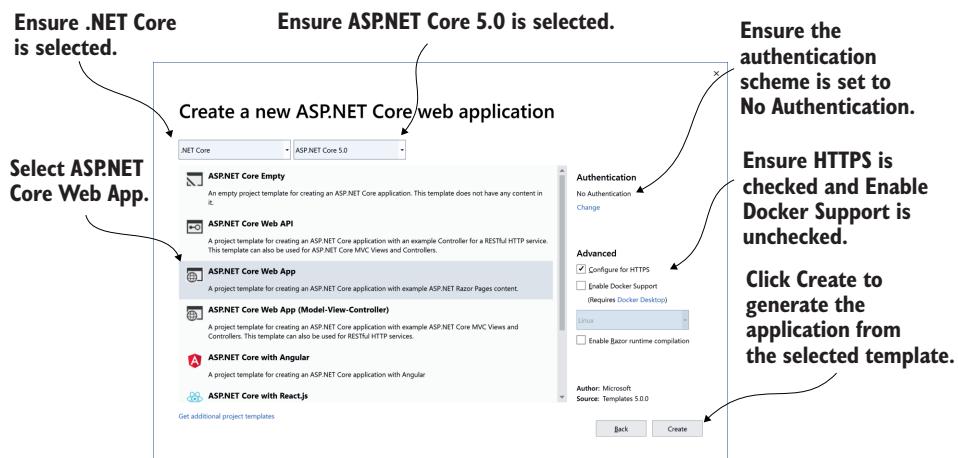
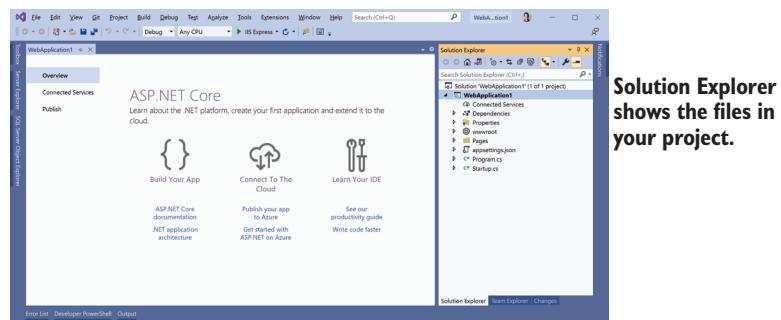


Figure 2.4 The web application template screen. This screen follows on from the Configure Your Project dialog box and lets you customize the template that will generate your application. For this starter project, you'll create a Razor Pages web application without authentication.

- 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.



**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.

If you’re not using Visual Studio, you can create a similar template using the .NET CLI. Create a folder to hold your new project. Open a PowerShell or cmd prompt in the folder (on Windows) or a terminal session (on Linux or macOS) and run the commands in the following listing.

#### Listing 2.1 Creating a new Razor Page application with the .NET CLI

```
dotnet new sln -n WebApplication1
dotnet new webapp -o WebApplication1
dotnet sln add WebApplication1
```

Add the new project to the solution file.

Create a solution file called `WebApplication1` in the current folder.

Create a Razor Pages project in a subfolder, `WebApplication1`.

Whether you use Visual Studio or the .NET CLI, you now have the basic files required to build and run your first ASP.NET Core application.

### 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 command-line interface

One of the foundational components of .NET 5.0 cross-platform development is the .NET command-line interface (CLI). This provides several basic commands for creating, building, and running .NET 5.0 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 used 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.*

Most ASP.NET Core applications have dependencies on various 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.

The restore process typically happens implicitly when you build or run your application, but it can sometimes be useful to run it explicitly, in continuous-integration build pipelines, for example.

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 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

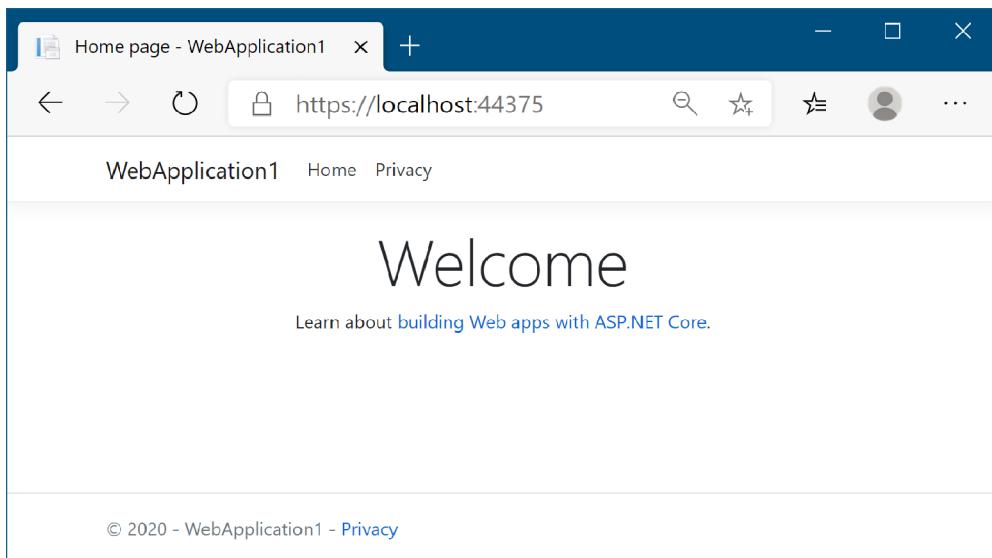
```
dotnet --help
```

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.



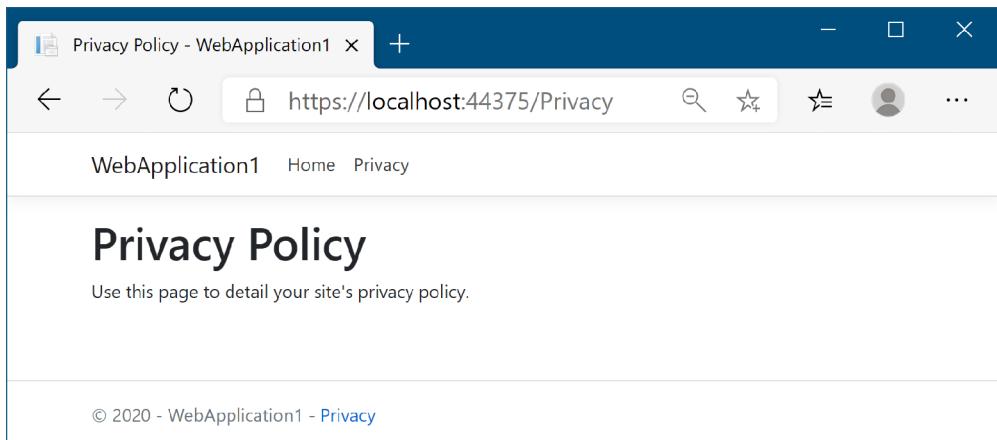
**Figure 2.6** The home page of your new ASP.NET Core application. When you run it from Visual Studio, IIS Express chooses a random port by default. If you're running from the command line with `dotnet run`, your application will be available at `http://localhost:5000` and `https://localhost:5001`.

**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.<sup>2</sup> 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

<sup>2</sup> You can install the development certificate on Windows and macOS. For instructions on trusting the certificate on Linux, see your distribution's instructions. Not all browsers (Mozilla Firefox, for example) use the certificate store, so follow your browser's guidelines for trusting the certificate. If you still have difficulties, see the troubleshooting tips at <http://mng.bz/1rmy>.

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 can use Razor Pages in your application to define these two pages and build the HTML they display.



**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.

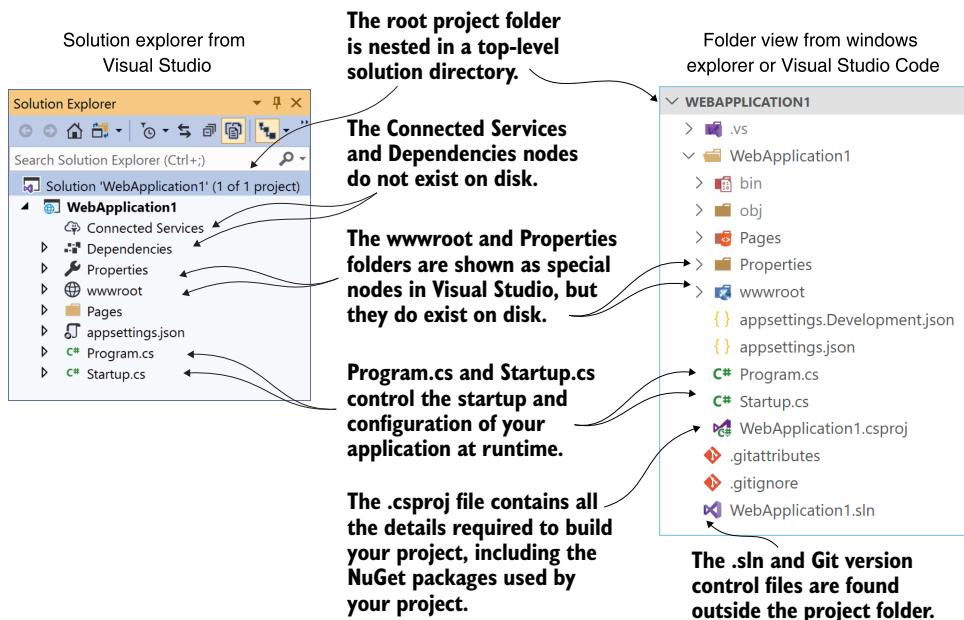


Figure 2.8 The Solution Explorer and folder on disk for a new ASP.NET Core application. The Solution Explorer also displays the Connected Services and Dependencies nodes, which list NuGet and other dependencies, though the folders themselves don't exist on disk.

The first thing to notice is that the main project, WebApplication1, is nested in a top-level directory with the name of the solution, which is 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,<sup>3</sup> 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 additional .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 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

<sup>3</sup> 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 “Think Like (a) Git,” <http://think-like-a-git.net/>. It helped me achieve Git enlightenment.

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 corresponding folders 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 that 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 project 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 5.0, 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, globally unique identifiers (GUIDs) were used for many things, but 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 .dll files*—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.config file. Now you can reference the NuGet package directly in your .csproj, and you 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 2.2 The .csproj project file, showing SDK, target framework, and references

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
  </PropertyGroup>
</Project>
```

The **SDK** attribute specifies the type of project you're building.

The **TargetFramework** is the framework you'll run on, in this case, .NET 5.0.

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 5.0, this will be `net5.0`.

**TIP** With the new csproj style, Visual Studio users can double-click a project in Solution Explorer to edit the `.csproj` file 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 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 existing reusable libraries.

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 publish (and host) NuGet packages to repositories other than <https://nuget.org>—see <https://docs.microsoft.com/nuget> for details.

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 next, 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>
```

If you’re using Visual Studio, you can manage packages with the NuGet Package Manager by right-clicking the solution name or a project and choosing Manage NuGet Packages.

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 <http://mng.bz/PPGg>.

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.

**TIP** .NET 5.0 and C# 9 introduced “top-level statements,” which implicitly create the `Main` entry point. I don’t use top-level statements in this book, but they are supported in ASP.NET Core 5.0. See the documentation for details: <http://mng.bz/JDaP>.

In ASP.NET Core applications, the `Main` entry point is 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 2.3 The default Program.cs file configures and runs an IWebHost**

```

public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args)
            .Build()
            .Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}

```

**Run the IHost and start listening for requests and generating responses.**

**Create an IHostBuilder using the CreateHostBuilder method.**

**Build and return an instance of IHost from the IHostBuilder.**

**Create an IHostBuilder using the default configuration.**

**The Startup class defines most of your application's configuration.**

**Configure the application to use Kestrel and listen to HTTP requests.**

The `Main` function contains all the basic initialization code required to create a web server and to start listening for requests. It uses an `IHostBuilder`, created by the call to `CreateDefaultBuilder`, to define how the `IHost` is configured, before instantiating the `IHost` 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* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (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 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.

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

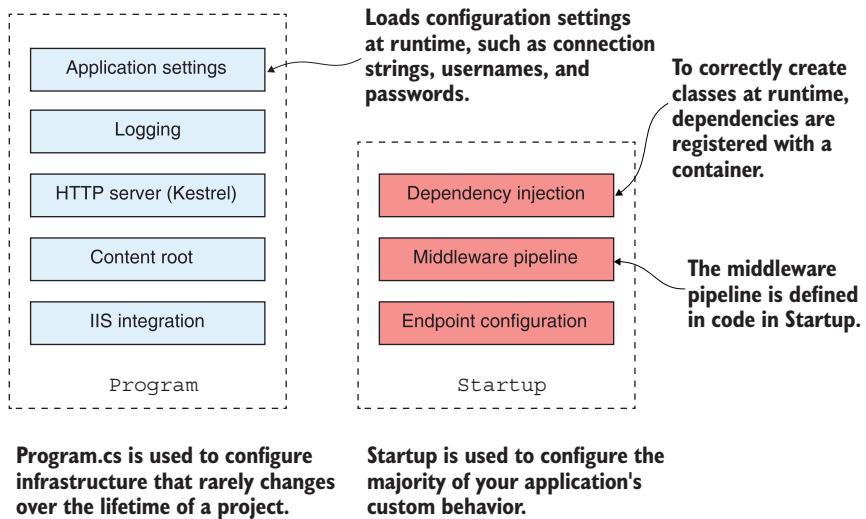


Figure 2.9 The difference in configuration scope for `Program` and `Startup`. `Program` is concerned with infrastructure configuration that will typically remain stable throughout the lifetime of the project. In contrast, you'll often modify `Startup` to add new features and to update application behavior.

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. `CreateDefaultBuilder` 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

**(continued)**

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 22 you'll see how to build applications using the generic host to run scheduled tasks and build 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.

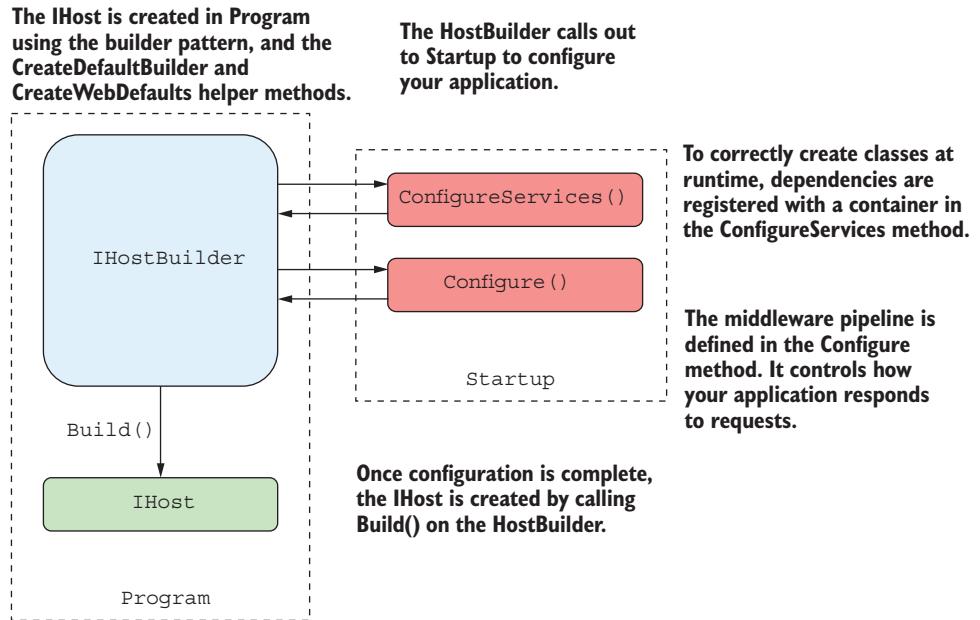
**Listing 2.4 An outline of `Startup.cs` showing how each aspect is configured**

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

Configure services by registering them with the `IServiceCollection`.

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`.



**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.

Because the `Startup` class is fundamental 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's 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 it's 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. These 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, 2013).

`OrderTotalCalculatorService` needs access to an instance of `ShippingCostService` and `TaxCalculator`. A naive 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 it 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 in the following listing, taken from the example application.

#### **Listing 2.5 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, formatting 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 Tax-Calculator discussed previously. `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` classes, you've defined the infrastructure of the application and registered your services with the IoC container. In the final configuration method of the `Startup` class, `Configure`, you define the middleware pipeline for the application, which defines how your app handles HTTP requests. Here's the `Configure` method for the template application.

**Listing 2.6 Startup.Configure: defining the middleware pipeline**

```

public class Startup
{
    public void Configure(
        IApplicationBuilder app,
        IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
        }
        else
        {
            app.UseExceptionHandler("/Error");
            app.UseHsts();
        }

        app.UseHttpsRedirection();
        app.UseStaticFiles();
        app.UseRouting();
        app.UseAuthorization();

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

```

The diagram illustrates the sequence of middleware setup in the `Configure` method:

- IAplicationBuilder is used to build the middleware pipeline.**
- Other services can be accepted as parameters.**
- Different behavior when in development or production**: The code checks the environment (`env.IsDevelopment()`) and applies different configurations based on the result.
- Only runs in a development environment**: The `UseDeveloperExceptionPage()` call is executed only in development.
- Only runs in a production environment**: The `UseExceptionHandler("/Error")` and `UseHsts()` calls are executed only in production.
- Adds the static-file middleware**: The `UseStaticFiles()` call adds middleware to serve static files.
- Adds the endpoint routing middleware, which determines which endpoint to execute**: The `UseRouting()` call adds middleware to handle endpoint routing.
- Adds the authorization middleware, which can block access to specific pages as required**: The `UseAuthorization()` call adds middleware to handle user authentication and authorization.
- Adds the endpoint middleware, which executes a Razor Page to generate an HTML response**: The `UseEndpoints` and `MapRazorPages` calls add middleware to handle requests for Razor Pages.

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 in which 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 `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.

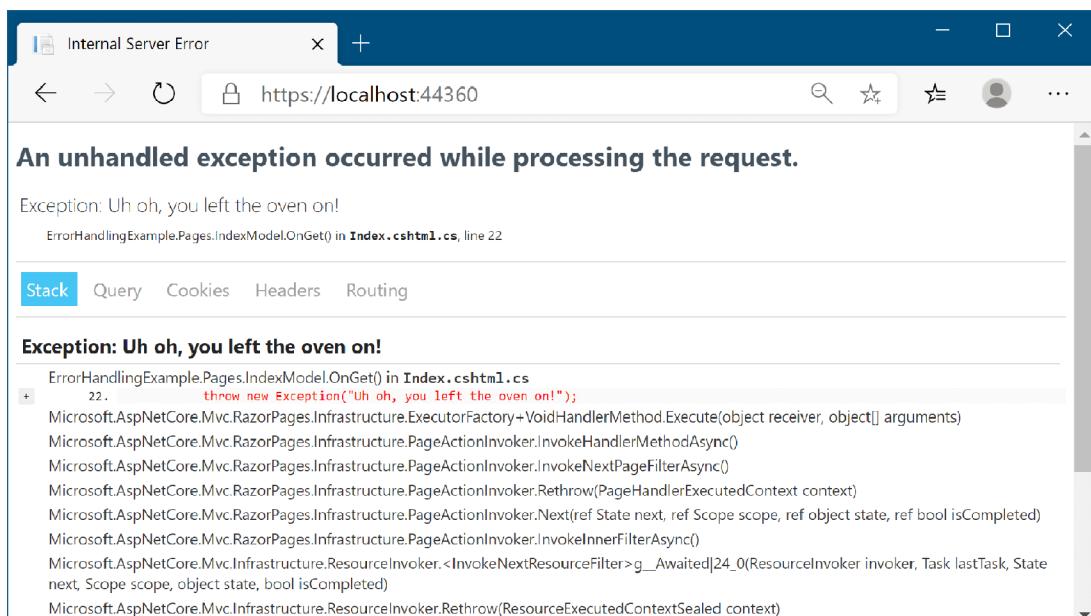


Figure 2.11 The developer exception page contains many different sources of information to help you diagnose a problem, including the exception stack trace and details about the request that generated the exception.

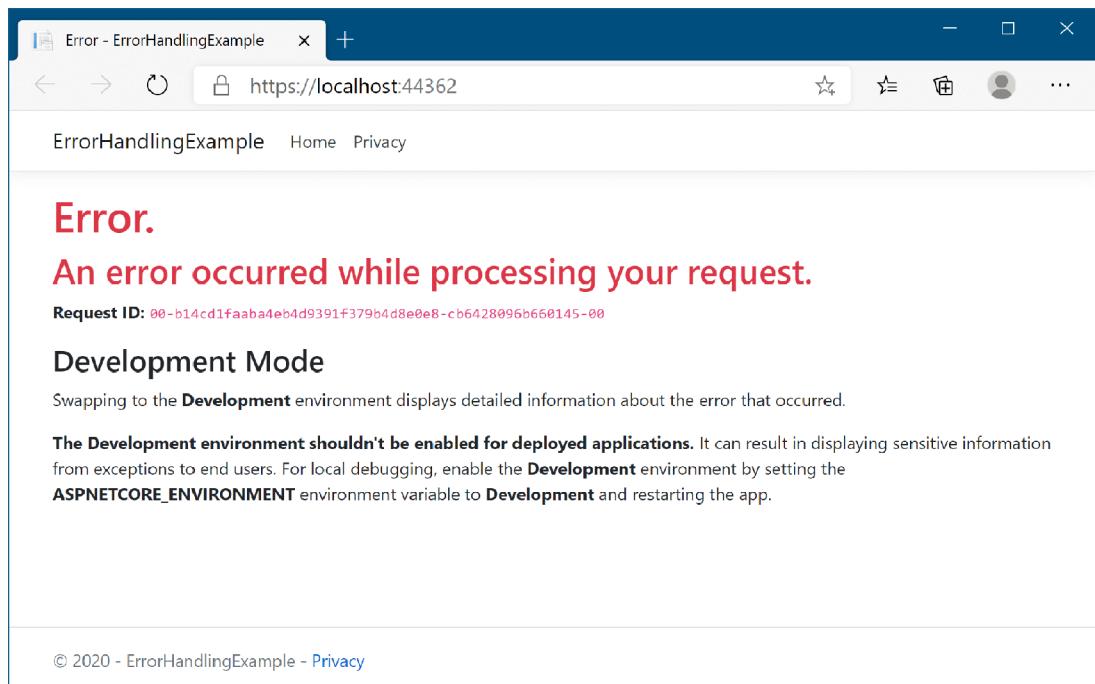
**NOTE** The default templates also add `HstsMiddleware` in production, which sets security headers in your response, in line with industry best practices. See chapter 18 for details about this and other security-related middleware.

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, 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.

The next piece of middleware added to the pipeline is `HttpsRedirectionMiddleware`, using this 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.

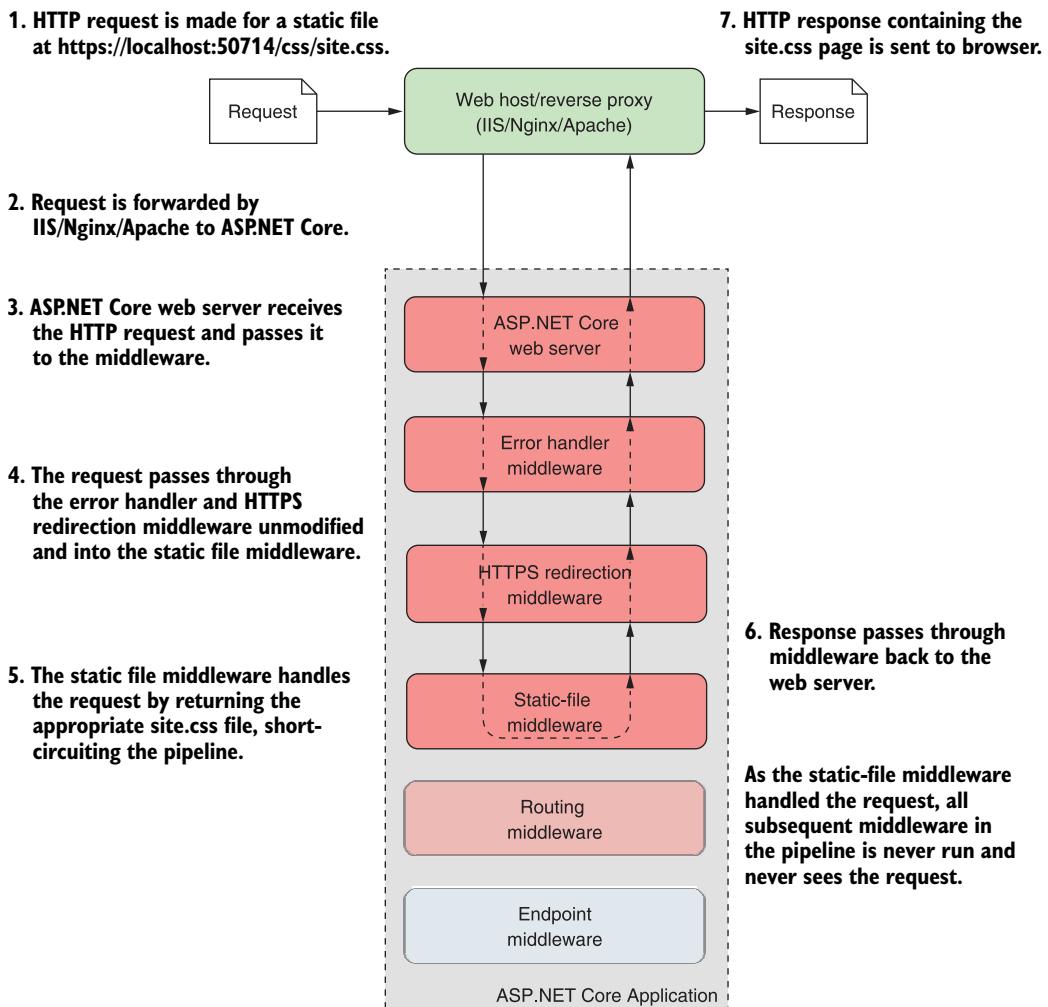


**Figure 2.12** The default exception-handling page. In contrast to the developer exception page, this one doesn’t reveal any details about your application to users. In reality, you’d update the message to something more user-friendly.

StaticFileMiddleware is added to the pipeline next with 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, 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



**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.

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.

Now we come 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. 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.

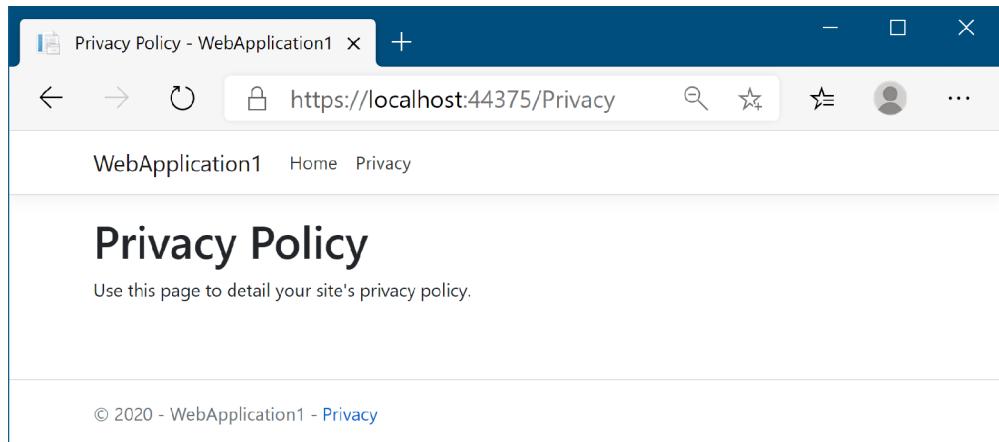
Once the application is configured, it can start handling requests. But *how* does it handle them? I've already touched on `StaticFileMiddleware`, which will serve the image and CSS files to the user, but what about the requests that require an HTML response? In the rest of this chapter, I'll give you 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](http://www.microsoft.com/account/manage), the path is `/account/manage`.

Once a Razor Page has been selected, the routing middleware notes 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.



**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 following listing.

#### Listing 2.7 The Privacy.cshtml Razor Page

```

    Indicates that this is a Razor Page
@page
    ←

    Links the Razor Page to
    a specific PageModel
@model PrivacyModel
    ←

    C# code that doesn't
    write to the response
@{
    ViewData["Title"] = "Privacy Policy";
}
    ←

    HTML with dynamic C# values
    written to the response
<h1>@ViewData["Title"]</h1>
    ←

    Standalone,
    static HTML
<p>Use this page to detail your site's privacy policy.</p>
    ←

```

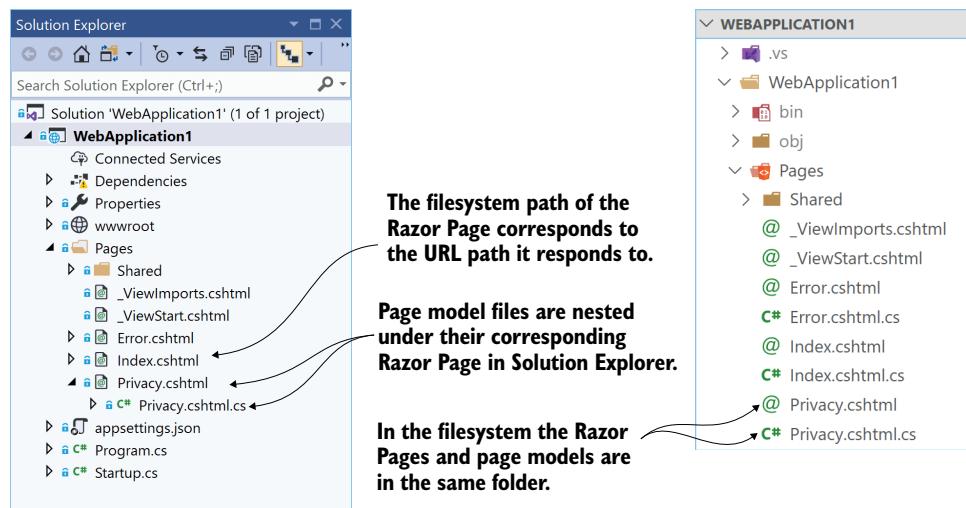
Razor Pages use a templating syntax, called *Razor*, that combines static HTML with dynamic C# code and HTML generation. The @page directive on the first line of the

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 files underneath the Razor Page `.cshtml` files in Solution Explorer. We'll look at the page model in the next section.



**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 that you can dynamically write 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 in listing 2.7 when it's 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 from having to duplicate code for the header and footer in every page, and it 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'll want to limit the code in your .cshtml file 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 that has a .cs extension, as you saw in figure 2.15. Page models should derive from the `PageModel` base class, and they 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 following listing shows the page model for the `Privacy.cshtml` Razor Page, found in the file `Privacy.cshtml.cs`.

**Listing 2.8 The `PrivacyModel` in `Privacy.cshtml.cs`—a Razor Page page model**

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

    public void OnGet()    ←
    {
    }
}
```

Razor Pages must inherit from `PageModel`.

You can use dependency injection to provide services in the constructor.

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` that contains other instructions for generating the response, such as redirecting the user to a different page.

**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 resource from the server so you can view it. The second most common verb is `POST`, which is used to *send* data to the server, such as 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.

**NOTE** Razor Pages are focused on building page-based apps, so you typically want to return HTML rather than JSON or XML. However, you can also use an `IActionResult` to return any sort of data, to redirect users to a new page, or to send an error. You'll learn more about `IActionResult`s in chapter 4.

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, a file, or 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 instances 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.

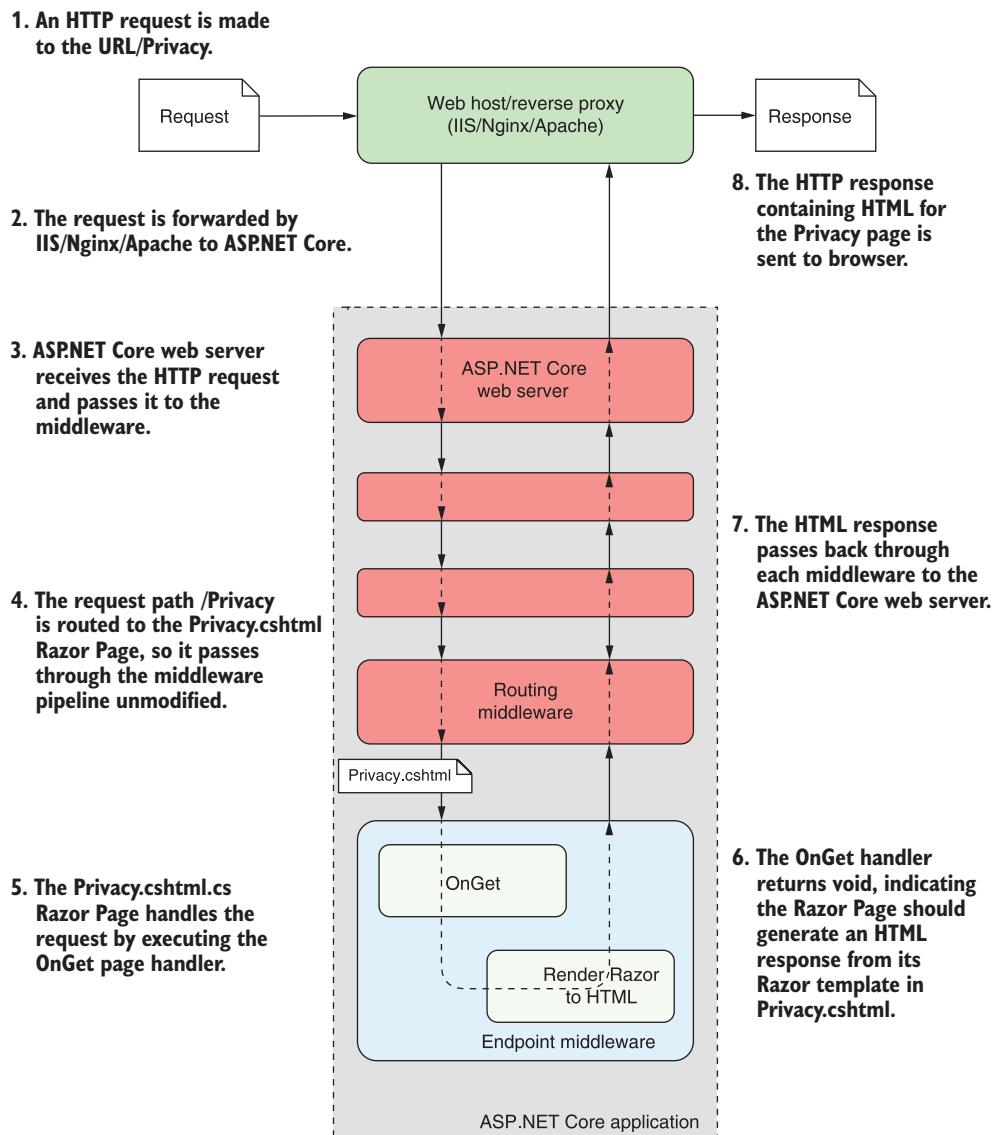
Clearly, the `PrivacyModel` page model does not do much in this case, and you may be wondering why it's worth having. If all they do is tell the Razor Page to generate HTML, then why do we need page models at all?

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, let's 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.

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.



**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 the `OnGet` handler of the `Privacy.cshtml.cs` Razor Page. The Razor Page generates an HTML response by executing the Razor template in `Privacy.cshtml` and passes the response back through the middleware pipeline to the browser.

## 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.

# *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'll focus in on one small subsection: the middleware pipeline.

In ASP.NET Core, middleware are C# classes or functions that handle an HTTP request or response. They are chained together, with the output of one middleware acting as the input to the next middleware, to form a 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. Because all requests pass through the middleware pipeline, it's the preferred location to configure and handle this functionality.

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. In this example you'll explore the importance of ordering in the middleware pipeline and 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 they are one of the classic cross-cutting concerns, and middleware is well placed to provide the required functionality. In section 3.3 I'll show how you can handle exceptions and errors using 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<sup>1</sup> 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 (for Web API applications) 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.

**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 for 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

---

<sup>1</sup> Technically middleware just needs to be a *function*, as you'll see in chapter 19, but it's very common to implement middleware as a C# class with a single method.

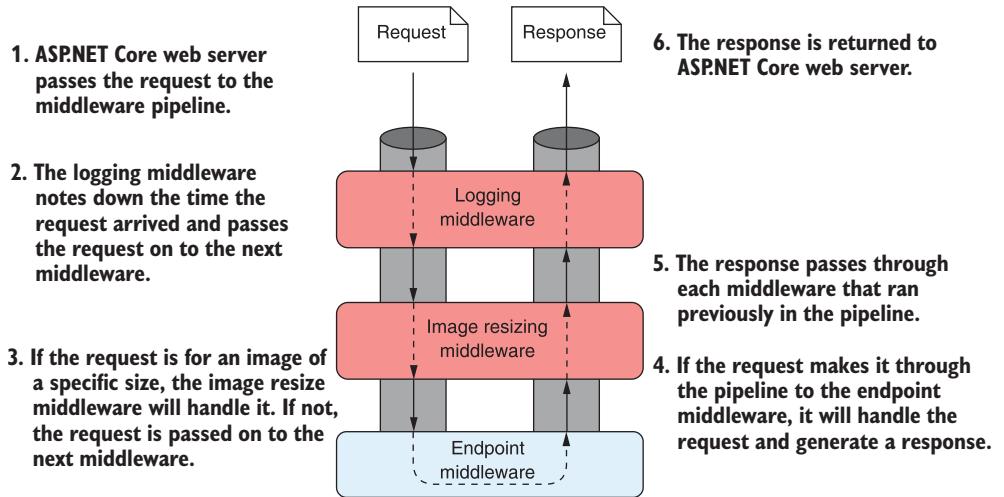


Figure 3.1 Example of a middleware pipeline. Each middleware handles the request and passes it on to the next middleware in the pipeline. After a middleware generates a response, it passes the response back through the pipeline. When it reaches the ASP.NET Core web server, the response is sent to the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

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.

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.

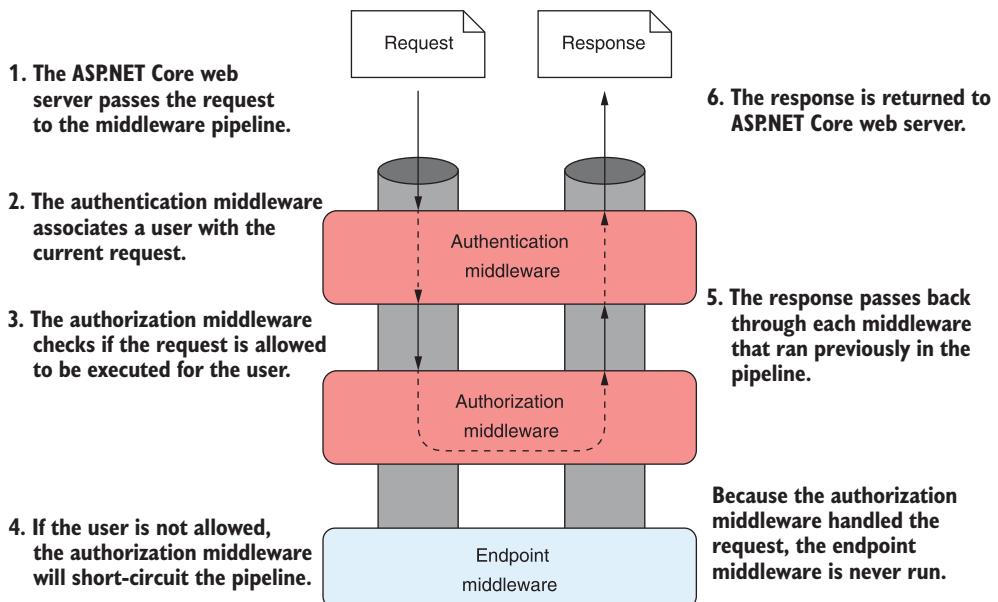


Figure 3.2 Example of a middleware component modifying a request for use later in the pipeline. Middleware can also short-circuit the pipeline, returning a response before the request reaches later middleware.

### The `HttpContext` object

I mentioned the `HttpContext` in chapter 2, and it's sitting behind the scenes here too. The ASP.NET Core web server constructs an `HttpContext` for each request, 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, to identify 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.

You can also think of your middleware pipeline as being a series of concentric components, similar to a traditional matryoshka (Russian) doll, as shown in figure 3.3. A request progresses “through” the pipeline by heading deeper into the stack of middleware until a response is returned. The response then returns through the middleware, passing through them in the reverse order to the request.

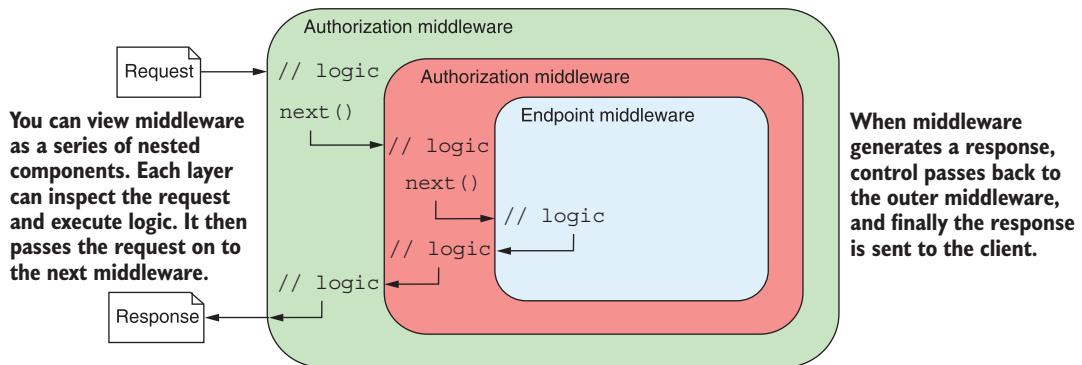


Figure 3.3 You can also think of middleware as being a series of nested components, where a request is sent deeper into the middleware, and the response resurfaces out of it. Each middleware can execute logic before passing the response on to the next middleware and can execute logic after the response has been created, on the way back out of the stack.

### 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

**(continued)**

.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 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 3.4. 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.

**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 you've seen before occurs when the application receives an HTTP request, as shown in figure 3.5.

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.

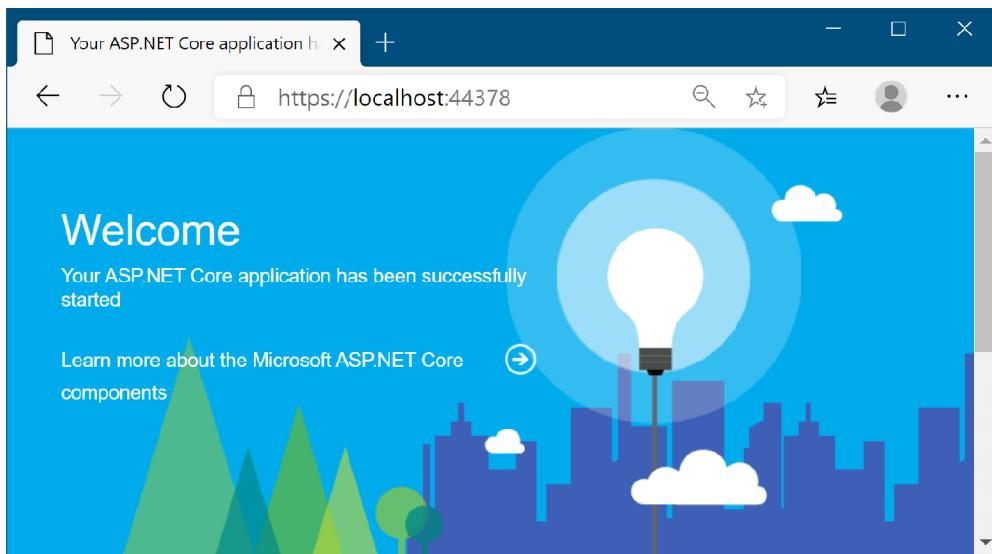


Figure 3.4 The Welcome page middleware response. Every request to the application, at any path, will return the same Welcome page response.

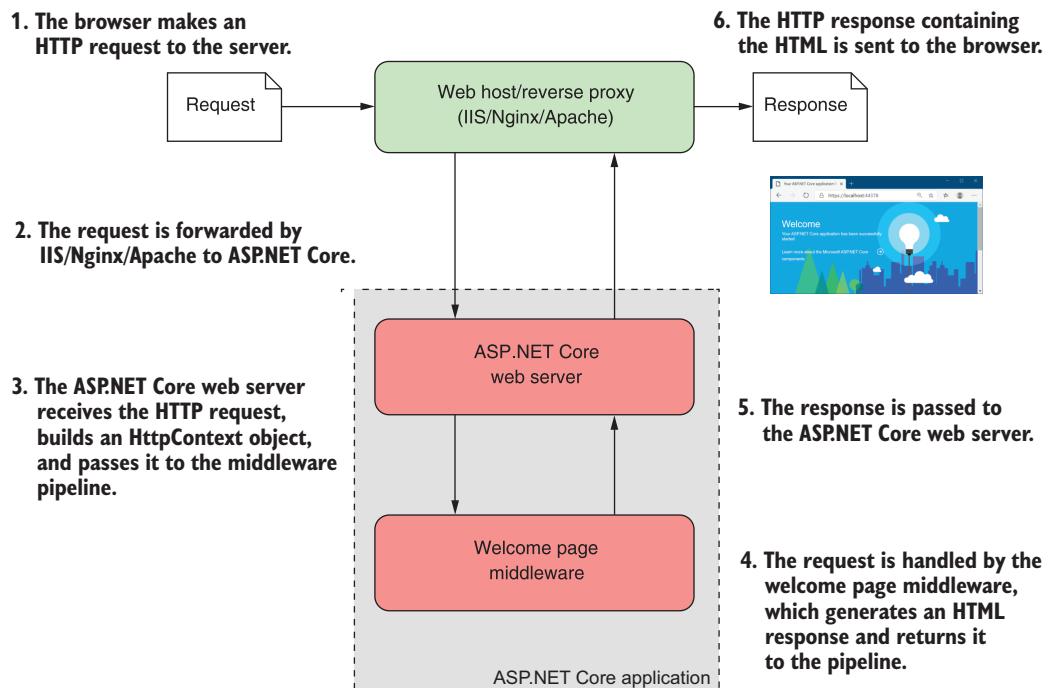


Figure 3.5 `WelcomePageMiddleware` handles a request. The request passes from the reverse proxy to the ASP.NET Core web server and, finally, to the middleware pipeline, which generates an HTML response.

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
    {
        public void Configure(IApplicationBuilder app)
        {
            app.UseWelcomePage();
        }
    }
}
```

The diagram shows three annotations pointing to specific parts of the code:

- The Startup class is very simple for this basic application.** Points to the `Startup` class definition.
- The Configure method is used to define the middleware pipeline.** Points to the `Configure` method definition.
- The only middleware in the pipeline** points to the `app.UseWelcomePage();` line.

As you can see, the `Startup` class 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.<sup>2</sup> 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 3.6.

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

---

<sup>2</sup> The downside to this approach is that it can hide exactly which middleware is being added to the pipeline. When the answer isn't clear, I typically search for the source code of the extension method directly on GitHub <https://github.com/aspnet/aspnetcore>.

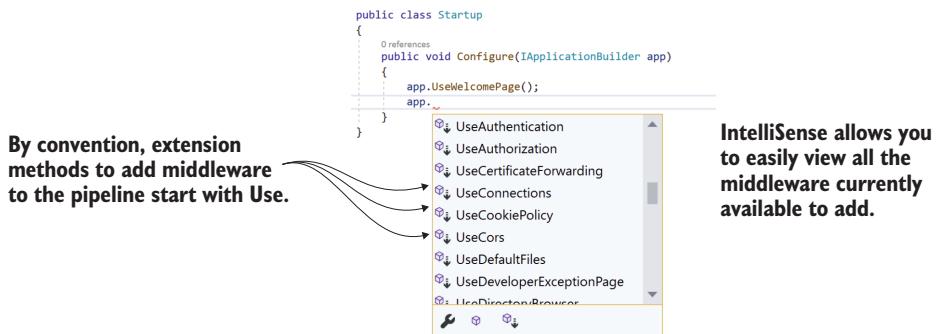


Figure 3.6 IntelliSense makes it easy to view all the middleware available to add to your middleware pipeline.

`IApplicationBuilder` in `Configure` defines the order in which 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 different responses when you make requests to different paths.

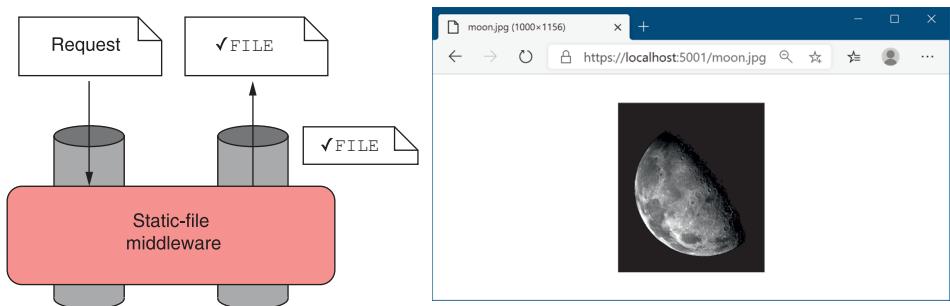
### 3.2.2 Simple pipeline scenario 2: Handling static files

In this section, I'll show you 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.

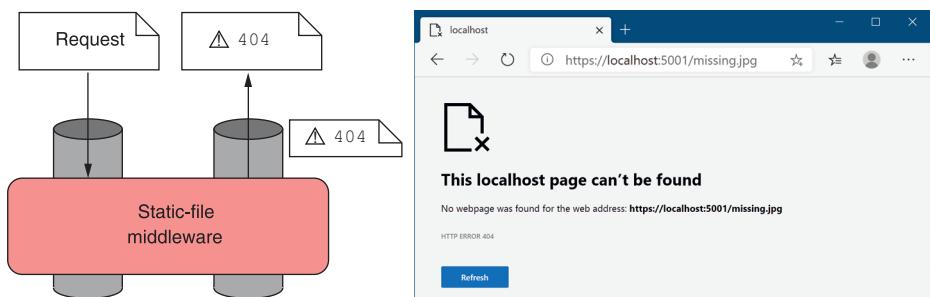
For now, you'll use `StaticFileMiddleware` to create an application that only serves static files from the `wwwroot` folder when requested, as shown in figure 3.7. In this example, an image called `moon.jpg` exists in the `wwwroot` folder. When you request the file using the `/moon.jpg` path, it's loaded and returned as the response to the request.

If the user requests a file that doesn't exist in the `wwwroot` folder, such as `missing.jpg`, 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 3.8.



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 3.7 Serving a static image file using the static-file middleware



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

Figure 3.8 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.

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

Building the middleware pipeline for this application is easy. It consists 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
    {
        public void Configure(IApplicationBuilder app)
        {
            app.UseStaticFiles();
        }
    }
}
```

The Startup class is very simple for this basic static-file application.

The Configure method is used to define the middleware pipeline.

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 3.9.

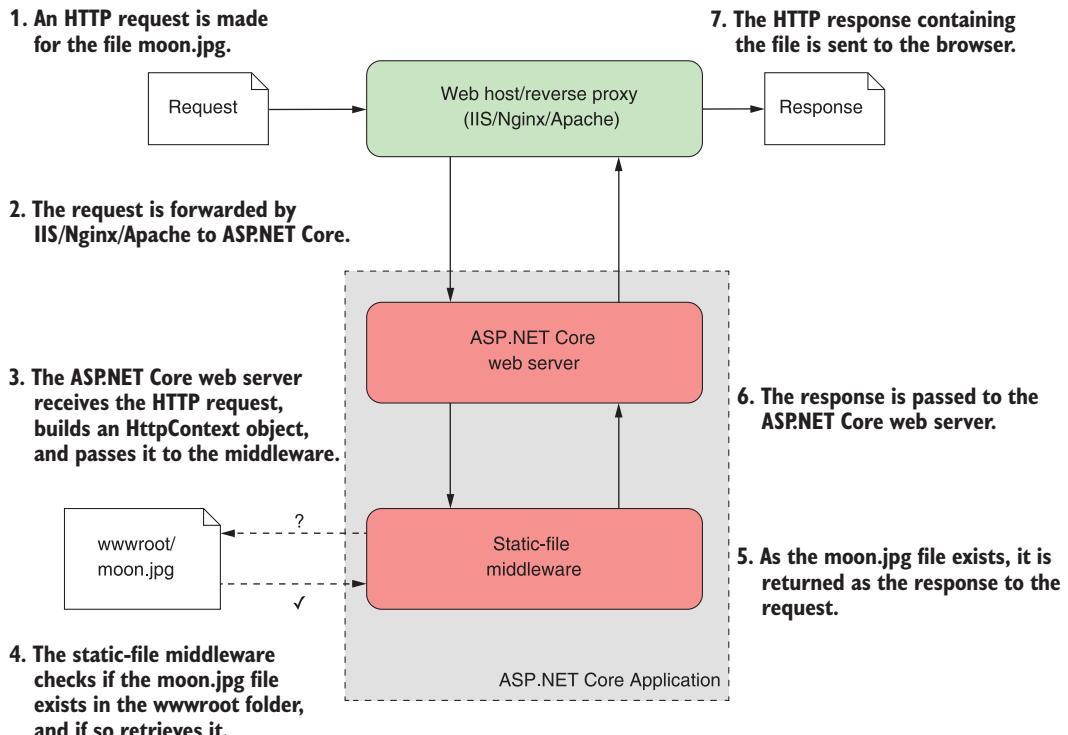


Figure 3.9 StaticFileMiddleware handles a request for a file. The middleware checks the wwwroot folder to see if the requested moon.jpg file exists. The file exists, so the middleware retrieves it and returns it as the response to the web server and, ultimately, to the browser.

If the file doesn't exist, 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 a "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.

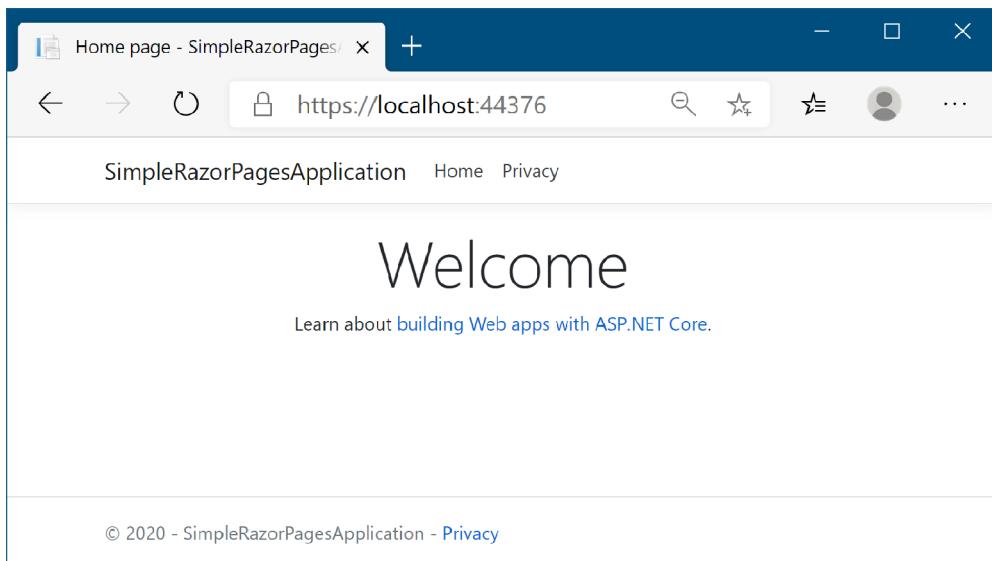
Error codes are found in the 4xx and 5xx classes. Common codes include a 404 response when a file couldn't be found, a 400 error when a client sends invalid data (an invalid email address for example), and a 500 error when an error occurs on the server. HTTP responses for error codes may or may not include 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 you'll see how to combine multiple middleware as we look at a simple Razor Pages application.

### 3.2.3 Simple pipeline scenario 3: A Razor Pages application

By this point, you should 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 several standard middleware components to form a pipeline. 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 home page of the application is shown in figure 3.10—identical to the sample application shown in chapter 2.



**Figure 3.10** 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 listing 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 the methods for adding middleware 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 order in which they're added to the pipeline. This creates a pipeline similar to that shown in figure 3.11.

The exception handler middleware is called first, and it 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 3.10. Figure 3.12 shows what happens if you re-introduce a piece of

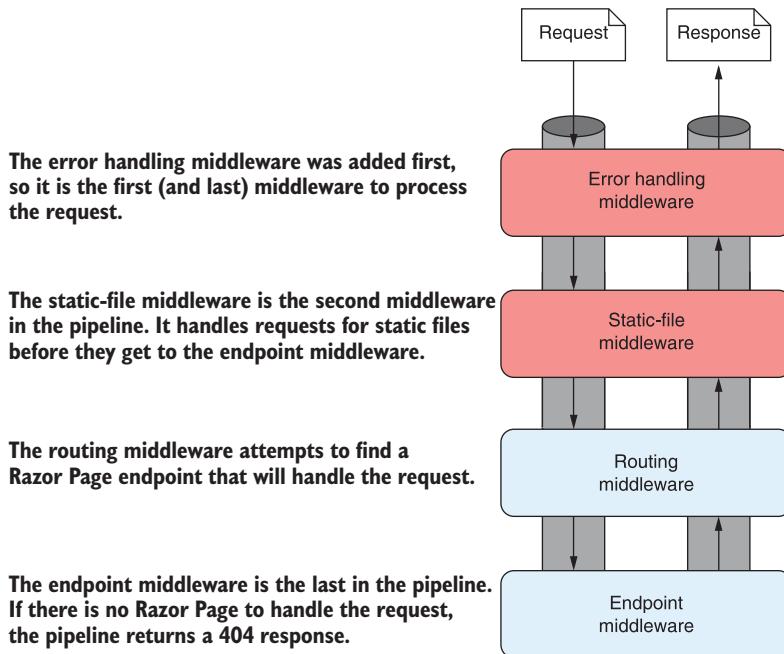


Figure 3.11 The middleware pipeline for the example application in listing 3.3. The order in which you add the middleware to `IApplicationBuilder` defines the order of the middleware in the pipeline.

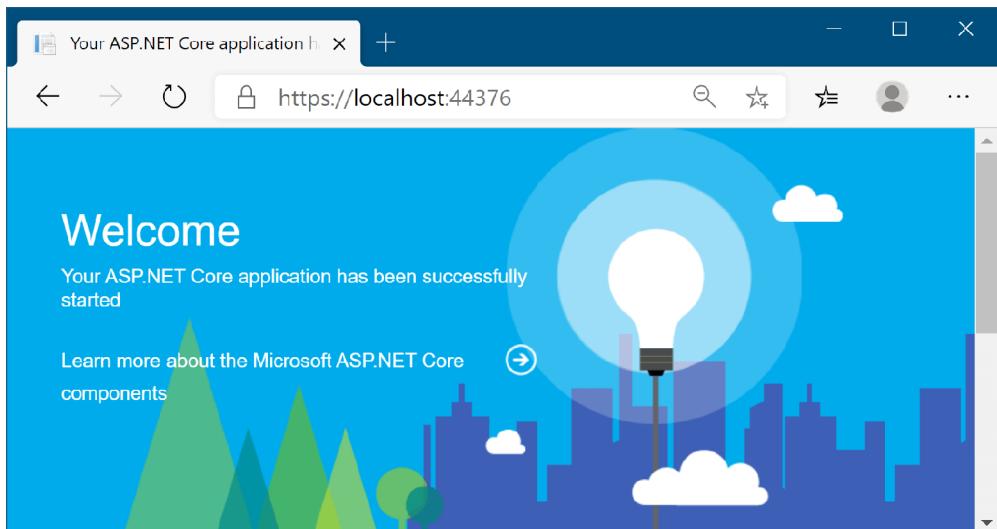


Figure 3.12 The Welcome page middleware response. The Welcome page middleware comes before the endpoint middleware, so a request to the home page returns the Welcome page middleware instead of the Razor Pages response.

middleware you saw previously, `WelcomePageMiddleware`, and configure it to respond to the `"/"` path as well.

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("/");
        app.UseExceptionHandler("/Error");
        app.UseStaticFiles();
        app.UseRouting();
        app.UseEndpoints(endpoints =>
        {
            endpoints.MapRazorPages();
        });
    }
}
```

← **WelcomePageMiddleware handles all requests to the `"/"` path and returns a sample HTML response.**

← **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 3.13. None of the other middleware in the pipeline runs for the request, so none has an opportunity to generate a response.

If you moved `WelcomePageMiddleware` to the end of the pipeline, after the call to `UseEndpoints`, 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 it 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 bidirectional. 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

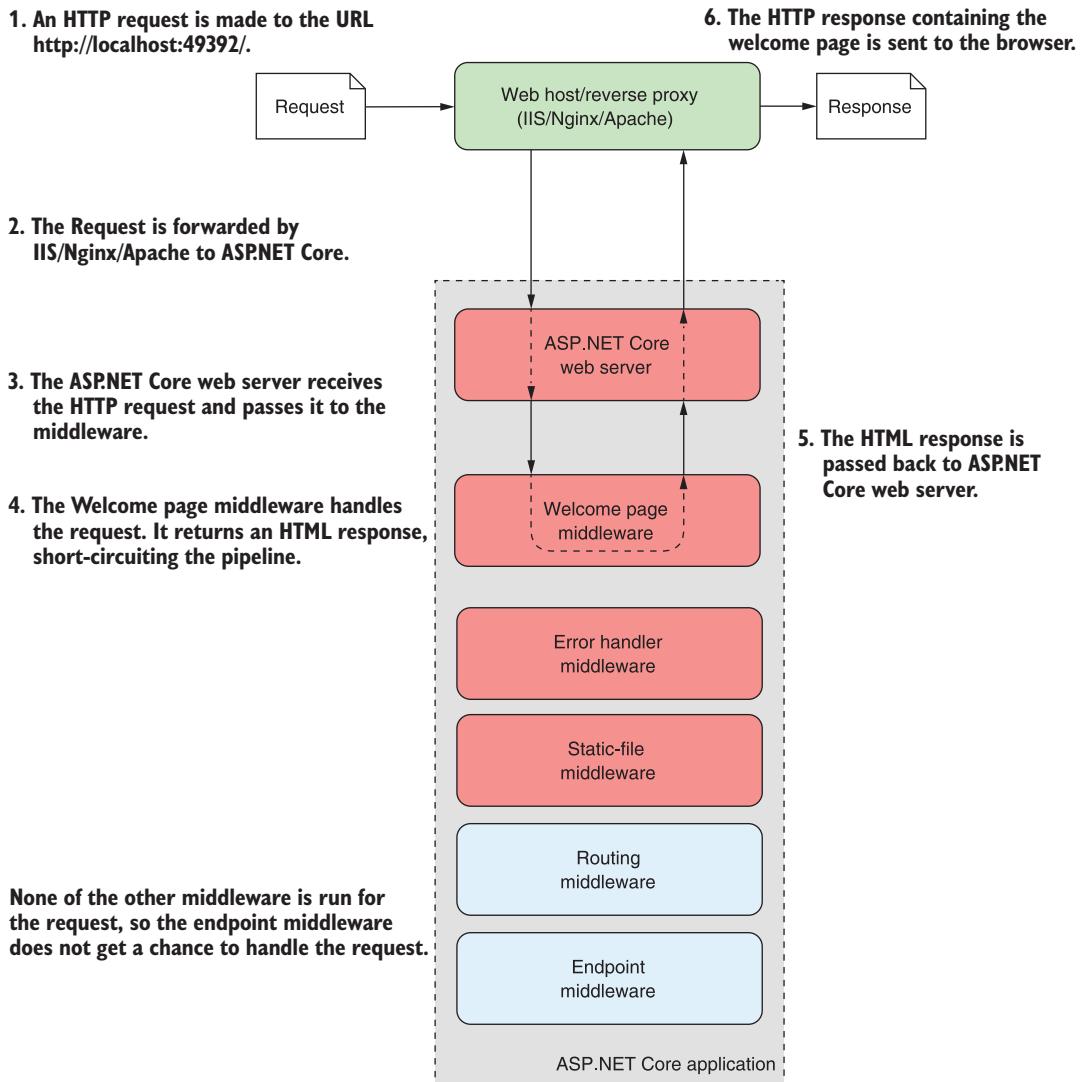


Figure 3.13 Overview of the application handling a request to the “/” path. The welcome page middleware is first in the middleware pipeline, so it receives the request before any other middleware. It generates an HTML response, short-circuiting the pipeline. No other middleware runs for the request.

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 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.<sup>3</sup> If an exception occurs in a middleware component, it propagates up the pipeline, as shown in figure 3.14. If the pipeline doesn't handle the exception, the web server will return a 500 status code back to the user.

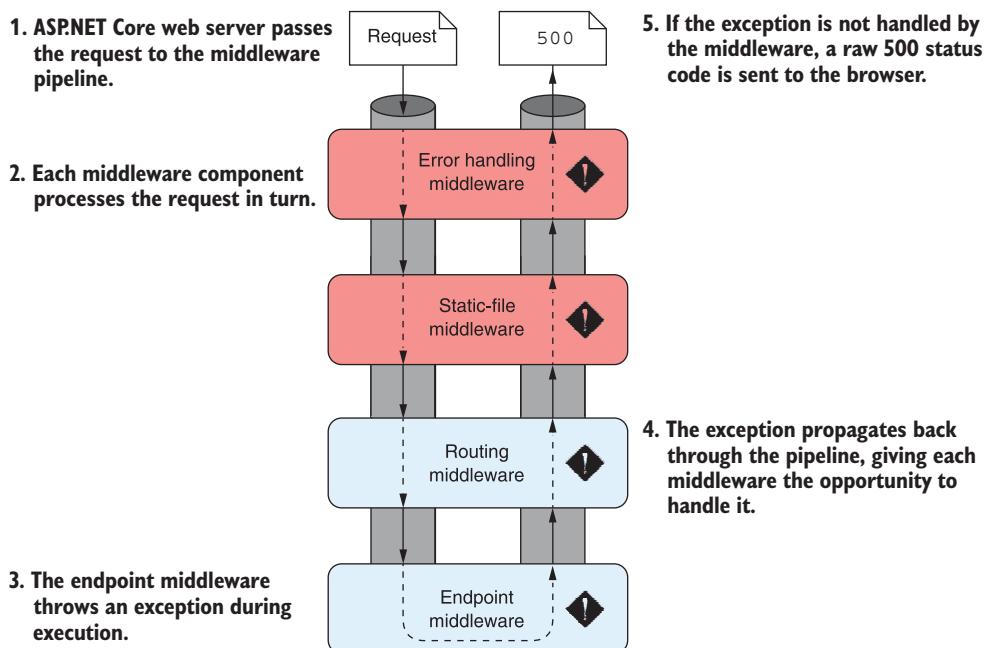


Figure 3.14 An exception in the endpoint middleware propagates through the pipeline. If the exception isn't caught by middleware earlier in the pipeline, then a 500 "Server error" status code is sent to the user's browser.

<sup>3</sup> C# 8.0 introduced non-nullable reference types. These provide a way to handle null values more clearly, 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: <http://mng.bz/7V0g>.

In some situations, an error won't cause an exception. Instead, middleware might generate an error status code. One such case 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 3.8. 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 3.15. Any responses generated by middleware earlier in the pipeline than the error handling middleware can't be intercepted.

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 3.16. 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.

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.

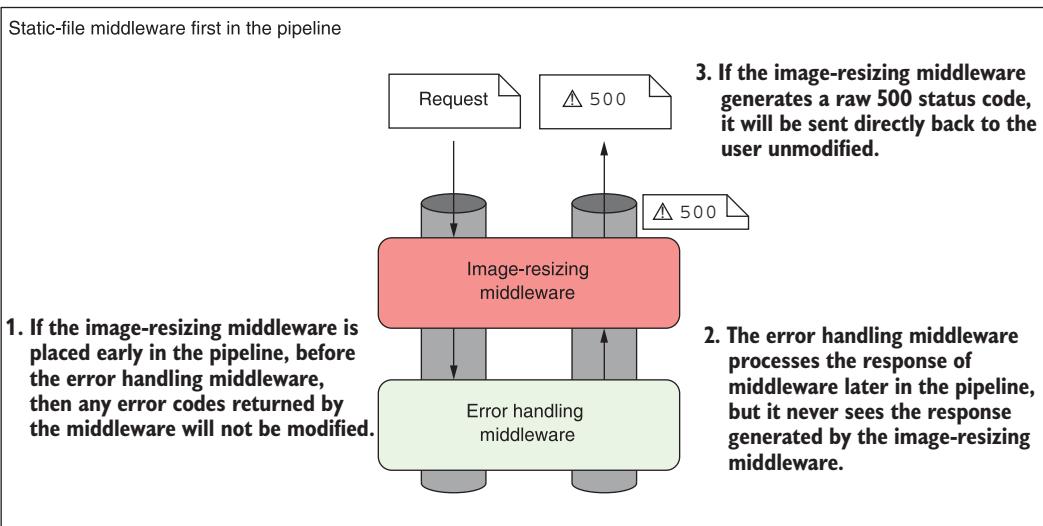
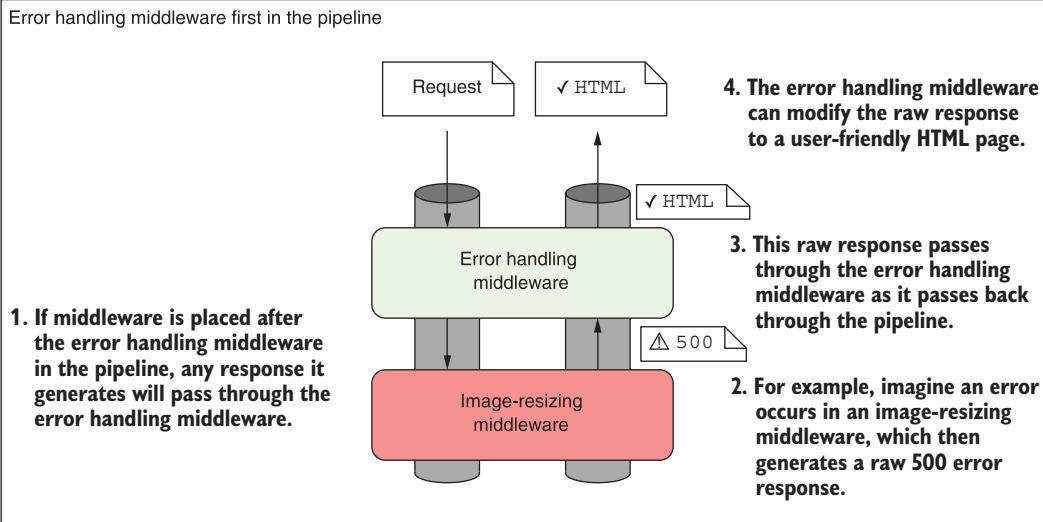
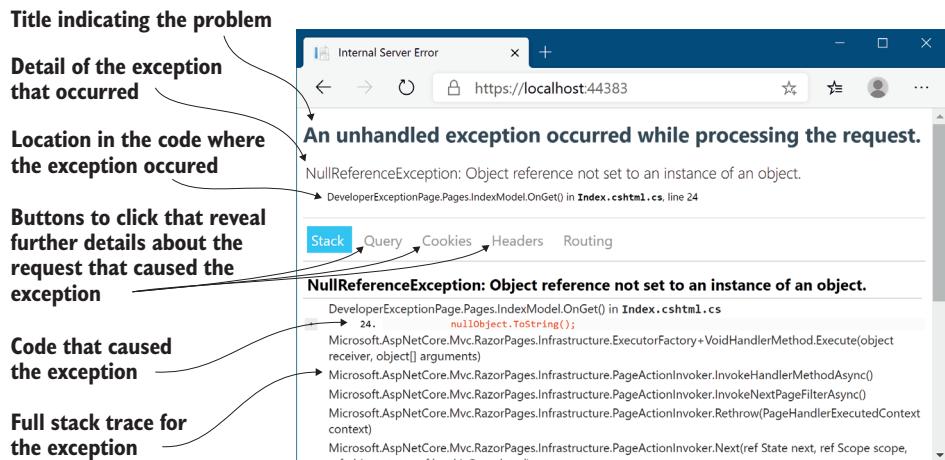


Figure 3.15 Error handling middleware should be placed early in the pipeline to catch raw status code errors. In the first case, the error handling middleware is placed before the image-resizing middleware, so it can replace raw status code errors with a user-friendly error page. In the second case, the error handling middleware is placed after the image-resizing middleware, so raw error status codes can't be modified.



**Figure 3.16** 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.

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. It's 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 3.17, can keep the look and feel of the application by using the same header, displaying the currently logged-in user, and displaying an appropriate message to the user instead of the full details of the exception.

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 in listing 3.5.

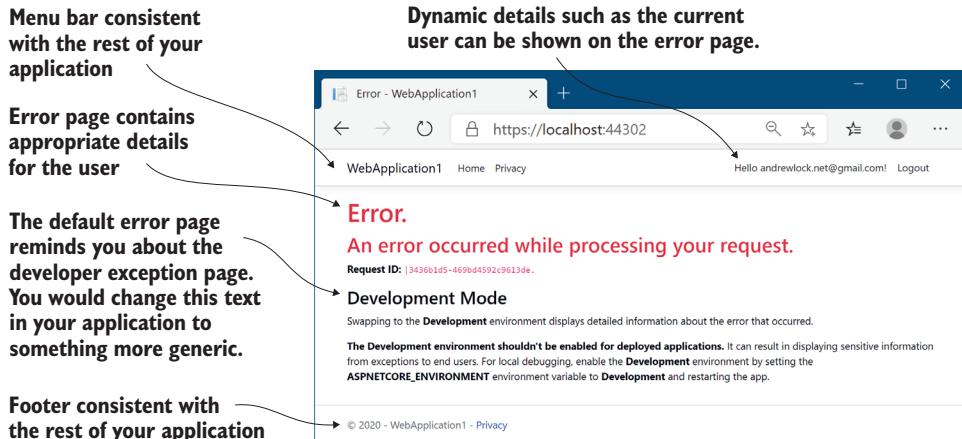


Figure 3.17 A custom error page created by `ExceptionHandlerMiddleware`. The custom error page can keep the same look and feel as the rest of the application by reusing elements such as the header and footer. More importantly, you can easily control the error details displayed to users.

### Listing 3.5 Configuring exception handling for development and production

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsDevelopment())
        ← Configure a different pipeline
        when running in development.

        app.UseDeveloperExceptionPage();
    else
        ← The developer exception page
        should only be used when
        running in development mode.

        app.UseExceptionHandler("/Error");
    }

    // additional middleware configuration
}
```

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 in listing 3.5, 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 3.18 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.

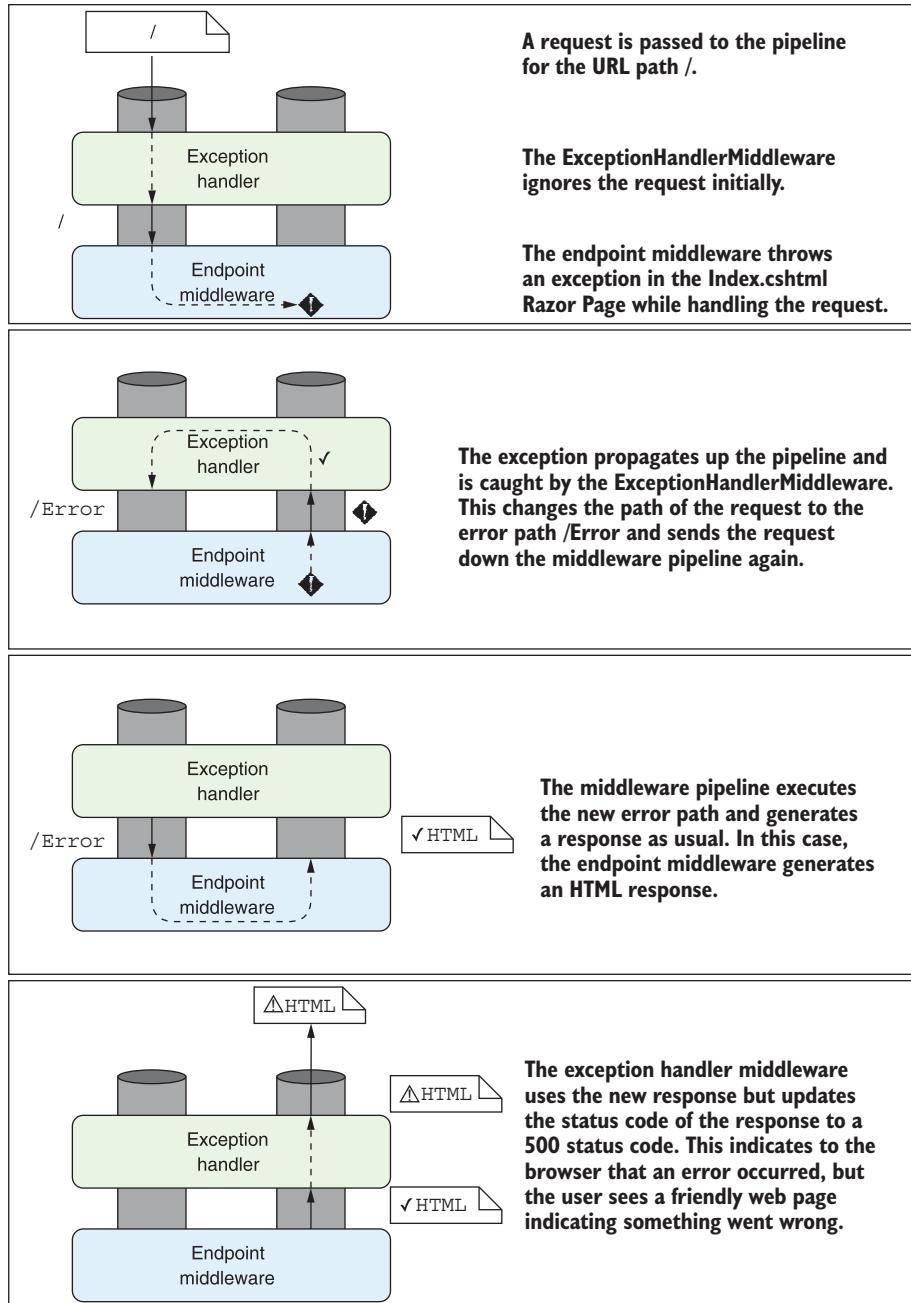
The sequence of events when an exception occurs somewhere in the middleware pipeline after `ExceptionHandlerMiddleware` is as follows:

- 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 3.17. 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



**Figure 3.18** `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.

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.

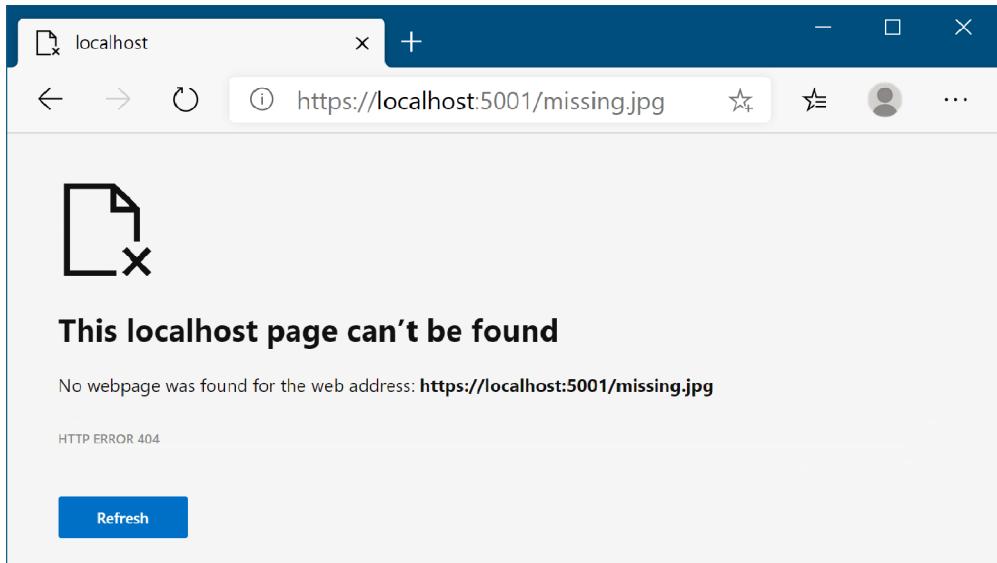
`ExceptionHandlerMiddleware` and `DeveloperExceptionPageMiddleware` are great for catching exceptions in your application, but exceptions aren't the only sort of errors you'll encounter. In some cases, your middleware pipeline will return an HTTP error status code in the response. It's important to handle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

### 3.3.3 **Handling other errors: `StatusCodesMiddleware`**

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.

If you don't handle these status codes, users will see a generic error page, such as in figure 3.19, 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.



**Figure 3.19** 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.

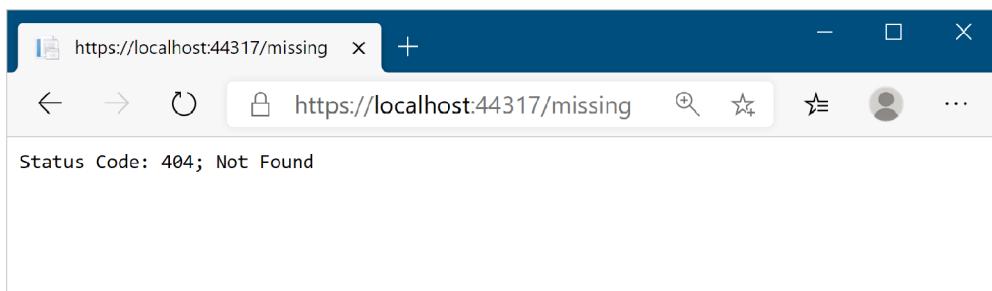
Microsoft provides `StatusCodesMiddleware` for handling this use case. As with all error handling middleware, you should add it early in your middleware pipeline, as it will only handle errors generated by later middleware components.

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 3.20.

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.



**Figure 3.20** Status code error page 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 `StatusCodePagesMiddleware` 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 `StatusCodePagesMiddleware` 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 3.21.

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.

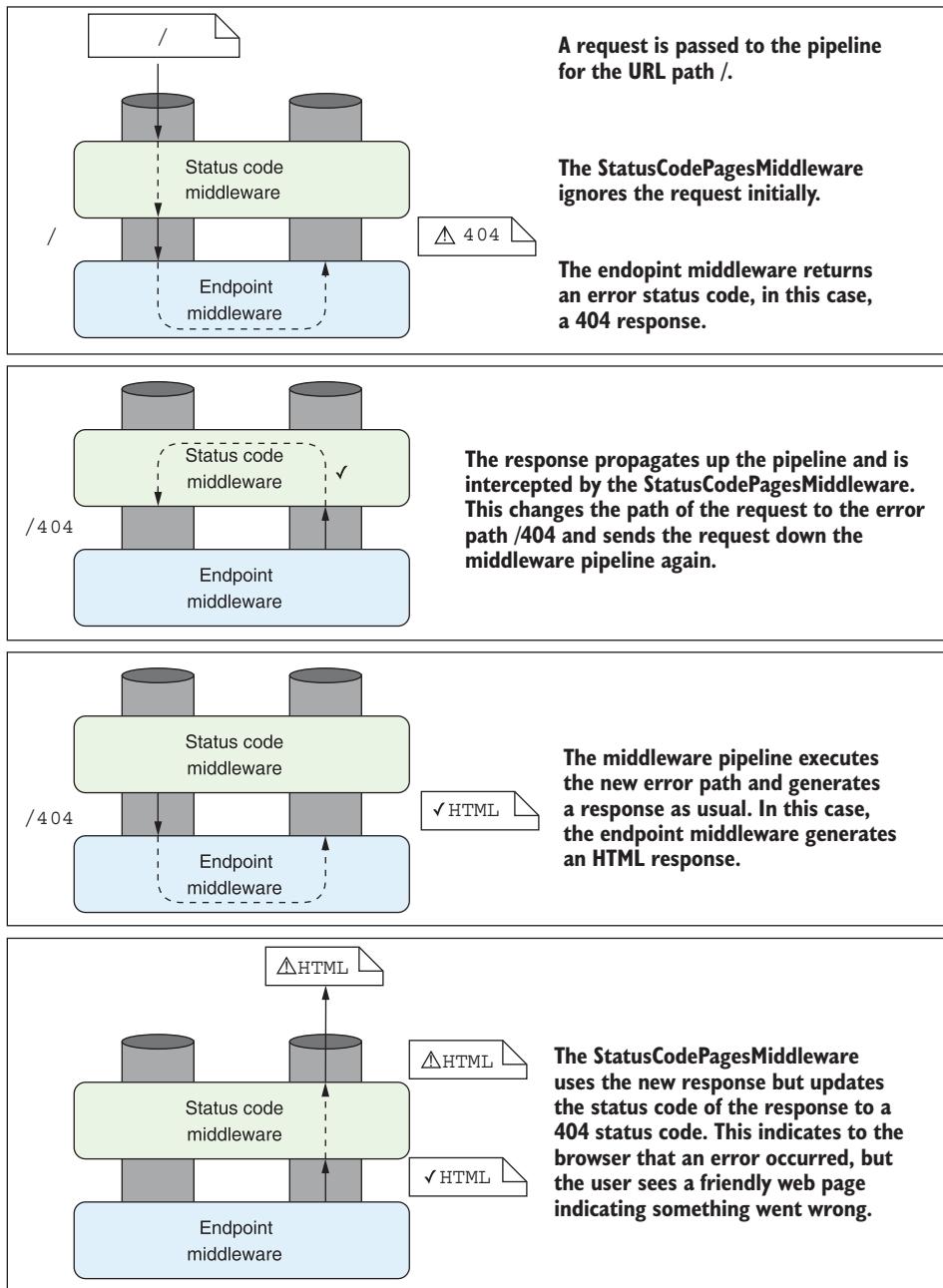
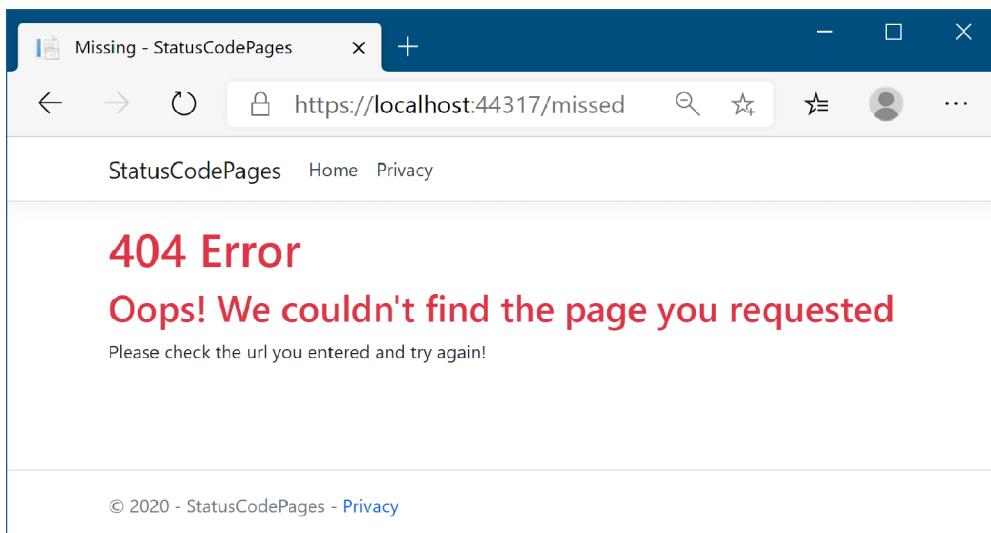


Figure 3.21 `StatusCodePagesMiddleware` re-executing the pipeline to generate an `HTML` body for a `404` response. A request to the `/` path returns a `404` response, which is handled by the `status code middleware`. The pipeline is re-executed using the `/404` path to generate the `HTML` response.

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 3.22. 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.



**Figure 3.22** 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.

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

You can use `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, such as `ExceptionHandlerMiddleware`, returns a message body along with an error code, it won't be modified.

**NOTE** `StatusCodePagesMiddleware` 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 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 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 at 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.

## 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 it's 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 it should never be used in production.
- `ExceptionHandlerMiddleware` lets you provide user-friendly custom error handling messages when an exception occurs in the pipeline. It is safe for use in production, as it does not expose sensitive details about your application.
- `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.

# *Creating a website 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, 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, how it builds on top of the ASP.NET Core MVC framework, and compare 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 in 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 makes 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” websites. 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* website 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 will 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.

To give you more of a flavor of how typical Razor Pages work, in this section we’ll 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 following listing shows only the `PageModel` 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;
    public CategoryModel(ToDoService service)
    {
        _service = service;
    }

    public ActionResult OnGet(string category)
    {
        Items = _service.GetItemsForCategory(category);
    }
}
```

The `ToDoService` is provided in the model constructor using dependency injection.

The `OnGet` handler takes a parameter, `category`.

The handler calls out to the `ToDoService` to retrieve data and sets the `Items` property.

```

    return Page();
}

public List<ToDoListModel> Items { get; set; } ← 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 binding 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 that 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 we'll 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, so it 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 that 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.

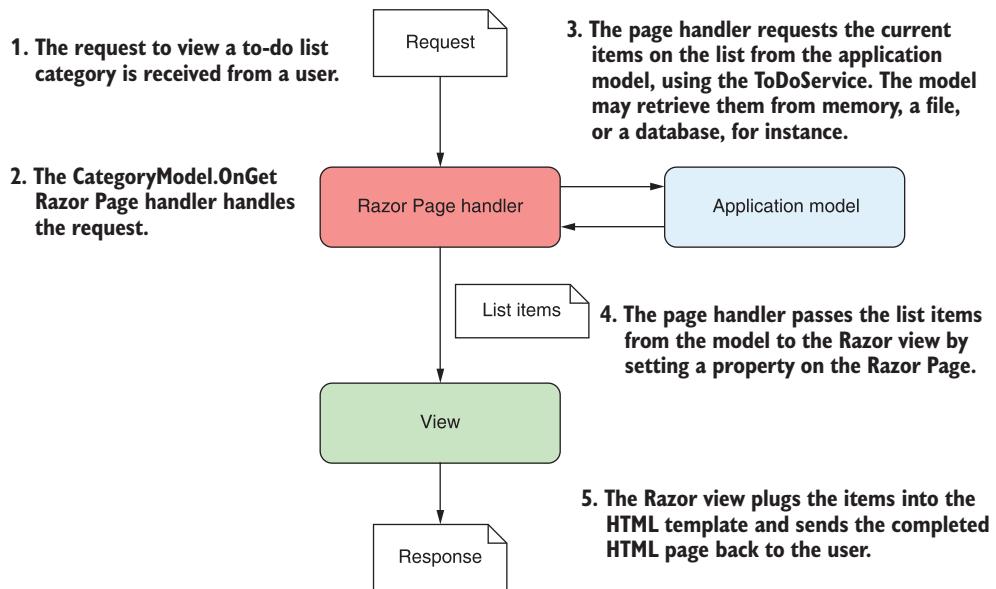


Figure 4.1 Requesting a to-do list page for a Razor Pages application. A different “component” handles each aspect of the request.

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

- *Model*—This is the data that needs to be displayed, the global state of the application. It's accessed via the ToDoService in listing 4.1.
- *View*—The template that displays the data provided by the model.
- *Controller*—This 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, which, 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. 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 that a controller receives a request to add a product to the user's shopping cart. The controller would receive the request and most likely would 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 that 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 applications. 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 it's typically used in web applications; Razor Pages use this pattern. 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'll 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 some client-side frameworks. It differs from MVC in that there is a bidirectional 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 data binding where a view model is "bound" to a view.

**(continued)**

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 it 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 `RoutingMiddleware` and `EndpointMiddleware`, as shown in figure 4.2. Once a request has been processed by earlier middleware (and assuming none of them has handled the request and short-circuited the pipeline), the routing middleware will select which Razor Page handler should be executed, and the endpoint middleware executes the page handler.

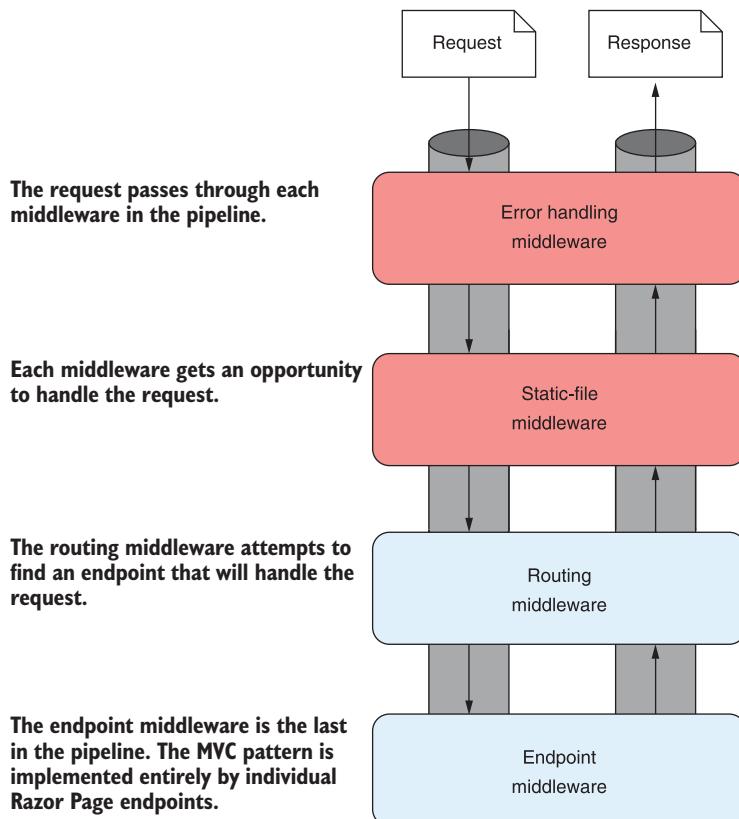


Figure 4.2 The middleware pipeline for a typical ASP.NET Core application. The request is processed by each middleware in sequence. If the request reaches the routing middleware, the middleware selects an endpoint, such as a Razor Page, to execute. The endpoint middleware executes the selected endpoint.

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` path.

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

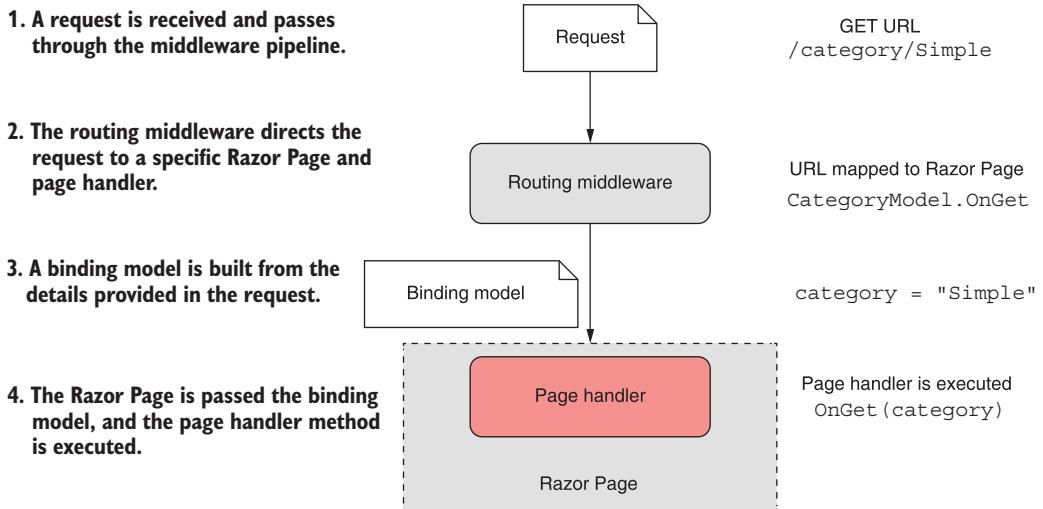
**TIP** I'm using the term *Razor Page* to refer to the combination of the Razor view and the `PageModel` that includes the page handler. Note that that `PageModel` class is *not* the “model” we're referring to when describing the MVC pattern. It fulfills other roles, as you will see later in this section.

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—data that's required by a page handler.

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.

The binding model in this case corresponds to the method parameter of the `OnGet` page handler. An instance of the Razor Page is created using its constructor, and the



**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`.

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

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 parts 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

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 to-do items, for example.
3. The services load the details of the to-do items from the database and return them to the action method.

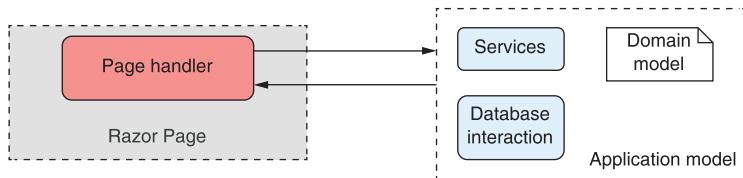


Figure 4.4 When executed, an action will invoke the appropriate methods in the application model.

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* in the MVC pattern 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, such as 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.

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.

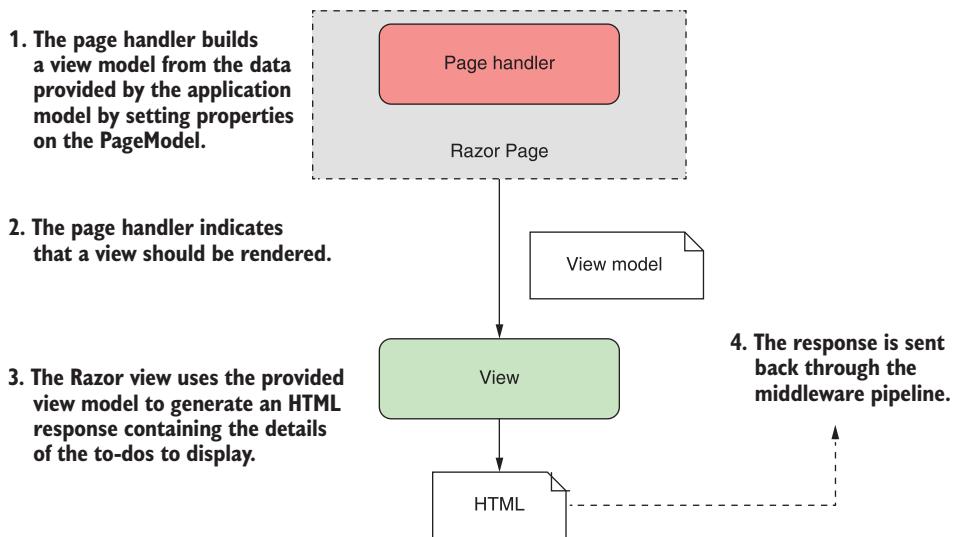


Figure 4.5 The page handler builds a view model by setting properties on the `PageModel`. It's the view that generates the response.

### 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.

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 which response is required, 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.

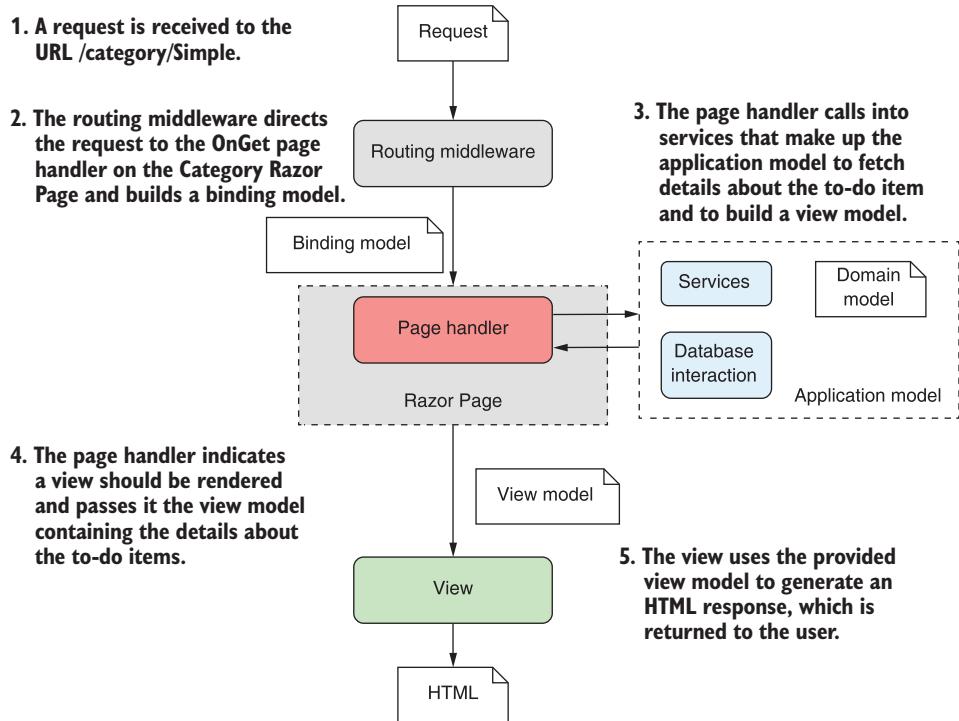


Figure 4.6 A complete Razor Pages request for the list of to-dos in the “Simple” category

### 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 that 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, which 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 you how to start with a basic empty application and add Razor Pages to it from scratch.

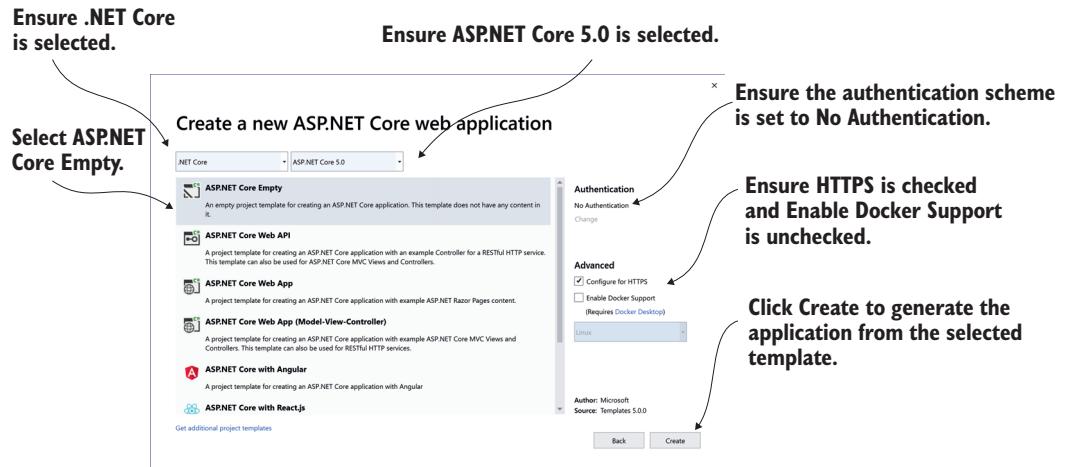
The result of your efforts won't be exciting yet. We'll display "Hello World" on a web page, but this will show how simple it is to convert an ASP.NET Core application to use Razor Pages. It will also emphasize the pluggable nature of ASP.NET Core—if you don't need the functionality provided by Razor Pages, 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 solution name, and click Create.
- 4 On the following screen, create a basic template without MVC or Razor Pages by selecting the ASP.NET Core Empty project template in Visual Studio, as shown in figure 4.7. You can create a similar empty project using the .NET CLI with the `dotnet new web` command.
- 5 Add the necessary Razor Page services (shown 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



**Figure 4.7** 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.

(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 4.8.
- 9 On the following page, select Razor Page – Empty and click Add. In the following dialog box, name your page `Index.cshtml`, as shown in figure 4.9.

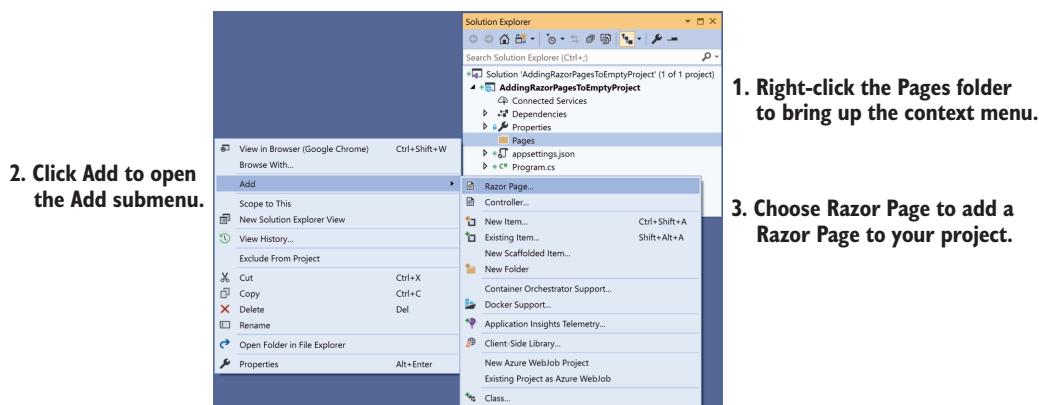


Figure 4.8 Adding a new Razor Page to your project

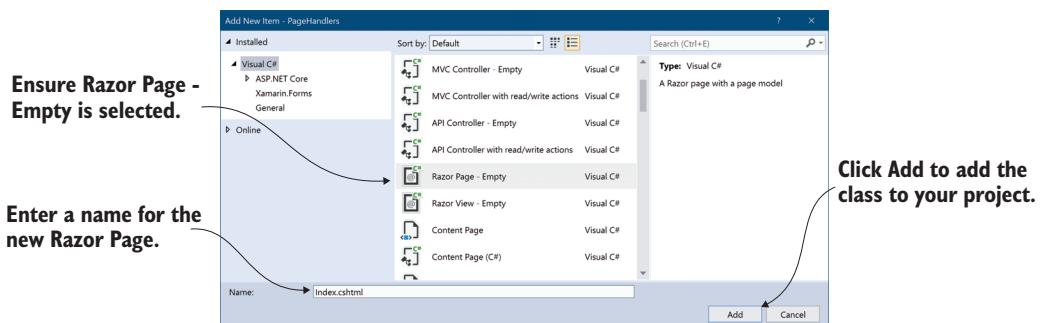


Figure 4.9 Creating a new Razor Page using the Add Razor Page dialog box

- 10** After Visual Studio has finished generating the file, open the Index.cshtml file, and update the HTML to say Hello World! by replacing the file contents with the following:

```

@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 home page.

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 filename. 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 specific Razor Page handlers 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 that'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'll 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, 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 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 at 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 the concept of *controllers* and *action methods*. These are almost directly analogous to their Razor Pages counterparts, as you can see in figure 4.10, 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.

Listing 4.2 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.

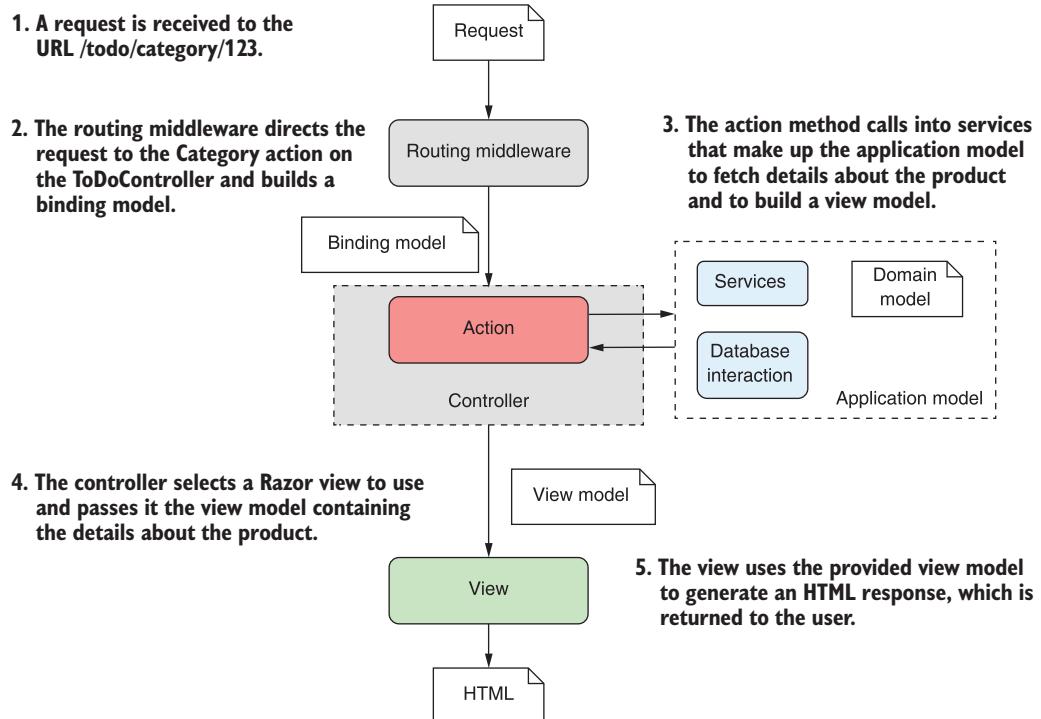


Figure 4.10 A complete MVC controller request for a category. The MVC controller pattern is almost identical to that of Razor Pages, shown in figure 4.6. The controller is equivalent to a Razor Page, and the action is equivalent to a page handler.

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

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

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

        return View(viewModel);
    }
}
```

The ToDoService is provided in the controller constructor using dependency injection.

The Category action method takes a parameter, id.

The action method calls out to the ToDoService to retrieve data and build a view model.

The view model is a simple C# class, defined elsewhere in your application.

Returns a ViewResult indicating the Razor view should be rendered, passing in the view model

```
public ActionResult Create(TodoListModel model)
{
    // ...
}
```

MVC controllers often contain multiple action methods that 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 interdependencies, and that 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 it 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 it’s 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.<sup>1</sup>

**NOTE** You don't *have* to make your controllers very large like this. It's just a common pattern. You could, for example, create a separate controller for every action instead.

Another pitfall of the MVC controllers is 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 Pages 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.

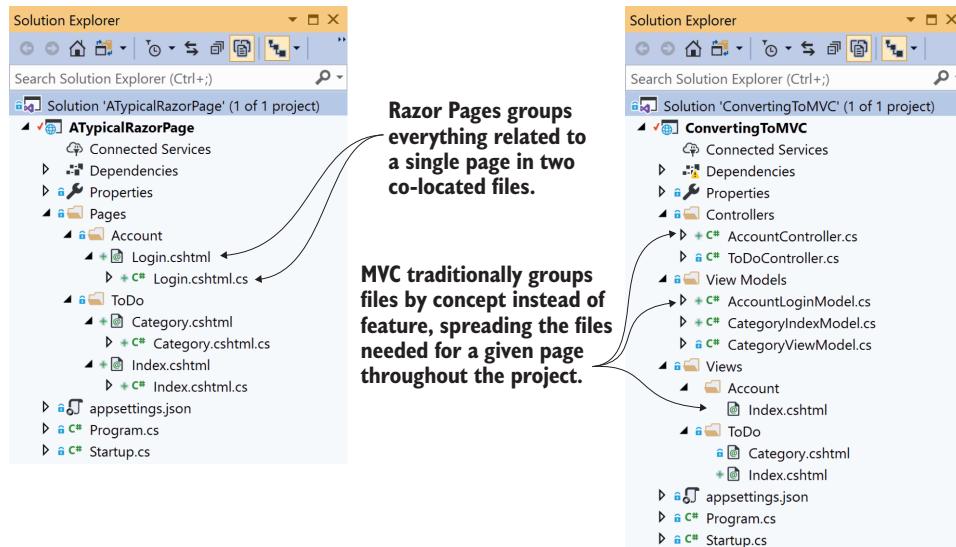


Figure 4.11 Comparing the folder structure for an MVC project to the folder structure for a Razor Pages project

<sup>1</sup> 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!

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. MVC controllers contain action methods that 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 of using Razor Pages 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 you're 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. These are a few such scenarios:

- *When you don't want to render views*—Razor Pages are best for page-based applications, where you're rendering a view for the user. If you're building a Web API, 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, it's probably not worth converting your existing MVC controllers to Razor Pages. It makes more sense to keep your

existing code, and perhaps to 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, you might consider using custom middleware directly (see chapter 19) or an alternative protocol like gRPC (<https://docs.microsoft.com/aspnet/core/grpc>). You might want to also take a look at *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 WebSockets. SignalR also provides simple transport fallbacks and a remote procedure call (RPC) app model. For details, see the documentation at <https://docs.microsoft.com/aspnet/core/signalr>.

Another option available in ASP.NET Core 5.0 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/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 that 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 three 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 that 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 a page handler 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, such as a user logging in with their username and password, those values must be contained in the request. In other cases, there will be no values, such as when a user requests the home page 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 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 following listing. If any of the values provided in the request are *not* bound to a property or page handler argument, the additional values will go unused.

**Listing 4.3 Example Razor Page handlers**

```
public class SearchModel : PageModel
{
    private readonly SearchService _searchService;
    public SearchModel(SearchService searchService)
    {
        _searchService = searchService;
    }

    [BindProperty]
    public BindingModel Input { get; set; }
    public List<Product> Results { get; set; }
```

Undecorated properties  
will not be  
model-bound.

The `SearchService` is  
provided to the `SearchModel`  
for use in page handlers.

Properties decorated  
with the `[BindProperty]`  
attribute will be model-  
bound.

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

The page handler doesn't need to check if the  
model is valid. Returning void will render the view.

```
    public IActionResult OnPost(int max)
    { }
```

The max parameter in this page  
handler will be model-bound  
using the values in the request.

```

If the request  
was not valid, the  
method indicates  
the user should  
be redirected to  
the Index page. | if (ModelState.IsValid)
{ |
    Results = _searchService.Search (Input.SearchTerm, max);
    return Page();
}
return RedirectToPage("./Index");
}
}

```

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.

**NOTE** Unlike most .NET classes, you can't use method overloading to have multiple page handlers on a Razor Page with the same name.

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, 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'll 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 decide what *type* of response to return, but they don't generate the response themselves. It's the `IActionResult` returned by a page handler that, 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 that 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
- `StatusCodesResult`—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 `StatusCodesResult`. 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'll 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, usually you’ll be using `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.

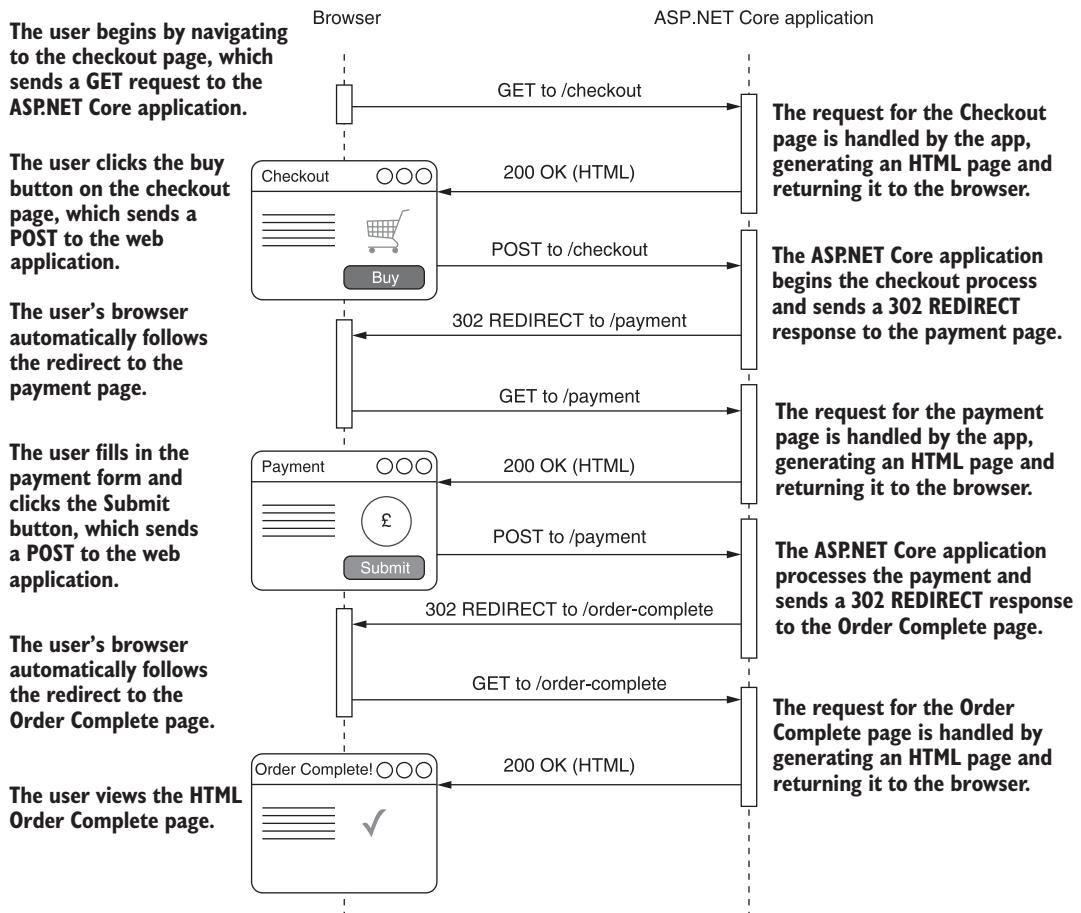


Figure 4.12 A typical POST, REDIRECT, GET flow through a website. A user sends their shopping basket to a checkout page, which validates its contents and redirects to a payment page without the user having to manually change the URL.

In this flow, whenever you return HTML you use a `PageResult`; when you redirect to a new page, you use a `RedirectToPageResult`.

**TIP** Razor Pages are generally designed to be stateless, so if you want to persist data between multiple pages, you need to place it in a database or similar store. If you just want to store data for a single request, you may be able to use `TempData`, which stores small amounts of data in cookies for a single request. See the documentation for details: <http://mng.bz/XdXp>.

### 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 `StatusCodeResult` and setting the status code returned explicitly to 404.

Note that `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 `StatusResult` 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, providing 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. You don't want 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 it's 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.

## 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 they 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`.

# *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 that 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 to 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 you 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?

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.

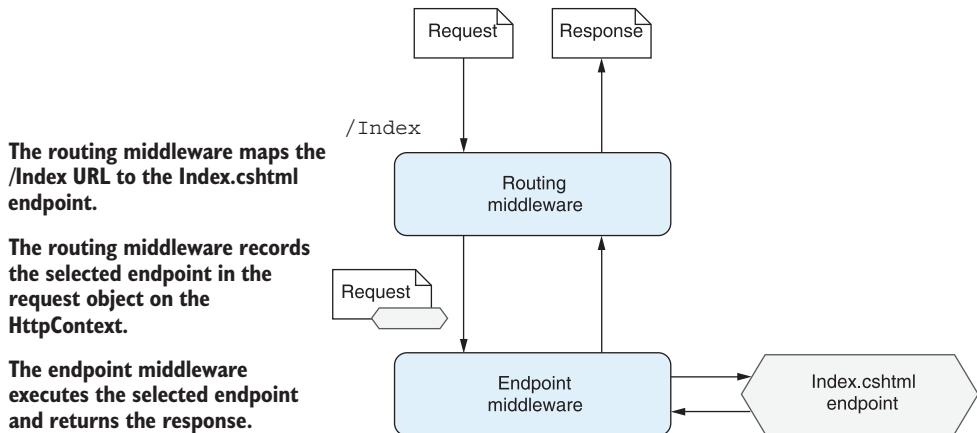
In chapter 3 you saw that an ASP.NET Core application contains a middleware pipeline, which defines the behavior of your application. Middleware is well suited to handling both cross-cutting concerns, such as logging and error handling, and narrowly focused requests, such as requests for images and CSS files.

To handle more complex application logic, you'll typically use the `Endpoint-Middleware` 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 in a controller.

At this point you've already seen several simple applications built with Razor Pages in previous chapters, so you've 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 the `Index.cshtml` Razor Page should be executed, as shown in figure 5.1.



**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.

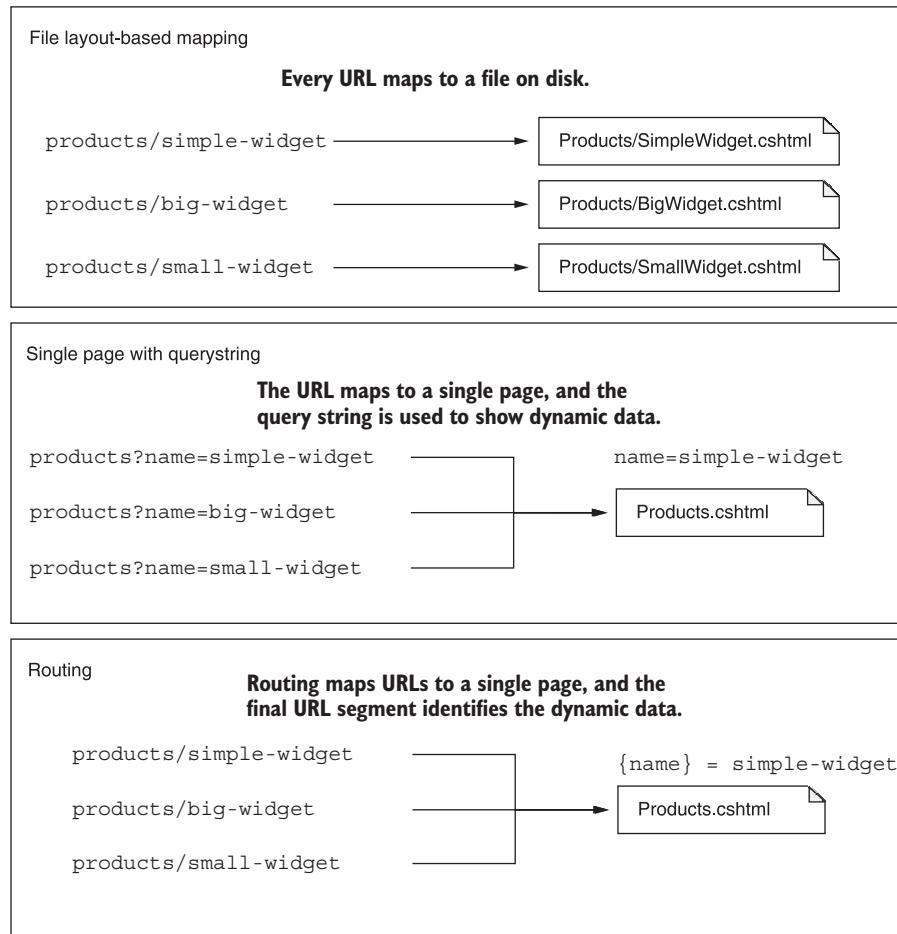
For example, consider an e-commerce application that sells multiple products. Each product needs to have its own URL, so if you were using a purely file layout-based routing system, you'd only have two options:

- *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 you 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 it 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 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, you can completely customize the URLs, as shown in figure 5.2. This gives you much more flexibility and can be important in real-life applications for search engine optimization (SEO) reasons.<sup>1</sup>



**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.

<sup>1</sup> 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>.

As well as enabling dynamic URLs, routing fundamentally decouples the URLs in your application from the filenames 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 filenames or locations. Depending on your routing configuration, you could set the URL pointing to the `View.cshtml` 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**

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. ASP.NET Core 5.0 uses the same endpoint routing system as ASP.NET Core 3.0.

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 sense 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();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
        endpoints.MapHealthChecks("/healthz");
    });
}
```

**Add the EndpointRoutingMiddleware to the middleware pipeline.**

**Add the EndpointMiddleware to the pipeline and provide a configuration lambda.**

**Register all the Razor Pages in your application as endpoints.**

**Register a health-check endpoint at the route /healthz.**

```

endpoints.MapGet("/test", async context =>
{
    await context.Response.WriteAsync("Hello World!");
}) ;
}
}

```

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, 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.

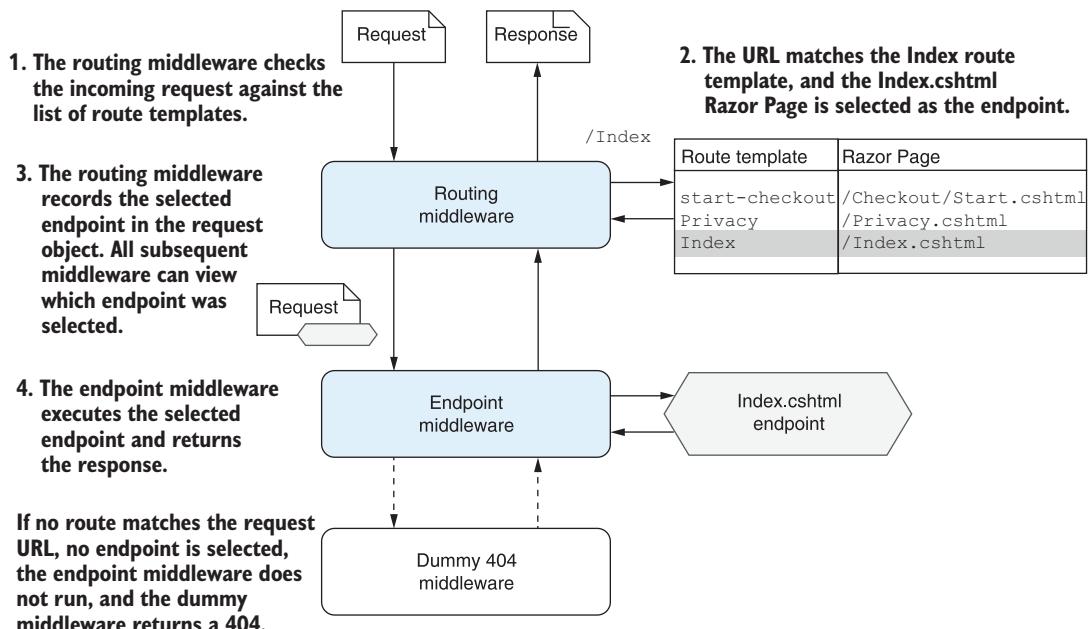


Figure 5.3 Endpoint routing uses a two-step process. The `RoutingMiddleware` selects which endpoint to execute, and the `EndpointMiddleware` executes it. If the request URL doesn't match a route template, the endpoint middleware will not generate a response.

If the request URL *doesn't* match a route template, 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`.

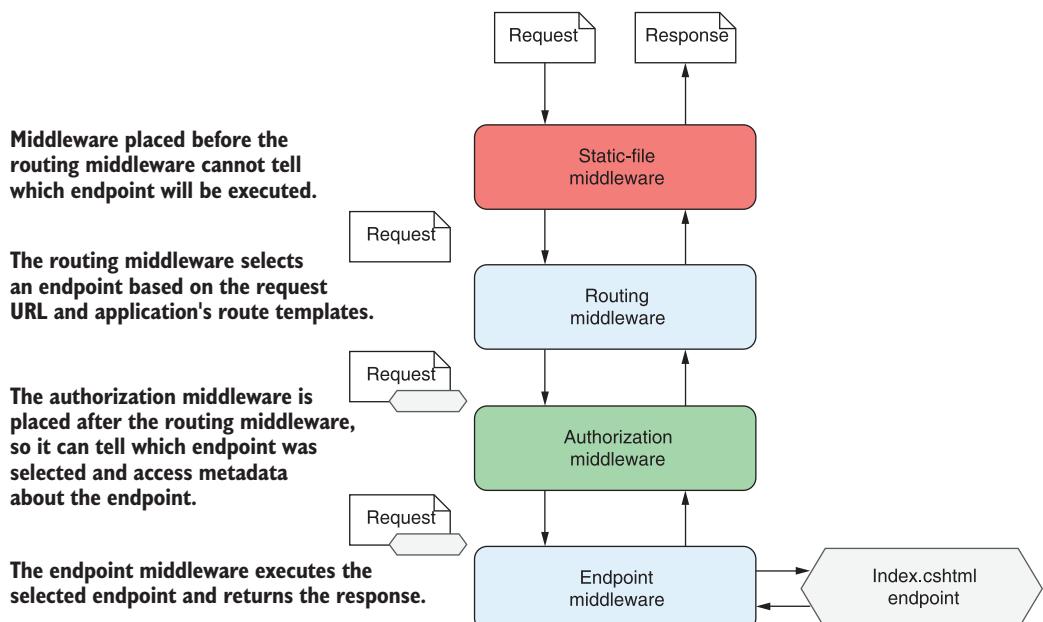


Figure 5.4 Middleware placed before the routing middleware doesn't know which endpoint the routing middleware will select. Middleware placed between the routing middleware and the endpoint middleware can see the selected endpoint.

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'll 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 toward 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 that 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 controllers	Routes are defined in a different place from your controllers. Overriding the route conventions 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, attribute routing is the best option and is the recommended approach.<sup>2</sup>

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, I would still stick with attribute routing in that scenario.

---

<sup>2</sup> .NET 5.0 also includes support for building simple APIs without the overhead (or convenience) of either MVC or Razor Pages. Filip W. has an excellent post on creating small, simple API apps using C# 9 and .NET 5.0: <http://mng.bz/yYey>.

**NOTE** For the reasons above, this book focuses on attribute routing. Virtually all the features described in this section also apply to convention-based routing. For details on convention-based routing, see Microsoft’s “Routing to controller actions in ASP.NET Core” documentation: <http://mng.bz/ZP0O>.

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, such as ensuring that it’s numeric

Most applications will use a variety of these features, but you’ll 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’ll 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 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.cshtml`, the framework creates a route template with the value "`Products/View`", as shown in figure 5.5.

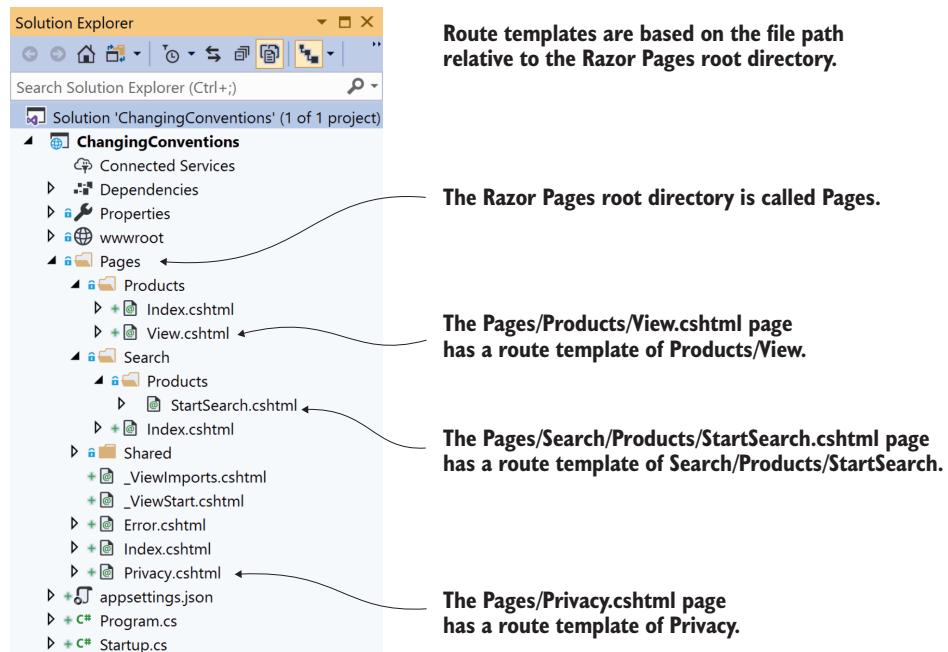


Figure 5.5 By default, route templates are generated for Razor Pages based on the path of the file relative to the root directory, `Pages`.

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" and 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 e-commerce site where you want each product to have its own URL, but which map 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.

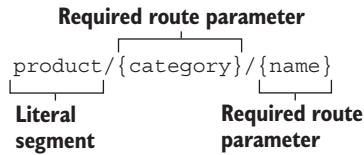


Figure 5.6 A simple route template showing a literal segment and two required route parameters

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"`.

**TIP** Literal segments in ASP.NET Core are not case-sensitive.

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 e-commerce application, such as

- `/bags/rucksack-a`—Where `category=bags` and `name=rucksack-a`
- `/shoes/black-size9`—Where `category=shoes` and `name=black-size9`

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 they are 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 preceding directive, 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 e-commerce 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 preceding 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 e-commerce 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 this 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 you 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`.

**TIP** You can also think of custom route templates that start with `"/"` as absolute route templates, while other route templates are relative to their location in the file hierarchy.

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'll look in more depth at the route template syntax and some of the other features available.

## 5.4 **Exploring the route template syntax**

In addition to 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 place additional constraints on the values that are valid for a given route parameter. This section takes a look at these features and the 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.

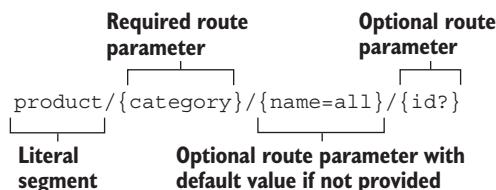


Figure 5.7 A more complex route template showing literal segments, named route parameters, optional parameters, and default values

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, 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=computers, 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 it 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, given the route template in figure 5.7, the following two URLs 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, so the router happily assigns route values and matches the template when you might not want it to! These are the route values assigned:

- /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, 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 in Microsoft's "Routing in ASP.NET Core" documentation at <http://mng.bz/xmae>.

**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

**Table 5.3** A few route constraints and their behavior when applied (*continued*)

Constraint	Example	Match examples	Description
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 by separating the constraints with colons.

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, 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, such as 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, 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

**(continued)**

find yourself needing to manually control the order, this is a very strong indicator that your URLs are confusing.<sup>a</sup>

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!

---

<sup>a</sup> If you're really sure you do need to control route template ordering, see the documentation at <http://mng.bz/MXqo>. Note that you can only control the order for additional routes added using conventions, as you'll see in section 5.7.

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 rates to GBP and EUR
- /USD/convert/GBP/EUR/CAD—Show USD with exchange rates for GBP, EUR, and CAD

If you want to support showing *any* number of currencies as the URLs above do, you need a way of capturing *everything* after the convert segment. You could achieve this for the Pages/Convert.cshtml Razor Page by using a catch-all parameter in the @page directive, as shown in figure 5.8.

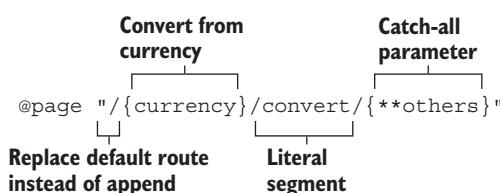


Figure 5.8 You can use catch-all parameters to match the remainder of a URL. Catch-all parameters may include the "/" character or may be an empty string.

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.<sup>3</sup>

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.

## 5.5 Generating URLs from route parameters

In this section, we'll 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.

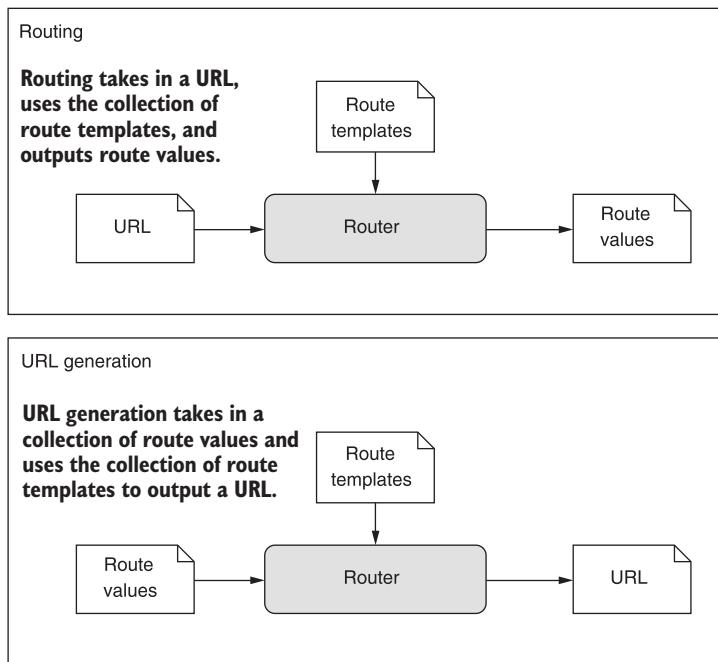
One of the by-products of using the routing infrastructure in ASP.NET Core is that your URLs can be somewhat fluid. If you rename a Razor Page, the URL associated with that page will also change. For example, renaming the `Pages/Cart.cshtml` page to `Pages/Basket/View.cshtml` would cause the URL you use 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, 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.

---

<sup>3</sup> For details and examples of this behavior, see the documentation: <http://mng.bz/aoGo>.



**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
{
    public void OnGet()
    {
        var url = Url.Page("Currency/View", new { code = "USD" });
    }
}
  
```

Deriving from `PageModel` gives access to the `Url` property.

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, the route value will be used in the URL path, giving `/Currency/View/GBP`.

If a route doesn't contain the route value explicitly, as in the "`Currency/View`" template, 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 it's the usual approach taken in most cases. If you're using MVC controllers for your APIs, the process is much the same as for Razor Pages, though the methods are slightly different.

## 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
{
    [HttpGet("currency/index")]
    public IActionResult Index()
    {
        var url = Url.Action("View", "Currency",
            new { code = "USD" });
        return Content($"The URL is {url}");
    }

    [HttpGet("currency/view/{code}")]
    public IActionResult View(string code)
    {
```

**Deriving from Controller gives access to the `Url` property.**

**You provide the action and controller name, along with any additional route values.**

**This will return "The URL is /Currency/View/USD".**

**The URL generated will route to the `View` action method.**

```

        /* method implementation */
    }
}

```

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 Microsoft’s “Routing in ASP.NET Core” documentation for further details: <http://mng.bz/xmae>.

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`

You’ve seen how to generate a string containing a URL 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 following listing shows how you can generate a URL that automatically redirects a user to a different Razor Page 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 });
    }
}
```

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 of 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, 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, 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 to change completely. For example, you may want to create a mobile application that reuses the business logic from 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 frontend application design out of your business logic. This pattern is known generally as the Dependency Inversion principle.<sup>4</sup>

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.

<sup>4</sup> 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>.

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` 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 following listing. 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 if you have problems.<sup>5</sup>

### Listing 5.5 Generating URLs using the `LinkGeneratorClass`

```
public class CurrencyModel : PageModel
{
    private readonly LinkGenerator _link;
    public CurrencyModel(LinkGenerator linkGenerator)
    {
        _link = linkGenerator;
    }

    public void OnGet()
    {
        var url1 = Url.Page("Currency/View", new { id = 5 });
        var url3 = _link.GetPathByPage(
            HttpContext,
            "/Currency/View",
            values: new { id = 5 });
        var url2 = _link.GetPathByPage(
            "/Currency/View",
            values: new { id = 5 });
        var url4 = _link.GetUriByPage(
            page: "/Currency/View",
            handler: null,
            values: new { id = 5 },
            scheme: "https",
            host: new HostString("example.com"));
    }
}
```

Url can generate relative paths using `Url.Page`.  
You can use relative or absolute Page paths.

`LinkGenerator` can be accessed using dependency injection.

`GetPathByPage` is equivalent to `Url.Page` when you pass in `HttpContext`. You can't use relative paths.

Other overloads don't require an `HttpContext`.

`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—you have a typo in your path or you forgot to include a required route parameter, for example—the URL generated will be null. It's worth checking the generated URL for null explicitly, just to be sure there's no problems.

<sup>5</sup> You can find the `LinkGenerator` documentation here: <http://mng.bz/goWx>.

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'll look at page handlers and how you 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 how to choose which handler to execute.

Consider the Razor Page `SearchModel` shown in the following listing. This Razor Page has three handlers: `OnGet`, `OnPostAsync`, and `OnPostCustomSearch`. The bodies 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()           ← Handles GET requests
    {
        // Handler implementation
    }

    public Task OnPostAsync()     ← Handles POST requests. The
    {
        // Handler implementation
    }                                async suffix is optional and is
                                    ignored for routing purposes.

    public void OnPostCustomSearch() ← Handles POST requests where
    {
        // Handler implementation
    }                                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

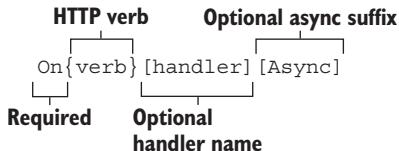
The handler route value typically comes from a query string value in the request URL, for example, `/Search?handler=CustomSearch`. If you don't like the look of query strings (I don't!) you can include the `{handler}` route parameter in your Razor Page's

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 is 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`.<sup>6</sup>



**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 it 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() { }
```

---

<sup>6</sup> The `async` suffix naming convention is suggested by Microsoft, though it is unpopular with some developers. NServiceBus provides a reasoned argument against it here (along with Microsoft's advice): <http://mng.bz/e59P>.

**DEFINITION** If a page handler does not match a request's HTTP verb and handler value, an *implicit* page handler is executed that renders the associated Razor view. Implicit page handlers take part in model binding and use page filters 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).<sup>7</sup>

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, the next section is for you. In the next section we'll 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 filenames 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/Product-Details`.

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 you can 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();
```

Add the standard  
Razor Pages services.

<sup>7</sup> 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.

```

Configure the
RouteOptions
object by
providing a
configuration
method.    ↗ services.Configure<RouteOptions>(options =>
{
    options.AppendTrailingSlash = true;
    options.LowercaseUrls = true;
    options.LowercaseQueryStrings = true;
});

```

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 following listing 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
    : IOutboundParameterTransformer
{
    public string TransformOutbound(object value)
    {
        if (value == null) return null;    ← Guard against null values to
                                         avoid runtime exceptions.

        return Regex.Replace(value.ToString(),
            "([a-z])([A-Z])", "$1-$2").ToLower();    ← 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 following listing 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`**

```

Registers the
parameter transformer
as a convention used
by all Razor Pages    ↗ public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages()
        .AddRazorPagesOptions(opts =>
    {
        opts.Conventions.Add(
            new PageRouteTransformerConvention(
                new KebabCaseParameterTransformer()));
        opts.Conventions.AddPageRoute(
            "/Search/Products/StartSearch", "/search-products");
    });
}

```

AddRazorPagesOptions can be used to customize the conventions used by Razor Pages.

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, such as to add an extra header to every page's response, you can use a custom convention. Microsoft's "Razor Pages route and app conventions in ASP.NET Core" documentation contains details on everything that's available: <http://mng.bz/A0BK>.

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, such as 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 may 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 templates, as shown in figure 5.11.

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. We'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 (MVC) or page handler (Razor Pages) parameters, and perhaps more importantly, how to validate the values you're provided.

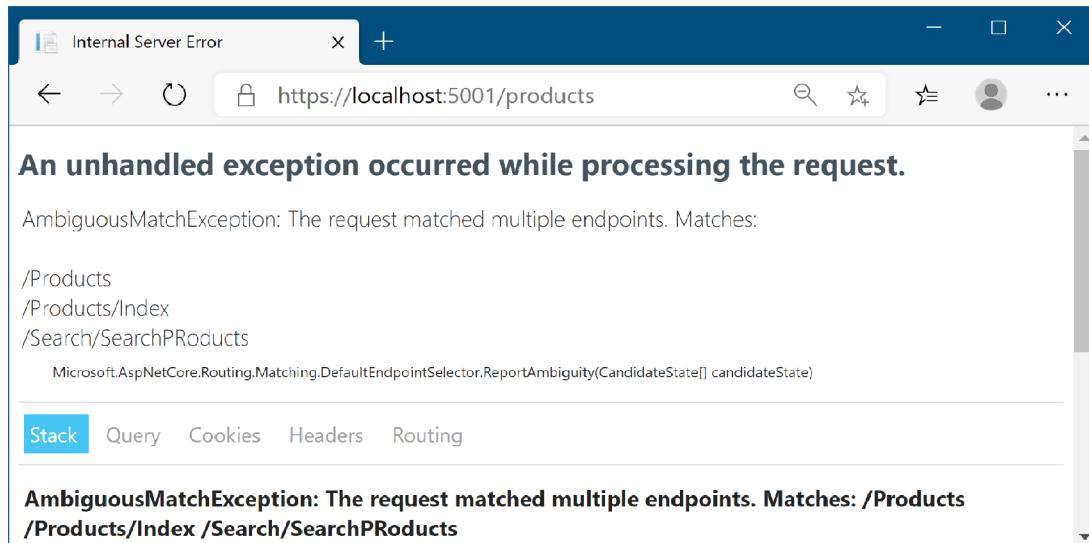


Figure 5.11 If multiple Razor Pages are registered with overlapping route templates, you'll get an exception at runtime when the router can't work out which one to select.

## 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 `EndpointRoutingMiddleware` 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 use default values when a value is 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, such as 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.

# *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'll 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 though, 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).<sup>1</sup> 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 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.

---

<sup>1</sup> POCOs are regular C# classes and objects that live in memory. There are no special inheritance requirements or attributes required to use them with ASP.NET Core.

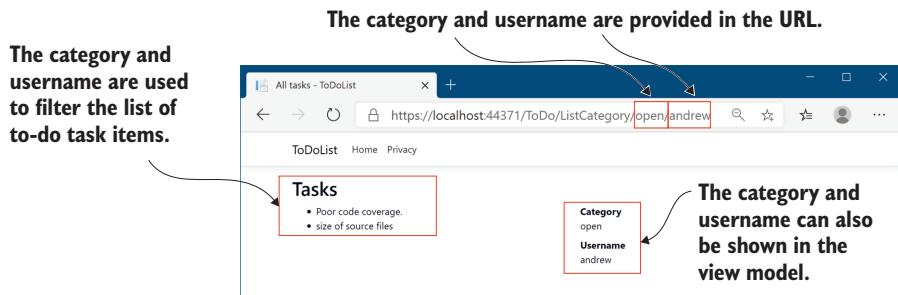


Figure 6.1 A basic to-do list application that displays to-do list items. A user can filter the list of items by changing the category and username parameters in the URL.

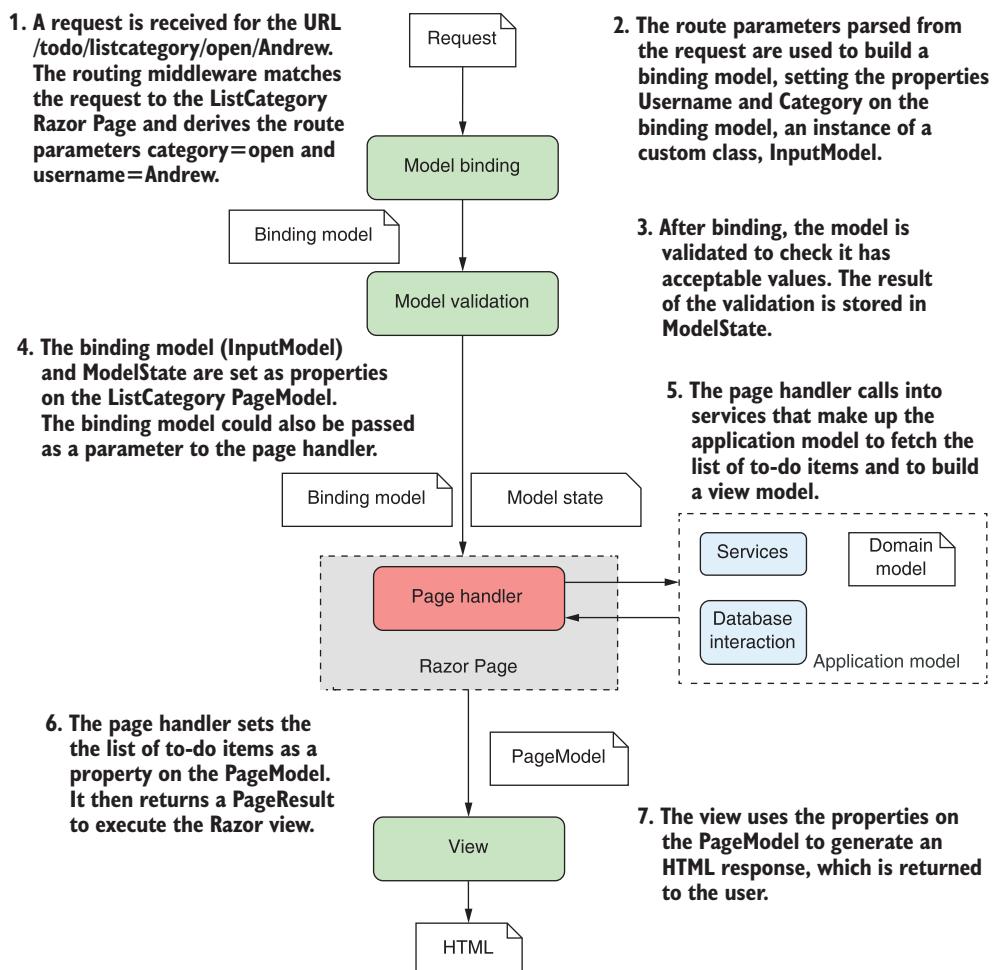


Figure 6.2 The MVC pattern in ASP.NET Core handling a request to view a subset of items in a to-do list Razor Pages application.

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 `/search/{query}` URL, where `{query}` holds their search term:

- *Binding model*—This would take the `{query}` route parameter from the URL and any values posted in the request body (maybe a sort order, or the number of items to show), and bind them to a C# class, which typically acts as a throw-away data transport class. This would be set as a property on the `PageModel` when the page handler is invoked.
- *Application model*—This is the services and classes that perform the logic. When invoked by the page handler, this model would load all the clothes that match the query, applying the necessary sorting and filters, and return the results 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 and 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 long as you don't try to, for example, invoke a page handler from inside a Razor view, 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 data binding 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 following listing. 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]
    public string Category { get; set; }

    [BindProperty(SupportsGet = true)]
    public string Username { get; set; }

    public void OnGet()
    {
    }

    public void OnPost(ProductModel model)
    {
    }
}
```

**Properties decorated with `[BindProperty]` take part in model binding.**

**Properties are not model-bound for GET requests, unless you use `SupportsGet`.**

**Parameters to page handlers are also model-bound when that handler is selected.**

As shown in the preceding 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, 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 follows:

```
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; }
        public ProductModel Model { 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

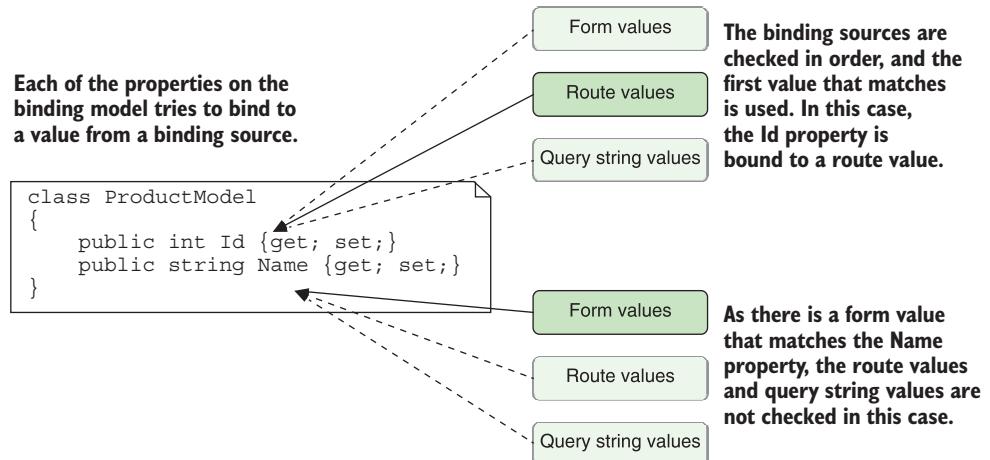


Figure 6.3 Model binding involves mapping values from binding sources, which correspond to different parts of a request.

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.

### 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?

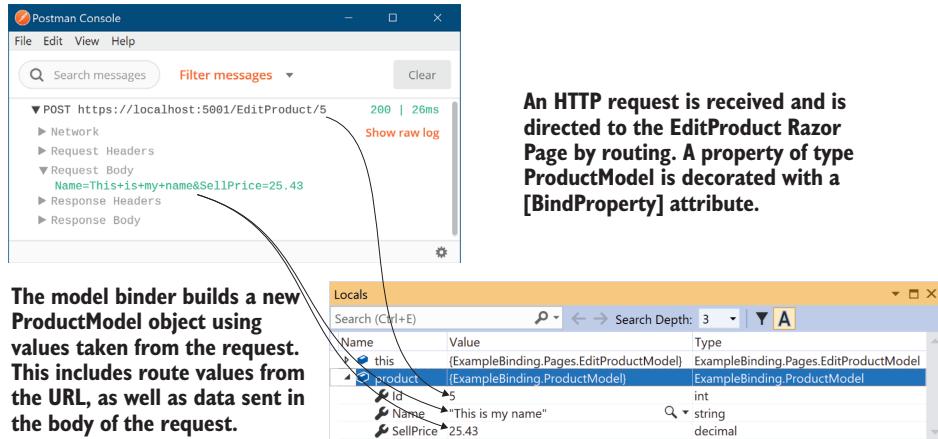
This answer to this question is largely a matter of taste. Setting properties on the `PageModel` and marking them with `[BindProperty]` is the approach you'll see most often in examples. If you use this approach, you'll be able to access the binding model when the view is rendered, as you'll see in chapters 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)
```



**Figure 6.4** Using model binding to create an instance of a model that's used to execute a Razor Page.

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.

**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*

We'll start our 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)
    {
        Square = number * number;
    }

    public int Square { get; set; }
}
```

The method parameter is the binding model.

A more complex example would do this work in an external service, in the application model.

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 /CalculateSquare/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, 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 (parameterless) 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 could unexpectedly have a default value (for value types).

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 then 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 following listing.

#### 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`.<sup>2</sup> 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!

---

<sup>2</sup> TypeConverters can be found in the `System.ComponentModel.TypeConverter` package. You can read more about them in Microsoft's "Type conversion in .NET" documentation: <http://mng.bz/A0GK>.

## 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.

If 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—you 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 it might look something like this.

#### **Listing 6.4 A binding model for capturing a user's details**

```
public class 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; }
```

Now you can 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 creates 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. As for all model binding, the casing of the prefix does not matter.

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 on 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 that are public and settable.

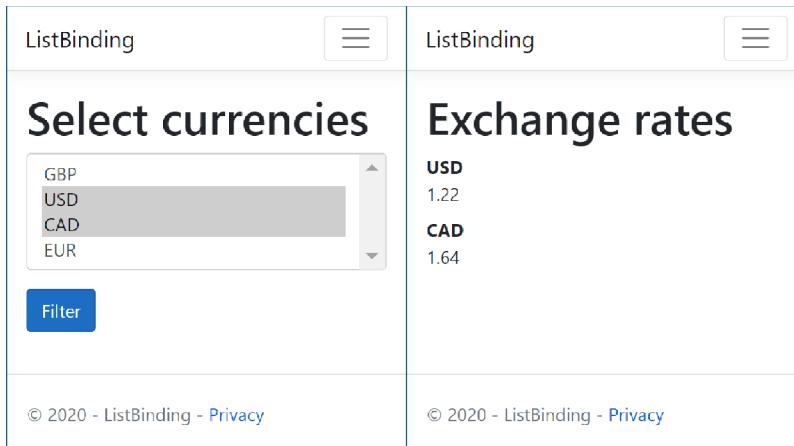
With this technique you can bind complex hierarchical models whose properties are *themselves* complex models. As long as each property exposes a type that can be model-bound, the binder can traverse it with ease.

#### 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.

To achieve this, you could create a page handler that accepts a `List<string>` type, such as

```
public void OnPost(List<string> currencies);
```



**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.

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]=GBP&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]=GBP&[1]=USD.
- currencies—Alternatively, you can omit the index and send currencies as the key for every value, for example, currencies=GBP&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, 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, such as a user uploading 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 in 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 you 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 Microsoft's "Upload files in ASP.NET Core" documentation at <http://mng.bz/S7X>.

**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 following listing.

#### Listing 6.5 Choosing a binding source for model binding

```
public class PhotosModel : PageModel
{
    public void OnPost(
        [FromHeader] string userId,           ← The userId will be bound from
        [FromBody] List<Photo> photos)       ← an HTTP header in the request.
    {
        /* method implementation */
    }
}
```

The list of photo objects 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 be tagged 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 `InputFormatters` 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.<sup>3</sup>
- [BindRequired]—If the parameter was not provided, or was empty, the binder will add a validation error.
- [FromServices]—This is used to indicate the parameter should be provided using dependency injection (see chapter 10 for details).

---

<sup>3</sup> You can use the [BindNever] attribute to prevent mass assignment, as discussed in these two posts on my blog: <http://mng.bz/QvfG> and <http://mng.bz/Vd90>.

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 `DataAnnotations` 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 it’s 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.

### 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 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 for 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 DataAnnotations attributes on its properties.
4. The UserBindingModel and validation ModelState are set on the SaveUser Razor Page, and the page handler is executed.

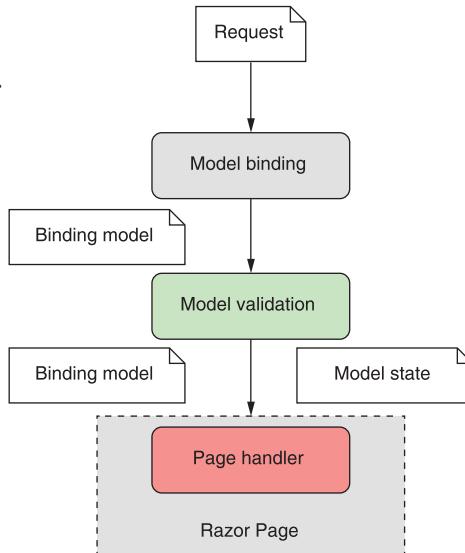


Figure 6.6 Validation occurs after model binding but before the page handler executes. The page handler executes whether or not validation is successful.

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:

- 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 different 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. 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 and .NET 5.0.

### 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]
    [StringLength(100)]
    [Display(Name = "Your name")]
    public string FirstName { get; set; }

    [Required]
    [StringLength(100)]
}
```

Values marked Required must be provided.

The `StringLengthAttribute` sets the maximum length for the property.

Customizes the name used to describe the property

The `StringLengthAttribute` sets the maximum length for the property.

```

[Display(Name = "Last name")]
public string LastName { get; set; }

[Required]
[EmailAddress]
public string Email { get; set; }

[Phone]
[Display(Name = "Phone number")]
public string PhoneNumber { get; set; }
}

```

The diagram illustrates the use of DataAnnotations on a C# class. It shows three properties: LastName, Email, and PhoneNumber. The LastName property has a [Display] attribute with Name set to "Last name". The Email property has both [Required] and [EmailAddress] attributes. The PhoneNumber property has a [Phone] attribute and a [Display] attribute with Name set to "Phone number". Callouts from the left side of the page point to these annotations with the following descriptions:

- Values marked Required must be provided.** Points to the [Required] attribute on the Email property.
- Validates the value of Email is a valid email address** Points to the [EmailAddress] attribute on the Email property.
- Customizes the name used to describe the property** Points to the [Display] attribute on the PhoneNumber property.

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, or 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 (<http://mng.bz/5jxZ>).

- `[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.<sup>4</sup>

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, 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 (<https://github.com/JeremySkinner/FluentValidation>) in place of the DataAnnotations attributes if you prefer. You'll see how to do this in chapter 20.

**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'll 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,

---

<sup>4</sup> 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 on the Twilio blog: <http://mng.bz/xmZe>.

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 following listing 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
{
    [BindProperty]
    public UserBindingModel Input { get; set; } ← The Input property contains
                                                the model-bound data.

    public IActionResult OnPost()
    {
        if (!ModelState.IsValid) ← The binding model is validated before
                               the page handler is executed.
        {
            return Page(); ← If there were validation errors,
                           IsValid will be false.
        }
        ← Validation failed, so redisplay the form
          with errors and finish the method early.

        /* Save to the database, update user, return success */ ← Validation passed, so it's safe
                                                               to use the data provided in
                                                               the model.

        return RedirectToPage("Success");
    }
}
```

If the `ModelState` property 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` property.

**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")]`.

If the request is successful, the page handler returns a `RedirectToPageResult` (using the `RedirectToPage()` helper method) 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.

The screenshot shows a web browser window titled "Checkout - ValidatingWithDataAnnotations". The page content is titled "Checkout" and contains instructions: "Enter the values and click Submit". It has four form fields: "Your name" (containing "Andrew"), "Last name" (empty), "Email" (containing "Not an email"), and "Phone number" (containing "Not a number"). Below each field is a red validation message: "The Last name field is required.", "The Email field is not a valid e-mail address.", and "The Phone number field is not a valid phone number.". A blue "Submit" button is at the bottom. At the very bottom of the page is a copyright notice: "© 2020 - ValidatingWithDataAnnotations - [Privacy](#)".

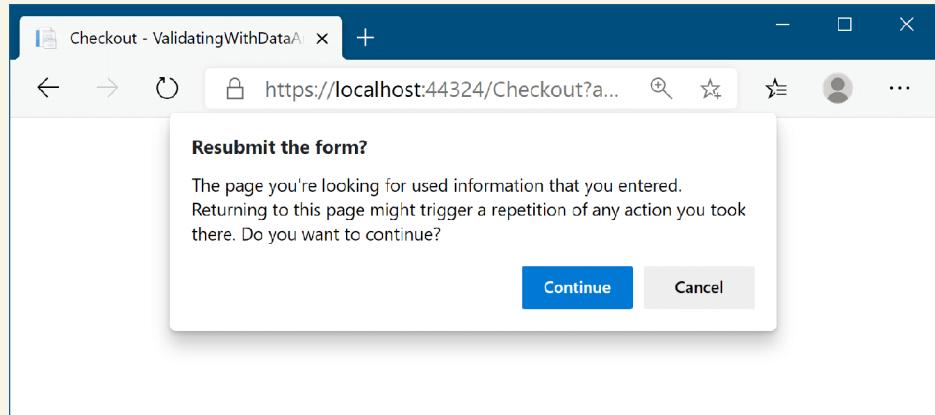
**Figure 6.7** When validation fails, you can redisplay the form to display `ModelState` validation errors to the user. Note the `Your Name` field has no associated validation errors, unlike the other fields.

### POST-REDIRECT-GET

The POST-REDIRECT-GET design pattern is a web development pattern that prevents users from accidentally 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, 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.



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, 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 `RedirectToActionResult` after a successful POST, your application will be safe if the user refreshes the page in their browser.

You might be wondering why ASP.NET Core doesn't handle invalid requests for you automatically—if validation has failed, 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, 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" HTML input type, the browser will automatically stop you from submitting an invalid format, as shown in figure 6.8.

The screenshot shows a simple HTML form with a single input field labeled "Email". The input field contains the text "Not an email". Below the input field, a red error message "The Email field is not valid" is displayed. To the right of the input field, a tooltip-style validation message is shown in a gray box with a yellow exclamation mark icon. The message reads: "Please include an '@' in the email address. 'Not an email' is missing an '@'." A blue "Submit" button is located below the input field.

**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.<sup>5</sup> 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.

<sup>5</sup> HTML5 constraint validation support varies by browser. For details on the available constraints, see the Mozilla documentation (<http://mng.bz/daX3>) and <https://caniuse.com/#feat=constraint-validation>.

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.

If you're building an SPA, 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.

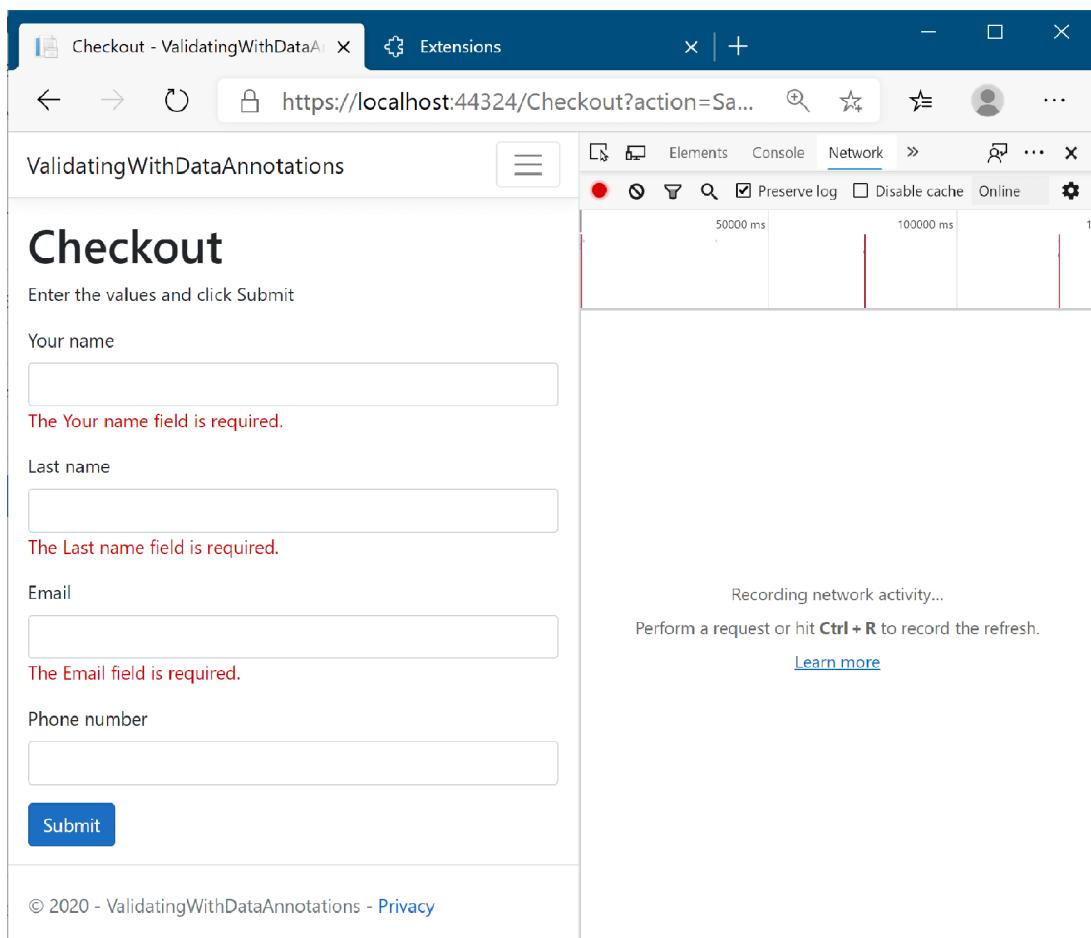


Figure 6.9 With client-side validation, clicking Submit will trigger validation to be shown in the browser before the request is sent to the server. As shown in the right pane, no request is sent.

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`, 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 20.

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; options for where to 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;
    public EditProductModel(ProductService productService)
    {
        _ productService = productService;
    }

    [BindProperty]
    public InputModel Input { get; set; }

    public IActionResult OnGet(int id)
    {
        var product = _ productService.GetProduct(id);
        ←
        ↗ A single property is marked
        ↗ with BindProperty.
        ↗ The id parameter is model-
        ↗ bound from the route
        ↗ template for both OnGet
        ↗ and OnPost handlers.
    }
}
```

```

Input = new InputModel
{
    Name = product.ProductName,
    Price = product.SellPrice,
};
return Page();
}

public IActionResult OnPost(int id)
{
    if (!ModelState.IsValid)
    {
        return Page();
    }

    _productService.UpdateProduct(id, Input.Name, Input.Price);

    return RedirectToPage("Index");
}

public class InputModel
{
    [Required]
    public string Name { get; set; }

    [Range(0, int.MaxValue)]
    public decimal Price { get; set; }
}

```

If the request was not valid, redisplay the form without saving.

Define the InputModel as a nested class in the Razor Page.

Build an instance of the InputModel for editing in the form from the existing product's details.

The id parameter is model-bound from the route template for both OnGet and OnPost handlers.

Redirect to a new page using the POST-REDIRECT-GET pattern.

Update the product in the application model using the ProductService.

This page shows the PageModel for a typical “edit form.” These are very common in many line-of-business applications, among others, and it’s 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 favor 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 I decorate that single property with `[BindProperty]`. Decorating a single property like this makes it harder to forget to add the attribute, and it 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 you’re 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’ll 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.

## Summary

- Razor Pages uses three distinct models, each responsible for a different aspect of a request. The binding model 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 it 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 and 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.



# *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 in your mind:

- *Razor Pages*—Razor Pages generally refers to the page-based paradigm that 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'll read data that your app displays, and they'll 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 *Tag Helpers* that make generating HTML forms easy.

**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.

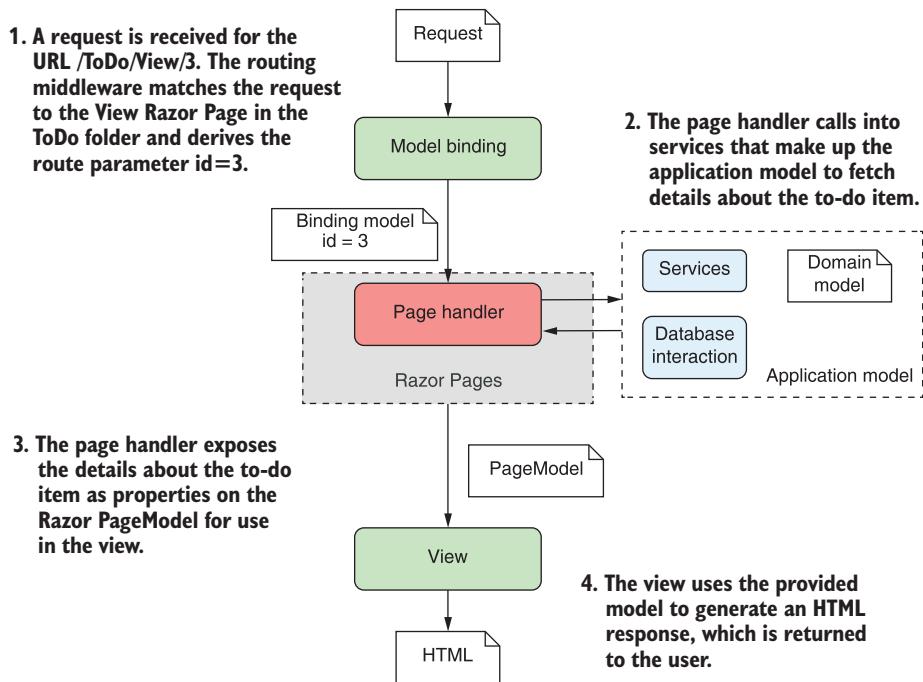
In this chapter we'll be focusing primarily on displaying data and generating HTML using Razor, rather than creating forms. You'll see how to render values from your `PageModel` to the HTML, and how to use C# to control the generated output. Finally, you'll learn how to extract the common elements of 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. I'll then introduce you to 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.



**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 View Razor Page in the ToDo folder.
- 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 to the user via the middleware pipeline.

A common thread throughout this discussion of MVC is the separation of concerns MVC brings, and it's 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.

With Razor templates, generating this sort of dynamic content is simple. For example, listing 7.1 shows a 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.

The PageModel contains the data you wish to display on the page.

```
Model.ExistingUsers = new[] {
    "Andrew",
    "Robbie",
    "Jimmy",
    "Bart"
};
```

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>
}
```

By combining the data in your view model with the Razor markup, HTML can be generated dynamically instead of being fixed at compile time.

Form elements can be used to send values back to the application.

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.

### Listing 7.1 A Razor template to list users, and a form for adding a new user

```
@page
@model IndexViewModel
<div class="row">
<div class="col-md-6">
<form method="post">
    <div class="form-group">
        <label asp-for="NewUser"></label>
        <input class="form-control" asp-for="NewUser" />
        <span asp-validation-for="NewUser"></span>
    </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>
<div class="row">
<div class="col-md-6">
<ul class="list-group">
@foreach (var user in Model.ExistingUsers)
```

Normal HTML is sent to the browser unchanged.

Tag Helpers attach to HTML elements to create forms.

Values can be written from C# objects to the HTML.

C# constructs like for loops can be used in Razor.

```
{  
    <li class="list-group-item d-flex justify-content-between">  
        <span>@user</span>  
        <a class="btn btn-info"  
            asp-page="ViewUser"  
            asp-route-userName="@user">View</a>  
    </li>  
}  
</ul>  
</div>  
</div>
```

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 there are various C# constructs 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 to modify your Razor Pages while your app is running, without having to explicitly stop and rebuild. This can be handy when developing locally, but it's best avoided when you deploy to production.<sup>1</sup>

**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 Angular 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 you can 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'll 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

<sup>1</sup> For details on enabling runtime compilation, including enabling conditional precompilation for production environments, see the documentation: <http://mng.bz/opwy>.

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 makes 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, such as `ToDoItem.cshtml` and `ToDoItem.cshtml.cs`, it's not the filename that's linking them together. But 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 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.<sup>2</sup>

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 following listing 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
 @{
    var tasks = new List<string>
    {
        "Buy milk", "Buy eggs", "Buy bread" };
}
<h1>Tasks to complete</h1>
<ul>
@for(var i=0; i< tasks.Count; i++)
{
    var task = tasks[i];
    <li>@i - @task</li>
}
</ul>
```

**Arbitrary C# can be executed in a template. Variables remain in scope throughout the page.**

**Standard HTML markup will be rendered to the output unchanged.**

**Mixing C# and HTML allows you to dynamically create HTML at runtime.**

The pure HTML sections in this template are in the angle brackets. The Razor engine copies this HTML directly to the output, unchanged, as though you were writing a normal HTML file.

**NOTE** The ability of Razor syntax to know when you are switching between HTML and C# can be both uncanny and infuriating at times. I discuss the details on how to control this transition in section 7.3.

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

<sup>2</sup> 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.learnrazorpageds.com/razor-pages>.

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, the template in listing 7.2 would produce the following HTML.

### Listing 7.3 HTML output produced by rendering a Razor template

```
<h1>Tasks to complete</h1>
<ul>
    <li>0 - Buy milk</li>
    <li>1 - Buy eggs</li>
    <li>2 - Buy bread</li>
</ul>
```

HTML from the Razor template is written directly to the output.

The `<li>` elements are generated dynamically based on the data.

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>
```

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 what you might have on your home page—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 that 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.
- `@inject services`—You can use dependency injection to make services available in your views, though this should normally be used very sparingly. I describe dependency injection and the `@inject` directive in chapter 10.

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
{
    public List<string> Tasks { get; set; }
    public string Title { get; set; }

    public void OnGet(int id)
```

The `PageModel` is passed to the Razor view when it executes.

The public properties can be accessed from the Razor view.

```

        Title = "Tasks for today";
        Tasks = new List<string>
        {
            "Get fuel",
            "Check oil",
            "Check tyre pressure"
        };
    }
}

```

**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 it 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, the preceding template 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]
    public string Title { get; set; }

    public void OnGet()
    {
        Title = "Home Page";
        ViewData["Subtitle"] = "Welcome";
    }
}
```

The code shows annotations explaining the use of the `[ViewData]` attribute and direct assignment to the `ViewData` dictionary.

- An annotation points to the `[ViewData]` attribute on the `Title` property with the text: "Properties marked with the `[ViewData]` attribute are set in the `ViewData`".
- An annotation points to the assignment statement `ViewData["Subtitle"] = "Welcome";` with the text: "The value of `ViewData["Title"]` will be set to "Home Page".
- An annotation points to the assignment statement `ViewData["Subtitle"] = "Welcome";` with the text: "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 I 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 we'll 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 it 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 <http://mng.bz/opj2>.

### 7.3.1 Using C# in Razor templates

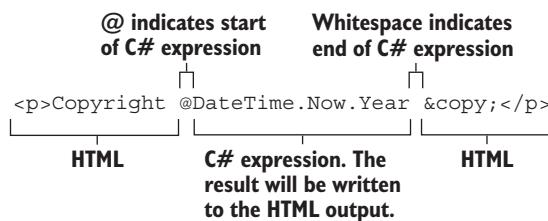
One of the most common requirements when working with Razor templates is to render a value you've calculated in C# to the HTML. For example, you might want to print the current year to use with a copyright statement in your HTML, to give this result:

```
<p>Copyright 2020 ©</p>
```

Or you might want to print the result of a calculation:

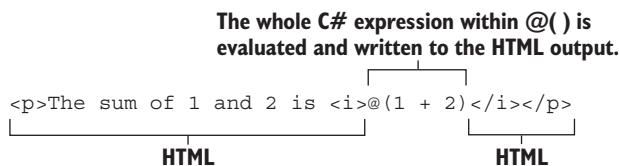
```
<p>The sum of 1 and 2 is <i>3</i></p>
```

You can do this in two ways, depending on the exact C# code you need to execute. If the code is a single statement, 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`.



**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.



**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*.

**TIP** If you want to write a literal @ character, rather than a C# expression, use a second @ character: @@.

Sometimes you'll 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 can 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
```

The `@model` directive indicates  
the type of PageModel in Model.

```

<div>
    @if (Model.IsComplete)
    {
        <strong>Well done, you're all done!</strong>
    }
    else
    {
        <strong>The following tasks remain:</strong>
        <ul>
            @foreach (var task in Model.Tasks)
            {
                <li>@task</li>
            }
        </ul>
    }
</div>

```

The **if** control structure checks the value of the `PageModel`'s `IsComplete` property at runtime.

The **foreach** structure will generate the `<li>` elements once for each task in `Model.Tasks`.

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, 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"
};

```

**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, 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.

```
1  @page "{id}"
2  @model ToDoList.Pages.ViewToDoModel
3
4  <p>
5  @if (Model.ToDo.IsComplete)
6  {
7      <strong>Well done, you're all done!</strong>
8  }
9  else
10 {
11     <strong>The following tasks remain:</strong>
12     <ul>
13         @foreach (var task in Model.ToDo.Tasks)
14         {
15             <li>@task</li>
16         }
17     </ul>
18 }
19 </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**

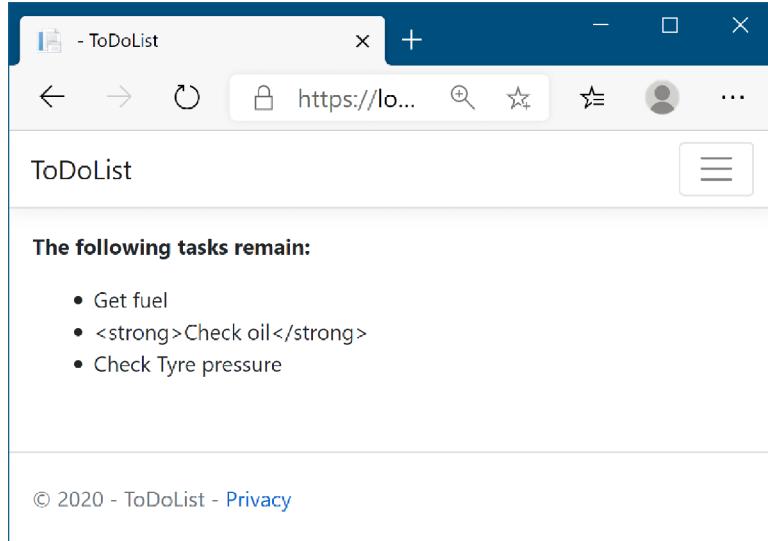
In the previous example, we rendered the list of tasks to HTML by writing the string `task` using the `@task` Razor expression. But what if the `task` variable contains HTML you want to display, so instead of "Check oil" it contains "`<strong>Check oil</strong>`"? If you use a Razor expression to output this as you did previously, 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.



**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, 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, 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.

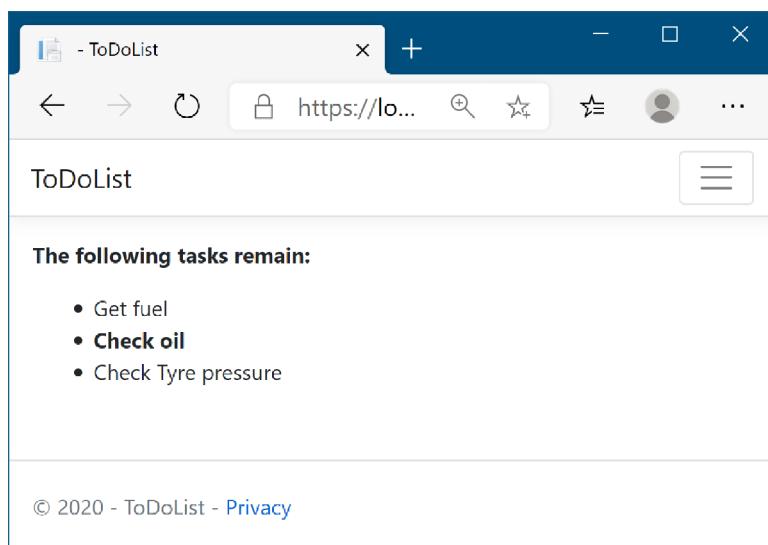


Figure 7.8 The second item, "`<strong>Check oil</strong>`" has been output using `Html.Raw()`, so it hasn't been HTML-encoded. The `<strong>` elements result in the second item being shown in bold instead. Using `Html.Raw()` in this way should be avoided where possible, as it is a security risk.

**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 `HtmlHelper3` methods on the `Html` property, but these have largely been superseded by a cleaner technique: Tag Helpers.

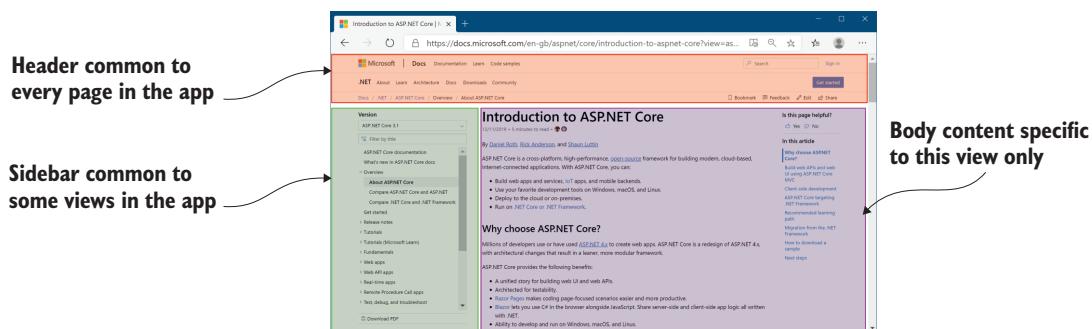
**NOTE** I'll discuss Tag Helpers, and how to use them to build HTML forms, in the next chapter.

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 in your application will also probably reference the same CSS and JavaScript files.



**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.

<sup>3</sup> HTML Helpers are almost obsolete, though they're still available if you prefer to use them.

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 involving 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.

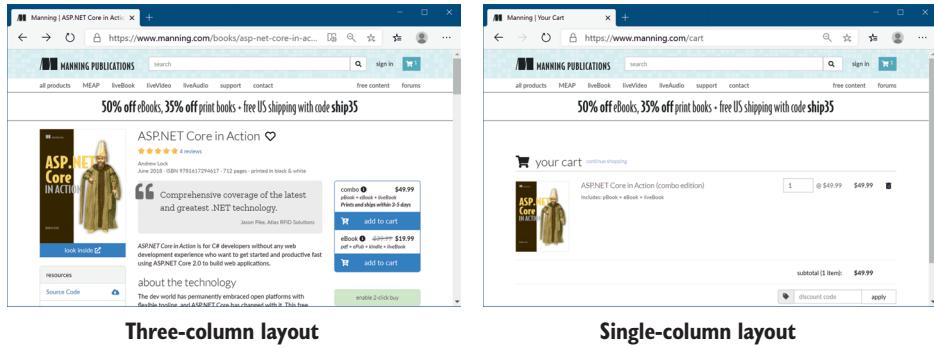


Figure 7.10 The <https://manning.com> website uses different layouts for different parts of the web application. The product pages use a three-column layout, but the cart page uses a single-column layout.

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>
    <link rel="stylesheet" href("~/css/site.css" />
</head>
<body>
    @RenderBody()
</body>
</html>
```

**ViewData is the standard mechanism for passing data to a layout from a view.**

**Elements common to every page, such as your CSS, are typically found in the layout.**

**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.

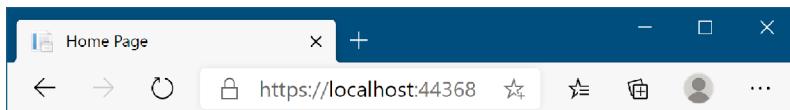


Figure 7.11 The contents of the `<title>` element is used to name the tab in the user's browser, in this case Home Page.

**NOTE** Layout files are not standalone Razor Pages and do not take part in routing, so they do not start with the `@page` directive.

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";
```

**Set the layout for the page to `_Layout.cshtml`.**

```
    ViewData["Title"] = "Home Page";  
}  
<h1>@ViewData["Title"]</h1>  
<p>This is the home page</p>
```

The content in the Razor view to render inside the layout.

ViewData is a convenient way of passing data from a Razor view to 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>  
    <link rel="stylesheet" href="/css/site.css" />  
  </head>  
  <body>  
    <h1>Home Page</h1>  
    <p>This is the home page</p>  
  </body>  
</html>
```

ViewData set in the view is used to render the layout.

The RenderBody call renders the contents of the view.

Judicious use of layouts can be extremely useful in reducing the duplication between pages. 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 {
    <p>This is the sidebar content</p>
}
<p>This is the main content </p>

```

All content inside the braces is part of the Sidebar section, not the main body content.

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, a view *must* declare the given `@section`; if they’re optional, they can be omitted, and the layout will skip them. Skipped sections won’t appear in the rendered HTML. The following 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

```

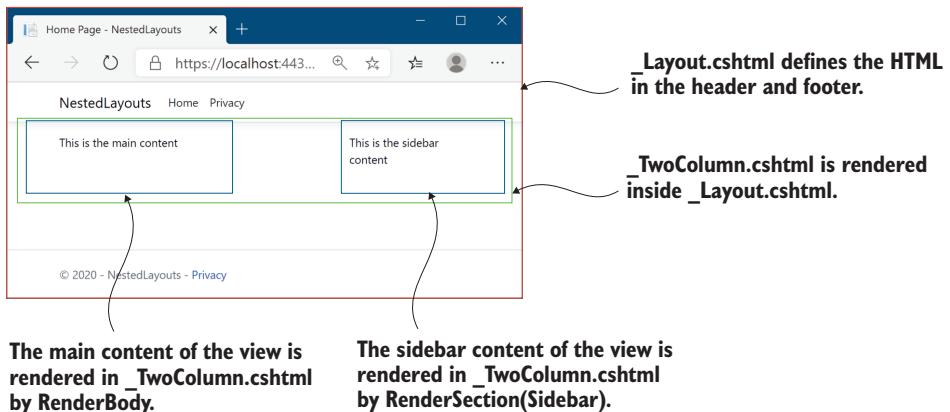
@{
    Layout = "_Layout";           ← This layout is nested
}                                inside a layout itself.
<div class="main-content">      ← Renders all the content
    @RenderBody()                from a view that isn't
</div>                            part of a section
<div class="side-bar">           ← Renders the Sidebar section; if
    @RenderSection("Sidebar", required: true)   the Sidebar section isn't defined
</div>                            in the view, throws an error
@RenderSection("Scripts", required: false) ← 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.

**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.



**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.

Layout files and sections provide a lot of flexibility for building 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` as in the following listing.

### Listing 7.12 Partial view \_ToDo.cshtml for displaying a ToDoItemViewModel

```
@model ToDoItemViewModel
<h2>@Model.Title</h2>
<ul>
    @foreach (var task in Model.Tasks)
    {
        <li>@task</li>
    }
</ul>
```

The content of the partial view, which previously existed in the ViewToDo.cshtml file

Partial views can bind to data in the Model property, like a normal Razor Page uses a PageModel.

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
@model RecentToDoListModel
@foreach(var todo in Model.RecentItems)
```

This is a Razor Page, so it uses the `@page` directive. Partial views do not use `@page`.

The PageModel contains the list of recent items to render.

```
{      <partial name="_ToDo" model="todo" /> }
```

Loop through the recent items. todo is a `ToDoItemViewModel`, as required by the partial view.

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, such as `_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.cshtml`, the framework would look in the following places for a file called `_ToDo.cshtml`:

- `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, that's the only place the framework will look.<sup>4</sup>

**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 20.

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.

#### 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";  
}
```

<sup>4</sup> 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: <http://mng.bz/nM9e>.

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 \_ViewImports.cshtml instead of in your Razor Pages.

##### **Listing 7.14 A typical \_ViewImports.cshtml file importing additional namespaces**

```
@using WebApplication1  
@using WebApplication1.Pages  
@using WebApplication1.Models  
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

The default namespace of your application and the Pages folder

Add this directive to avoid placing it in every view.

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, such as to set the Layout property, 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 non-default layout, you can override it by setting the value in the Razor Page itself.

##### **Listing 7.15 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 partial 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: <http://mng.bz/vzB1>. 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'll look at how you can render Razor views from your MVC controller actions, and how the framework locates the correct Razor view to render.

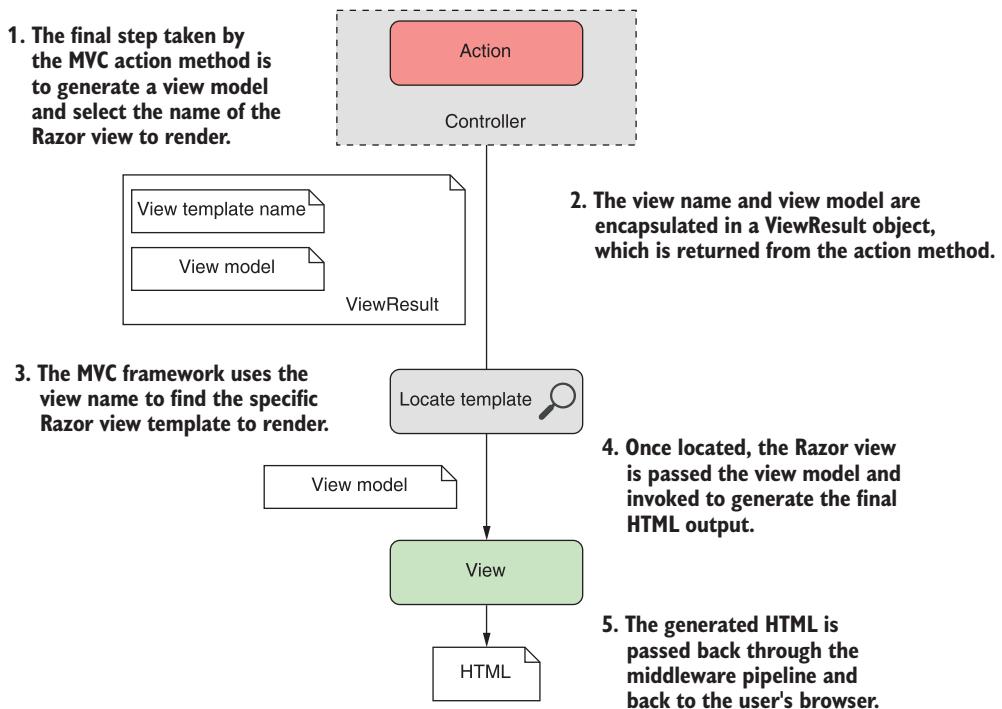
## 7.5 Selecting a view from an MVC controller

This section covers

- How MVC controllers use `ViewResults` 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, 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.x. 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, 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.



**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 4.6 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 ViewResults 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 then 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.

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.

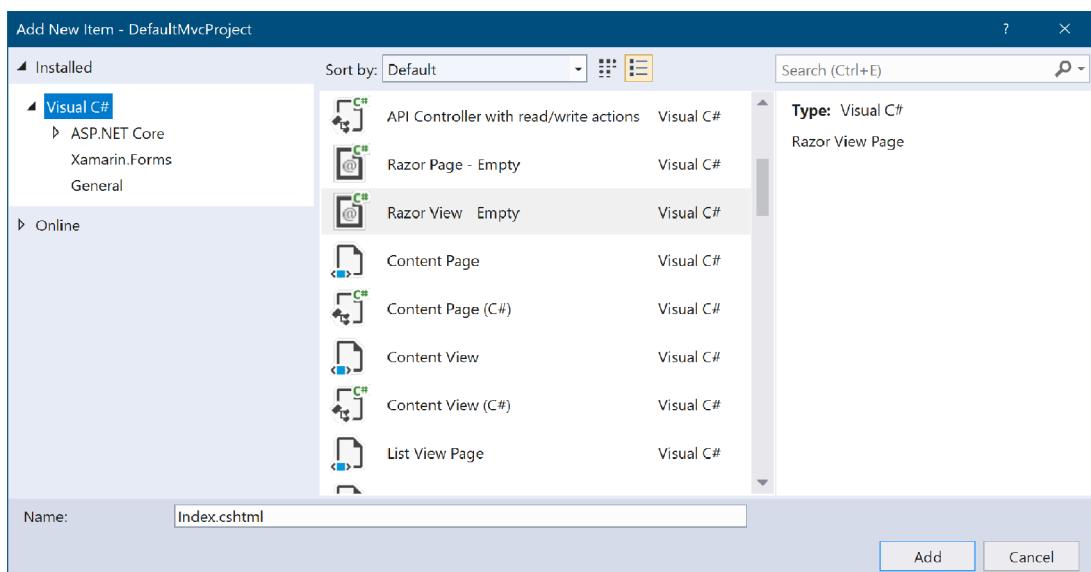


Figure 7.14 The Add New Item dialog box. Choosing Razor View - Empty will add a new Razor view template file to your application.

In the simplest case you can call the `View` method without any arguments, as shown in the following listing. 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 7.16 Returning `ViewResult` from an action method using default conventions**

```
public class HomeController : Controller
{
    public IActionResult Index()
    {
        return View();
    }
}
```

Inheriting from the `Controller` base class makes the `View` helper methods available.

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.

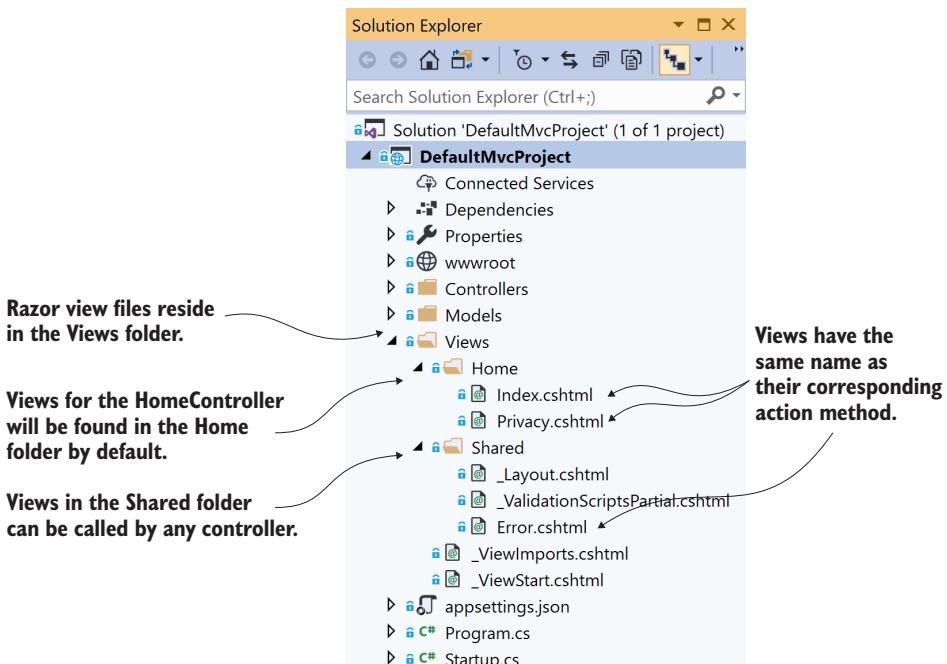


Figure 7.15 View files are located at runtime based on naming conventions. Razor view files reside in a folder based on the name of the associated MVC controller and are named with the name of the action method that requested them. Views in the Shared folder can be used by any controller.

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")`, 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, such as 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.

**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, 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 it's accessed in exactly the same way as for Razor Pages; using the `Model` property. The following listing shows two examples of passing a view model to a view.

#### Listing 7.17 Returning `ViewResult` from an action method using default conventions

```
public class ToDoController : Controller
{
    public IActionResult Index()
    {
        var listViewModel = new ToDoListModel();
        return View(listViewModel);
    }
}
```

Creating an instance of the view model to pass to the Razor view.

The view model is passed as an argument to `View`.

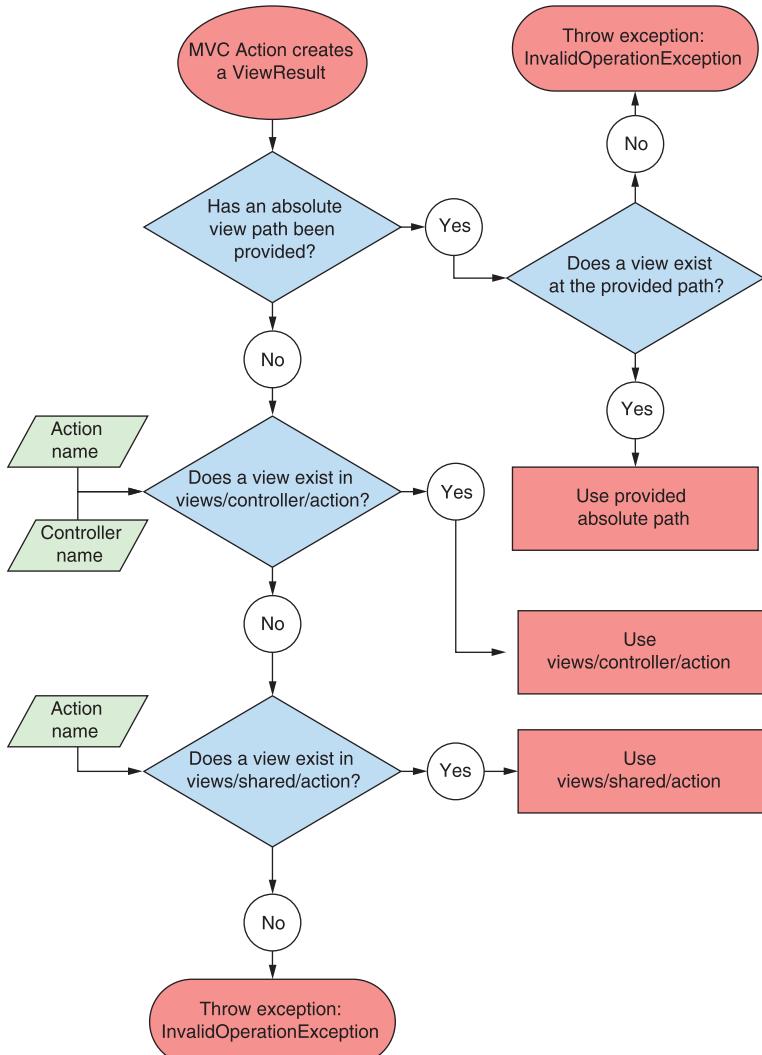
```

public IActionResult View(int id)
{
    var viewModel = new ViewToDoModel();
    return View("ViewToDo", viewModel);
}

}

```

You can provide the view template name at the same time as the view model.



**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!

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 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!

## 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.

# *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 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 attributes 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 legwork for you, as shown in figure 8.1.

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.

Figure 8.1 The currency converter application forms, built using Tag Helpers. The labels, drop-downs, input elements, and validation messages are all generated using Tag Helpers.

## 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
@model ConvertModel
```

This is the view for the Razor Page `Convert.cshtml`.  
The Model type is `ConvertModel`.

```

<form method="post">
    <div class="form-group">
        >   <label asp-for="CurrencyFrom"></label>
            <input class="form-control" asp-for="CurrencyFrom" />
            <span asp-validation-for="CurrencyFrom"></span>
        </div>
        <div class="form-group">
            >   <label asp-for="Quantity"></label>
                <input class="form-control" asp-for="Quantity" />
                <span asp-validation-for="Quantity"></span>
            </div>
            <div class="form-group">
                >   <label asp-for="CurrencyTo"></label>
                    <input class="form-control" asp-for="CurrencyTo" />
                    <span asp-validation-for="CurrencyTo"></span>
                </div>
                <button type="submit" class="btn btn-primary">Submit</button>
            </div>
        </form>
    
```

**asp-for on Labels generates the caption for labels based on the view model.**

**asp-for on Inputs generates the correct type, value, name, and validation attributes for the model.**

**Validation messages are written to a span using Tag Helpers.**

```

graph TD
    L1[asp-for on Labels generates the caption for labels based on the view model.] --> L2[asp-for on Inputs generates the correct type, value, name, and validation attributes for the model.]
    L2 --> S[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.<sup>1</sup>

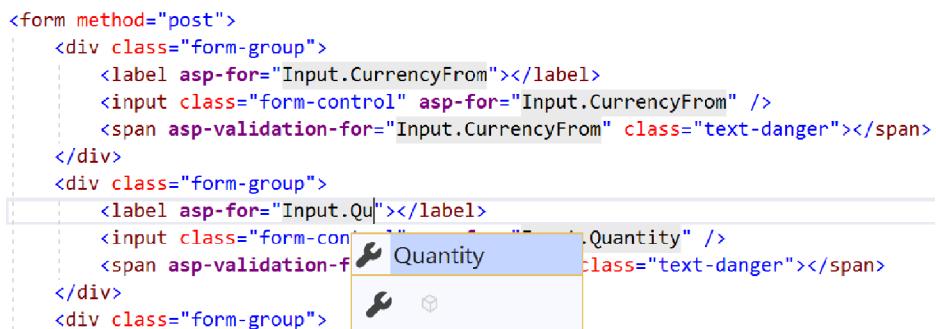


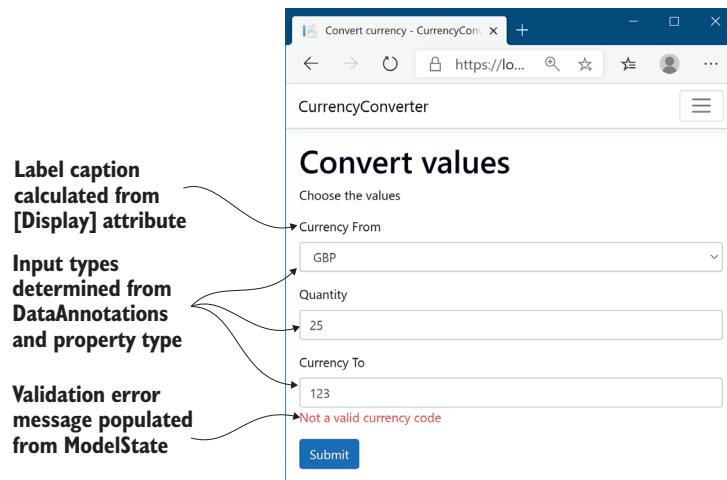
Figure 8.2 In Visual Studio, Tag Helpers are distinguishable from normal elements by being bold and a different color, C# is shaded, and IntelliSense is available.

<sup>1</sup> Other editors like Visual Studio Code, JetBrains Rider, and Visual Studio for Mac also include syntax highlighting and IntelliSense support.

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 POSTed 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.<sup>2</sup>



**Figure 8.3** Tag Helpers hook into the metadata provided by `DataAnnotations` attributes, 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 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!),

---

<sup>2</sup> To learn more about the internals of Tag Helpers, read the documentation at <http://mng.bz/Idb0>.

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.<sup>3</sup> 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.

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 `DataAnnotations` 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`.

<sup>3</sup> A good example is Damian Edwards' (of the ASP.NET Core team) Tag Helper pack: <https://github.com/DamianEdwards/TagHelperPack>.

**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.<sup>4</sup>

#### 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}")]
    [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.

---

<sup>4</sup> You can read more about over-posting attacks on my blog at <http://mng.bz/RXw0>.

The screenshot shows a browser window titled "Checkout - CurrencyConverter". The URL in the address bar is "https://lo...". The main content is a "Checkout" form. It includes fields for "Your name" (containing "Andrew"), "Last name" (empty), "Email" (containing "test@example.com"), and "Phone number" (empty). Below the "Last name" field, a red validation message says "The Last name field is required.". At the bottom is a blue "Submit" button.

**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  
@{
```

The `CheckoutModel` is the `PageModel`, which exposes a `UserBindingModel` on the `Input` property.

```

    ViewData["Title"] = "Checkout";
}
<h1>@ViewData["Title"]</h1>
<form asp-page="Checkout">
    <div class="form-group">
        <label asp-for="Input.FirstName"></label>
        <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" />
        <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>
    </div>
    <button type="submit" class="btn btn-primary">Submit</button>
</form>

```

**The Validation Tag Helper displays error messages associated with the given property.**

**Form Tag Helpers use routing to determine the URL the form will be posted to.**

**The Label Tag Helper uses DataAnnotations on a property to determine the caption to display.**

**The Input Tag Helper uses DataAnnotations to determine the type of input to generate.**

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-*` attributes 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.">
    </div>

```

```
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, 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 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 an `asp-page` Tag Helper attribute:

```
<form asp-page="Checkout">
```

This results in `action` and `method` attributes being added to the final HTML, indicating which URL the form should be sent to when it's 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 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 provided 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` that 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 (it's shown with a fixed value of 5 in the following example, but it 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 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 allow actions to be executed 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 you generally 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. Consider this model in which the `FirstName` property has a `[Display]` attribute, but the `Email` property doesn't:

```
public class UserModel
{
    [Display(Name = "Your name")]
    public string FirstName { get; set; }
    public string Email { get; set; }
}
```

The following Razor

```
<label asp-for="FirstName"></label>
<label asp-for="Email"></label>
```

would generate this 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 who 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 this 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. 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, as shown in figure 8.5.

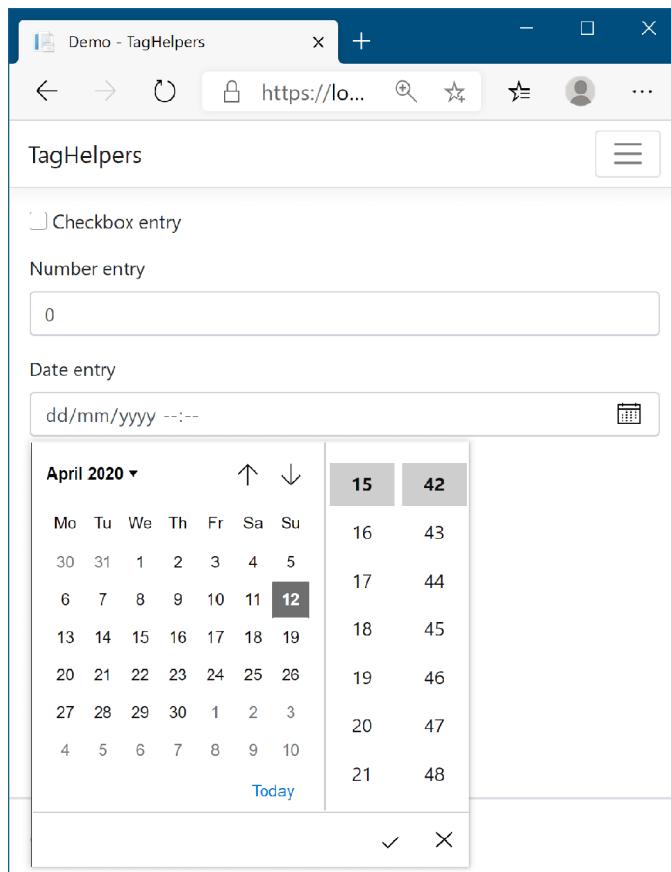


Figure 8.5 Various input element types. The exact way in which each type is displayed varies by browser.

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 `[EmailAddress]` 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.<sup>a</sup>

<sup>a</sup> You can also load these files from a content delivery network (CDN). If you wish to take this approach, you should consider scenarios where the CDN is unavailable or compromised, as I discuss in this blog post: <http://mng.bz/2e6d>.

The Input Tag Helper tries to pick the most appropriate template for a given property based on DataAnnotations 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 element in your Razor template. 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
<code>byte</code> , <code>int</code> , <code>short</code> , <code>long</code> , <code>uint</code>	Property type	<code>number</code>
<code>decimal</code> , <code>double</code> , <code>float</code>	Property type	<code>text</code>
<code>bool</code>	Property type	<code>checkbox</code>
<code>string</code>	Property type, <code>[DataType(DataType.Text)]</code> attribute	<code>text</code>
<code>HiddenInput</code>	<code>[HiddenInput]</code> attribute	<code>hidden</code>
<code>Password</code>	<code>[Password]</code> attribute	<code>password</code>
<code>Phone</code>	<code>[Phone]</code> attribute	<code>tel</code>
<code>EmailAddress</code>	<code>[EmailAddress]</code> attribute	<code>email</code>
<code>Url</code>	<code>[Url]</code> attribute	<code>url</code>
<code>Date</code>	<code>DateTime</code> property type, <code>[DataType(DataType.Date)]</code> attribute	<code>date</code>

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.

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 it's 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 drop-downs 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.

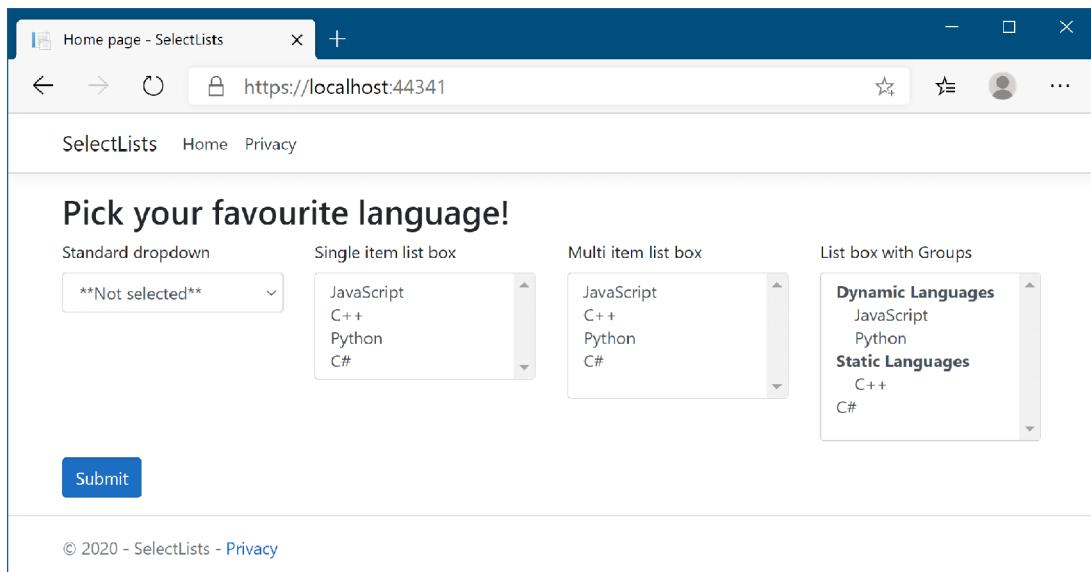


Figure 8.6 Some of the many ways to display `<select>` elements using the `Select` Tag Helper.

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 shown 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 drop-downs and list boxes

```
public class SelectListListModel : PageModel
{
    [BindProperty]
    public class InputModel Input { get; set; }

    public IEnumerable<SelectListItem> Items { get; set; }
        = new List<SelectListItem>
    {
        new SelectListItem{Value= "csharp", Text="C#"},
        new SelectListItem{Value= "python", Text= "Python"},
        new SelectListItem{Value= "cpp", Text="C++"},
        new SelectListItem{Value= "java", Text="Java"},
        new SelectListItem{Value= "js", Text="JavaScript"},
        new SelectListItem{Value= "ruby", Text="Ruby"},
    };
}

public class InputModel
```

**The InputModel for binding the user's selections to the select boxes**

**The list of items to display in the select boxes**

```

    public string SelectedValue1 { get; set; }
    public string SelectedValue2 { get; set; }
    public IEnumerable<string> MultiValues { get; set; }
}
}

These properties will hold the values selected by the single-selection select boxes.

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 value selected by the user. They're model-bound to the value selected from the drop-down/listbox and used to preselect 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 hardcoded.

**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.GetEnum-SelectList<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"
       asp-items="Model.Items"></select>
<select asp-for="Input.SelectedValue2"
       asp-items="Model.Items" size="4"></select>

Creates a standard drop-down select list by binding to a standard property in asp-for

Creates a single-select list box of height 4 by providing the standard HTML size attribute

```

```
<select asp-for="Input.MultiValues"
    asp-items="Model.Items"></select>
```

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 `SelectListItems`.

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 SelectListModel : PageModel
{
    [BindProperty]
    public IEnumerable<string> SelectedValues { get; set; }
    public IEnumerable<SelectListItem> Items { get; set; }

    public SelectListModel()
    {
        var dynamic = new SelectListGroup { Name = "Dynamic" };
        var stat = new SelectListGroup { Name = "Static" };
        Items = new List<SelectListItem>
        {
            new SelectListItem {
                Value= "js",
                Text="Javascript",
                Group = dynamic
            },
            new SelectListItem {
                Value= "cpp",
                Text="C++",
                Group = stat
            },
            new SelectListItem {
```

Used to hold the selected values where multiple selections are allowed

Initializes the list items in the constructor

Creates a single instance of each group to pass to SelectListItems

Sets the appropriate group for each SelectListItem

```
        Value= "python",
        Text="Python",
        Group = dynamic
    },
    new SelectListItem {
        Value= "csharp",
        Text="C#",
    }
}
};
```

← Sets the appropriate group for each SelectListItem

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. This Razor template,

```
@page
@model SelectListsModel
<select asp-for="SelectedValues" asp-items="Model.Items"></select>
```

would be rendered to HTML as follows:

```
<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 that no value has been selected, as shown in figure 8.7. Without this extra option, the default `<select>` drop-down will always have a value, and it will default to the first item in the list.

You can achieve this in one of two ways: you could either add the “not selected” option to the available `SelectListItems`, 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 that you might want to create a partial view to encapsulate this logic.

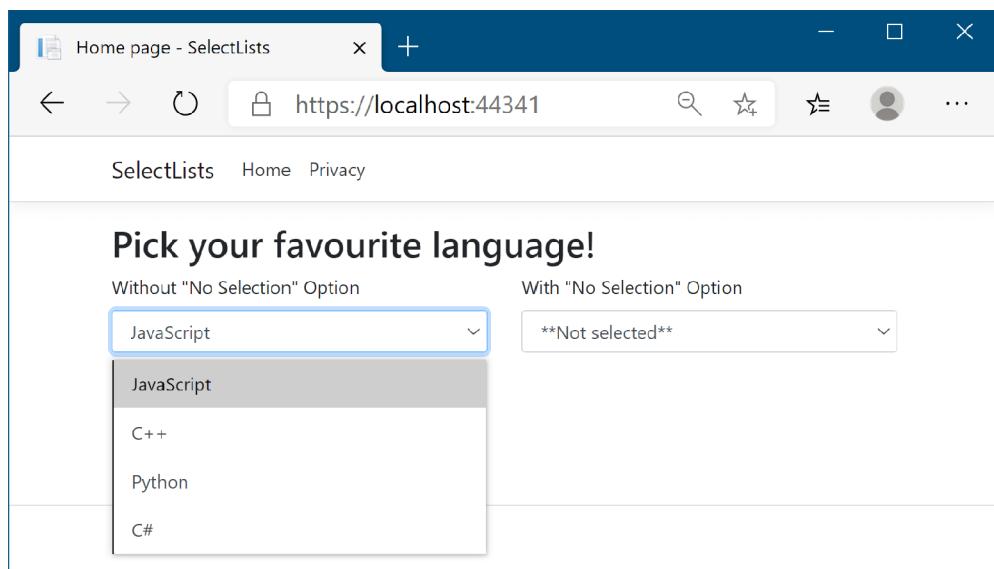


Figure 8.7 Without a “not selected” option, the `<select>` element will always have a value. This may not be the behavior you desire if you don’t want an `<option>` to be selected by default.

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.



**Figure 8.8** Validation messages can be shown in an associated `<span>` by using the Validation Message Tag Helper.

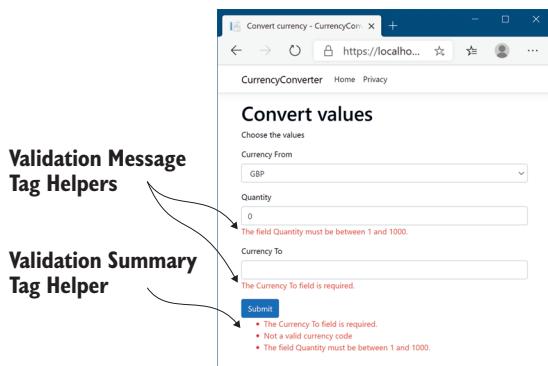
Any errors associated with the `Email` property stored in `ModelState` will be rendered in the element body, and the element will have appropriate attributes to hook into jQuery validation:

```
<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 `ModelError` errors.



**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 it 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)
        {
            ModelState.AddModelError(
                string.Empty,
                "Cannot convert currency to itself");
        }
        if (!ModelState.IsValid)
        {
            return Page();
        }
        //store the valid values somewhere etc
        return RedirectToPage("Checkout");
    }
}
```

Annotations for Listing 8.8:

- An annotation pointing to the line `if (Input.CurrencyFrom == Input.CurrencyTo)` with the text "Can't convert currency to itself".
- An annotation pointing to the line `ModelState.AddModelError(string.Empty, "Cannot convert currency to itself");` with the text "Adds model-level error, not tied to a specific property, by using empty key".
- An annotation pointing to the line `if (!ModelState.IsValid)` with the text "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 20.

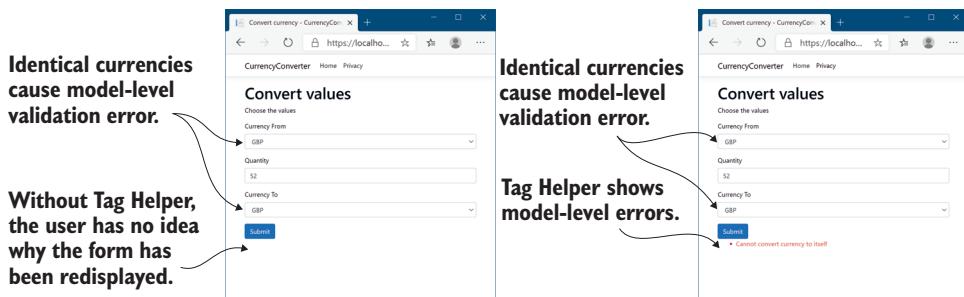


Figure 8.10 Model-level errors are only displayed by the Validation Summary Tag Helper. Without one, users won't have any indication that there were errors on the form, and so won't be able to correct them.

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 `ActionResults`. 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'll show how you can 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 following listing.

#### 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>
</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. These are the most commonly used with Razor Pages:

- `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.<sup>5</sup>
- `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.

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

---

<sup>5</sup> I won't cover areas in detail in 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>.

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, 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, 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 on whether 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, as shown in figure 8.11.

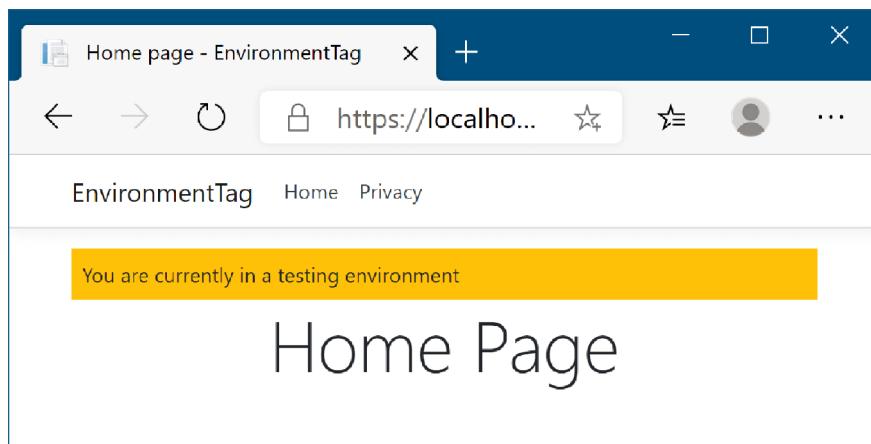


Figure 8.11 The warning banner will be shown whenever you're running in a testing environment, to make it easy to distinguish from production.

**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, you could use something like this:

```
@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.

## 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 customize 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 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.

# *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 400 Bad Requests 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-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).

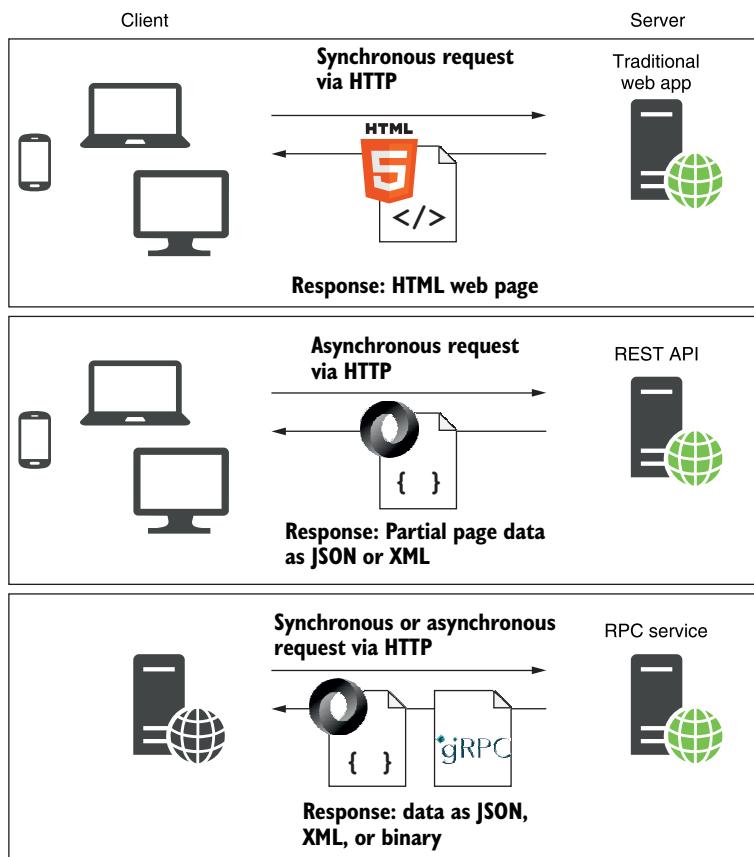
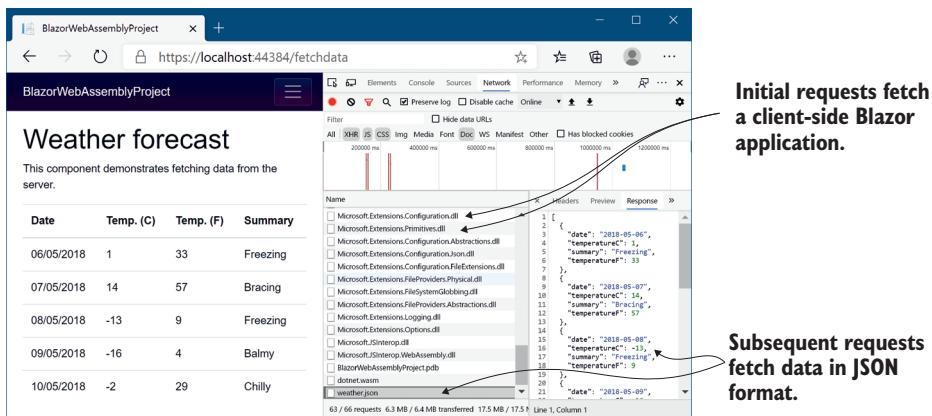


Figure 9.1 Modern developers have to consider a number of different consumers of their applications. As well as traditional users with web browsers, these could be SPAs, mobile applications, or other apps.

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 multiple URLs that can be used to access or change data on a server. It's typically accessed using HTTP.

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



**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.

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 backend 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, you need to develop a mobile application, or you already have an SPA build-pipeline configured, you'll most likely want to add Web APIs that they can use 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 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, you have no need to call your application from outside a web browser, or you don't want or need the effort involved in configuring a client-side application, you probably won't need Web APIs initially. You can stick to generating your UI using Razor Pages and you'll no doubt be highly productive!

**NOTE** Although there has been an industry 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 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, as well as Blazor:

- *React in Action* by Mark Tielens Thomas (Manning, 2018)
- *Angular in Action* by Jeremy Wilken (Manning, 2018)
- *Vue.js in Action* by Erik Hanchett with Benjamin Listwon (Manning, 2018)
- *Blazor in Action*, by Chris Sainty (Manning, 2021)

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**

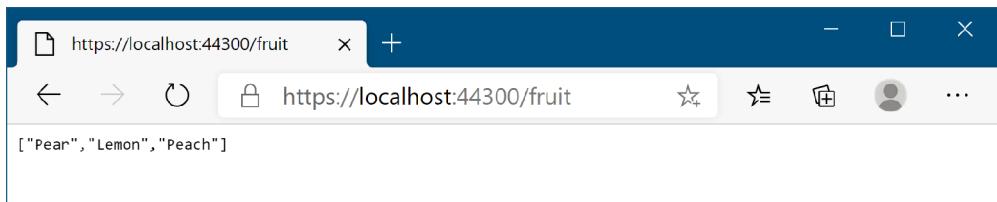
In this section you'll learn how to create an ASP.NET Core Web API project and create your first Web API controllers. You'll see how to use controller action methods to handle HTTP requests, and how to use `ActionResults` to generate a response.

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 their action methods, or an `IActionResult` such as `StatusCodesResult`, 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`.



**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.

**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.

ASP.NET Core 5.0 apps also include a useful endpoint for testing and exploring your Web API project in development called Swagger UI, as shown in figure 9.4. This lets you browse the endpoints in your application, view the expected responses, and experiment by sending requests.

**NOTE** Swagger UI is based on the industry standard OpenAPI specification (previously called Swagger, [www.openapis.org](http://www.openapis.org)), which is enabled by default in Web API apps. OpenAPI provides a way of documenting your API, so that you can automatically generate clients for interacting with it in dozens of different languages. For more on OpenAPI and Swagger in ASP.NET Core apps, see Microsoft’s documentation: <http://mng.bz/QmjR>.

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 box, select the ASP.NET Core Web API template, as shown in figure 9.5. If you’re using the CLI, you can create a similar template using `dotnet new webapi -o WebApplication1`.

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 Web API project

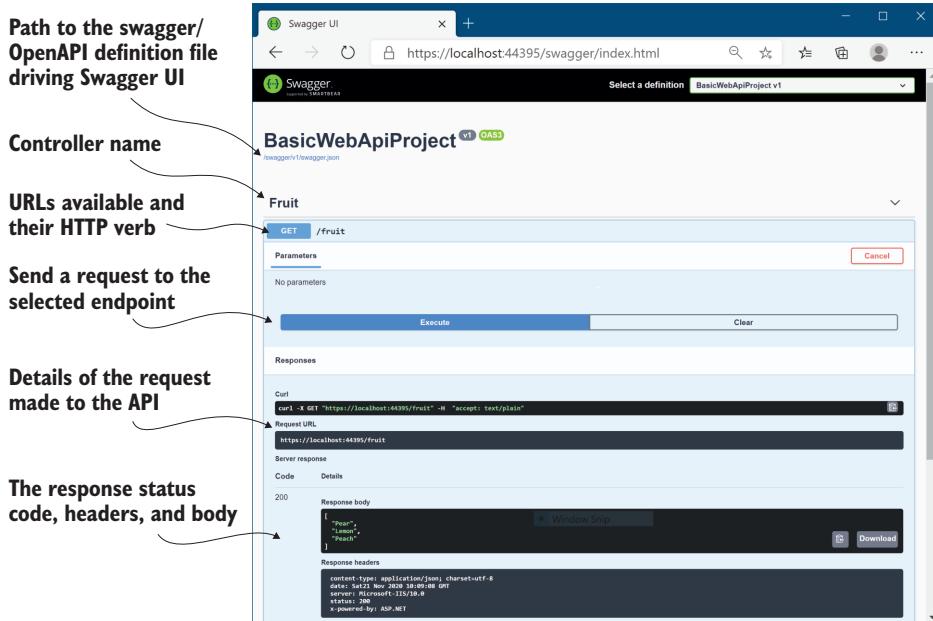


Figure 9.4 The default ASP.NET Core 5.0 Web API templates are configured to use Swagger UI. Swagger UI provides a convenient web page for exploring and interacting with your Web API. By default, this UI is only enabled in development, but you can also enable it in production if you wish.

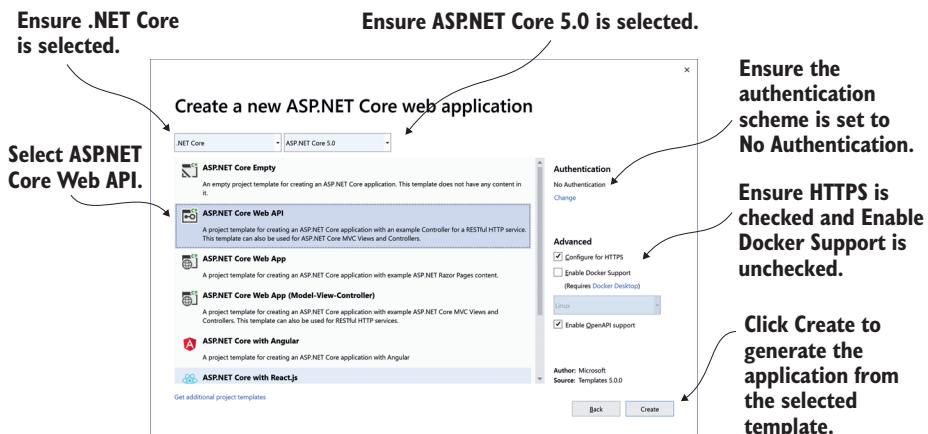


Figure 9.5 The web application template screen. This screen follows on from the Configure Your Project dialog box and lets you customize the template that will generate your application. Choose the ASP.NET Core Web API template to create a Web API project.

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. The default Web API template also adds the Swagger services and endpoints required by the Swagger UI shown previously in figure 9.4. 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();           ← AddControllers adds the
        services.AddSwaggerGen(c =>          necessary services for API
        {
            c.SwaggerDoc("v1", new OpenApiInfo {
                Title = "DefaultApiTemplate", Version = "v1" });
        });
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
    {
        if (env.IsDevelopment())
        {
            app.UseDeveloperExceptionPage();
            app.UseSwagger();
            app.UseSwaggerUI(c => c.SwaggerEndpoint(
                "/swagger/v1/swagger.json", "DefaultApiTemplate v1"));
        }

        app.UseHttpsRedirection();
        app.UseRouting();
        app.UseAuthorization();

        app.UseEndpoints(endpoints =>
        {
            endpoints.MapControllers();       ← MapControllers configures
        });
    }
}
```

**Adds services required to generate the Swagger/OpenAPI specification document**

**Adds Swagger UI middleware for exploring your Web API**

**AddControllers adds the necessary services for API controllers to your application.**

**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 by 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]
public class FruitController : ControllerBase
{
    List<string> _fruit = new List<string>
    {
        "Pear",
        "Lemon",
        "Peach"
    };

    [HttpGet("fruit")]
    public IEnumerable<string> Index()
    {
        return _fruit;
    }
}
```

The code defines a `FruitController` class that inherits from `ControllerBase`. It contains a private list `_fruit` with three items: "Pear", "Lemon", and "Peach". An `Index()` action method is defined with the `[HttpGet("fruit")]` attribute, which returns the list of fruits.

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 to 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

```
Defines the route template  
for the action method
[HttpGet("fruit/{id}")]
public ActionResult<string> View(int id)
{
    if (id >= 0 && id < _fruit.Count)
    {
        return _fruit[id];
    }
    return NotFound();
}

NotFound returns a NotFoundResult,  
which will send a 404 status code.
```

The action method returns an `ActionResult<string>`, so it can return a string or an `ActionResult`.

An element can only be returned if the id value is a valid `_fruit` element index.

Returning the data directly will return the data with a 200 status code.

In the successful path for the action method, the `id` parameter has a value greater than 0 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 9.6. You could also have returned the data using an `OkResult`, by returning `Ok(_fruit[id])`, using the `Ok` helper method<sup>1</sup> on the `ControllerBase` class—under the hood, the result is identical.

<sup>1</sup> 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 <http://mng.bz/goG8>.

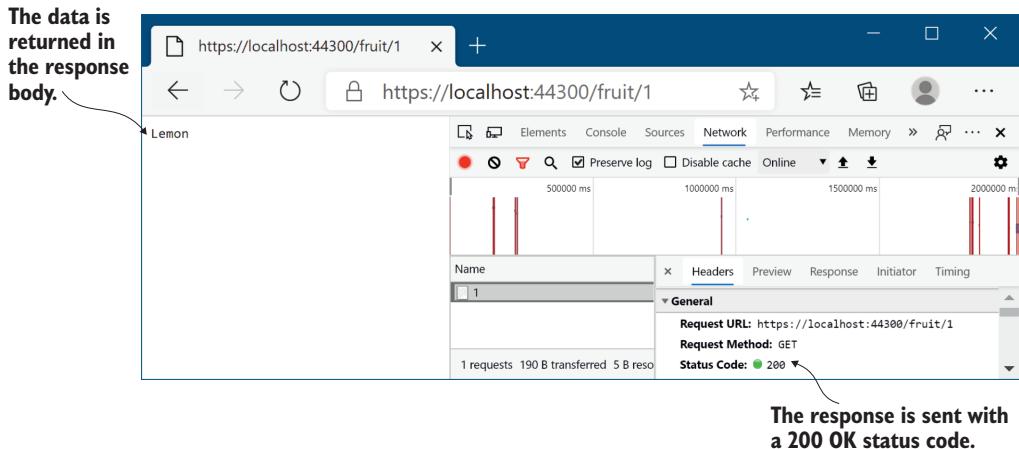


Figure 9.6 Data returned from an action method is serialized into the response body, and it generates a response with status code 200 OK.

If the `id` is outside the bounds of the `_fruit` list, 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 9.7.

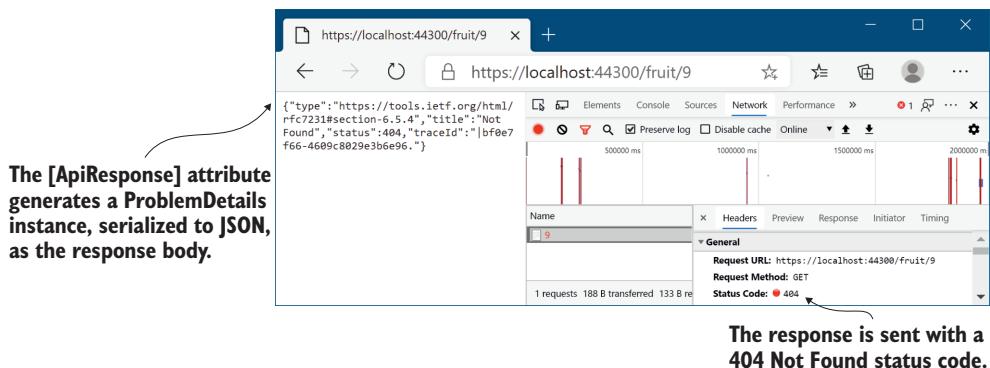


Figure 9.7 The `[ApiController]` attribute converts error responses (in this case a 404 response) into the standard `ProblemDetails` format.

**DEFINITION** `ProblemDetails` is a web specification for providing machine-readable errors for HTTP APIs. You'll learn more about them in section 9.5.

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.<sup>2</sup> 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.

You're free to return any type of `ActionResult` from your Web API controllers, but you'll commonly return `StatusCodesResult` instances, which set the response to a specific status code, with or without associated data. `NotFoundResult` and `OkResult` both derive from `StatusCodesResult`, 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.

**TIP** You learned about various `ActionResults` in chapter 4. `BadRequestResult`, `OkResult`, and `NotFoundResult` all inherit from `StatusCodesResult` 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.

---

<sup>2</sup> You can see how this is achieved in the source code: <http://mng.bz/5j27>.

In ASP.NET Core, that all changed. You now have a *single* framework that 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 looks 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.

By now you're, I hope, nicely familiar with how ASP.NET Core handles a request. But just in case you're not, figure 9.8 shows how the framework handles a typical

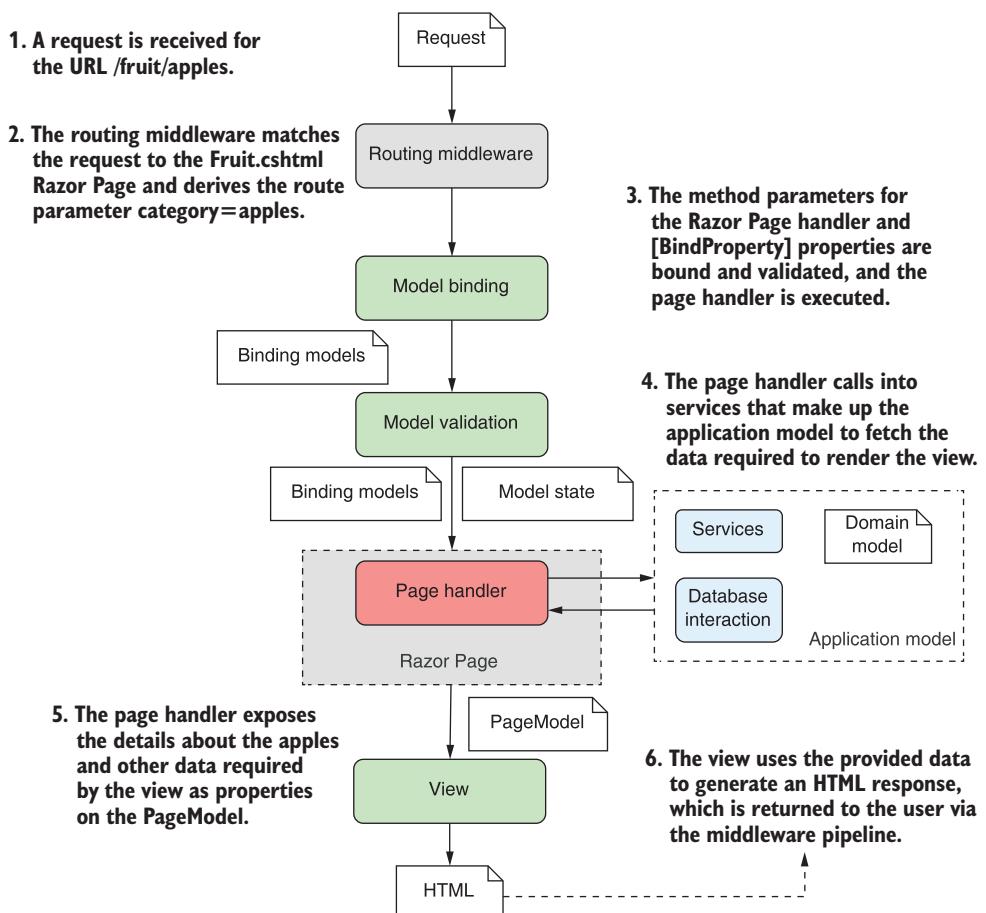


Figure 9.8 Handling a request to a traditional Razor Pages application, in which the view generates an HTML response that's sent back to the user. This diagram should be very familiar by now!

Razor Pages request after it passes through the middleware pipeline. This example shows how a request to view the available fruit on a traditional grocery store website might look.

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 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 9.9, 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.

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. Web APIs often *accept* data in more formats than Razor Pages, such as XML, but otherwise the model-binding process is the same as for the Razor Pages request. Validation also occurs in the same way, and the `ModelState` property on the  `ControllerBase` base class is populated with any validation errors.

**NOTE** Web APIs use *input formatters* to accept data sent to them in a variety of formats. Commonly these formats are JSON or XML, but you can create input formatters for any sort of type, such as CSV. I show how to enable the XML input formatter in section 9.6. You can see how to create a custom input formatter at <http://mng.bz/e5gG>.

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 practicable. 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 interacting with a user.

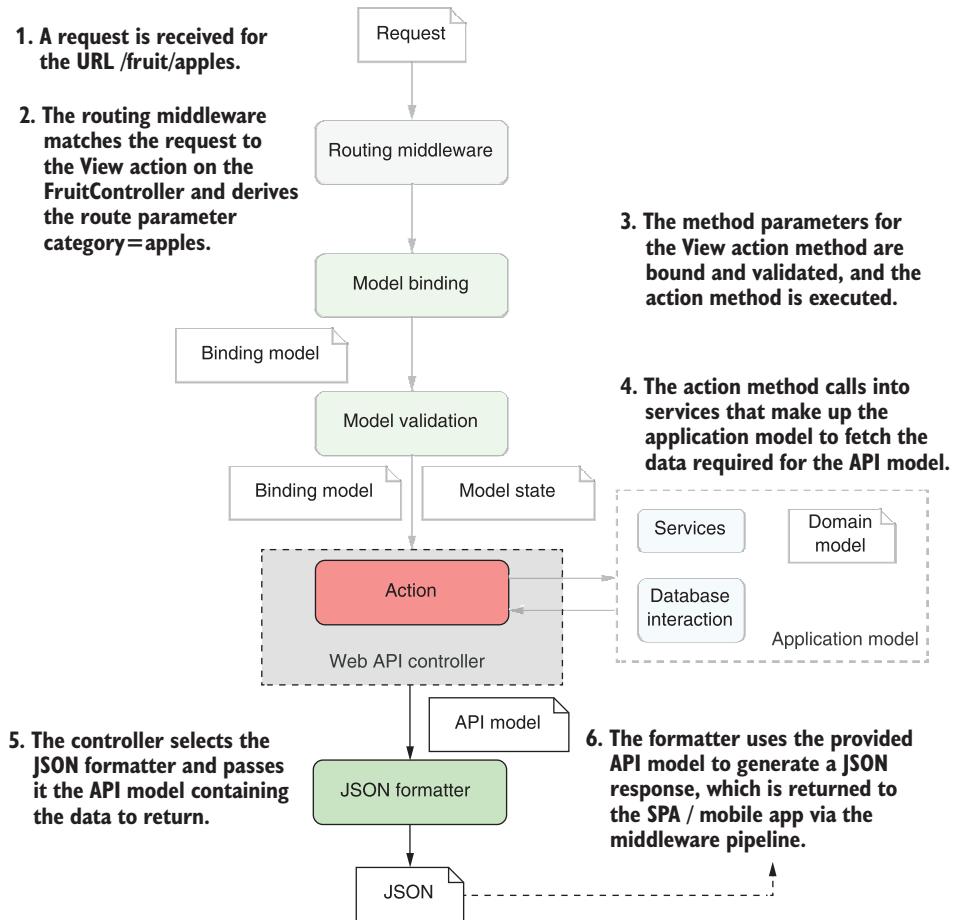


Figure 9.9 A call to a Web API endpoint in an e-commerce ASP.NET Core web application. The ghosted portion of the diagram is identical to figure 9.8.

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 the *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 an *output formatter*. An output formatter, as the name suggests, serializes the API model into a machine-readable response, such as JSON or XML. The output formatter forms the “V” in the Web API version of MVC, by choosing an appropriate representation of 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 the 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 have 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 @page directive, whereas with API controllers you use routing attributes.

**NOTE** 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("")]
    public IActionResult Index()
    {
        /* method implementation */
    }
}
```

The Index action will be executed when the / URL is requested.

```
[Route("contact")]
public IActionResult Contact()
{
    /* method implementation */
}
}
```

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")]
    [Route("car/ignition")]
    [Route("start-car")]
    public IActionResult Start()
    {
        /* method implementation */
    }

    [Route("car/speed/{speed}")]
    [Route("set-speed/{speed}")]
    public IActionResult SetCarSpeed(int speed)
    {
        /* method implementation */
    }
}
```

The Start method will be executed when any of these route templates are matched.

The name of the action method has no effect on the route template.

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 RouteAttributes 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 for other applications to consume.

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)
```

**NOTE** As discussed in chapter 5, don't use route constraints for validation. For example, if you call the preceding "`set-speed/{speed=20:int}`" route with an invalid value for speed, /`set-speed/oops`, you will get a `404 Not Found` response, as the route does not match. Without the `int` constraint, you would receive the more sensible `400 Bad Request` response.

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 with API controllers 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")]
    [Route("api/car/ignition")]
    [Route("/start-car")]
    public IActionResult Start()
    {
        /* method implementation*/
    }

    [Route("api/car/speed/{speed}")]
    [Route("/set-speed/{speed}")]
    public IActionResult SetCarSpeed(int speed)
    {
        /* method implementation*/
    }
}
```

A vertical bracket on the right side of the code spans from the first route attribute to the second. A callout box points to this bracket with the text "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")]
    [Route("ignition")]
    [Route("/start-car")]
    public IActionResult Start()
    {
        /* method implementation*/
    }

    [Route("speed/{speed}")]
    [Route("/set-speed/{speed}")]
    public IActionResult SetCarSpeed(int speed)
    {
        /* method implementation*/
    }
}
```

Annotations with arrows pointing to specific route attributes:

- An arrow points to the first route attribute in the controller definition with the text "Combines to give the 'api/car/start' template".
- An arrow points to the first route attribute in the `Start()` method definition with the text "Combines to give the 'api/car/ignition' template".
- An arrow points to the route attribute in the `Start()` method definition with the text "Does not combine because it starts with /; gives the 'start-car' template".
- An arrow points to the first route attribute in the `SetCarSpeed()` method definition with the text "Combines to give the 'api/car/speed/{speed}' template".
- An arrow points to the route attribute in the `SetCarSpeed()` method definition with the text "Does not combine because it 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. If you wish, 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]")]
public abstract class BaseController { }

public class CarController : BaseController
{
    [Route("[action]")]
    [Route("ignition")]
    [Route("/start-car")]
    public IActionResult Start()
    {
        /* method implementation */
    }
}
```

You can apply attributes to a base class, and derived classes will inherit them.

Token replacement happens last, so `[controller]` is replaced with "car" not "base".

Combines and replaces tokens to give the "api/car/start" template

Combines and replaces tokens to give the "api/car/ignition" template

Does not combine with base attributes because it starts with /, so it remains as "start-car"

**WARNING** If you use token replacement for `[controller]` or `[action]`, remember that renaming classes and methods will change your public API. If that worries you, you can stick to using static strings like "car" instead.

When combined with everything you learned in chapter 5, we've covered pretty much everything there is to know about attribute routing. 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, such as `GET` or `POST`, 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 of 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 following listing.

#### Listing 9.9 Using HTTP verb attributes with attribute routing

```
public class AppointmentController
{
    [HttpGet("/appointments")]
    public IActionResult ListAppointments()
    {
        /* method implementation */
    }

    [HttpPost("/appointments")]
    public IActionResult CreateAppointment()
    {
        /* method implementation */
    }
}
```

**Only executed in response to GET /appointments**

**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'll find yourself writing repeatedly. The `[ApiController]` attribute, introduced in ASP.NET Core 2.1, is designed to handle some of this for you and reduce the amount of boilerplate you need.

## 9.5 Using common conventions with the `[ApiController]` attribute

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**

```
Web APIs use attribute routing to define the route templates.
The [FromBody] attribute indicates that the parameter should be bound to the request body.

public class FruitController : ControllerBase
{
    List<string> _fruit = new List<string>
    {
        "Pear", "Lemon", "Peach"
    };

    [HttpPost("fruit")]
    public ActionResult Update([FromBody] UpdateModel model)
    {
        if (!ModelState.IsValid)
        {
            return BadRequest(
                new ValidationProblemDetails(ModelState));
        }

        if (model.Id < 0 || model.Id > _fruit.Count)
        {
            return NotFound(new ProblemDetails()
            {
                Status = 404,
                Title = "Not Found",
                Type = "https://tools.ietf.org/html/rfc7231"
                    + "#section-6.5.4",
            });
        }
    }
}

The list of strings serves as the application model in this example.

You need to check if model validation succeeded and return a 400 response if it failed.

If the data sent does not contain a valid ID, return a 404 ProblemDetails response.
```

```

    _fruit[model.Id] = model.Name;
    return Ok();
}

public class UpdateModel
{
    public int Id { get; set; }

    [Required]
    public string Name { get; set; }
}
}

```

**Update the model and return a 200 Response.**

**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 in the response body, 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 will 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]
public class FruitController : ControllerBase
{
    List<string> _fruit = new List<string>
    {
        "Pear", "Lemon", "Peach"
    };

    [HttpPost("fruit")]
    public ActionResult Update(UpdateModel model)
    {
        if (model.Id < 0 || model.Id > _fruit.Count)

```

**Adding the `[ApiController]` attribute applies several conventions common to API controllers.**

**The `[FromBody]` attribute is assumed for complex action method parameters.**

**The model validation is automatically checked, and if invalid, returns a 400 response.**

```

    {
        return NotFound();
    } ←
    _fruit[model.Id] = model.Name;

    return Ok();
}

public class UpdateModel
{
    public int Id { get; set; }

    [Required]
    public string Name { get; set; }
}
}

```

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 bold code in listing 9.10 can be removed and replaced with the `[ApiController]` attribute in listing 9.11. 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` format to return errors in a consistent format across all your controllers.<sup>3</sup> A typical `ProblemDetails` object looks something like the following, which shows an example `ValidationProblemDetails` object generated automatically for an invalid request:

---

<sup>3</sup> 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>.

```
{  
    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.

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 on my blog at <http://mng.bz/Gx7D>.

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, 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 following listing.<sup>4</sup>

<sup>4</sup> You can disable all the automatic features enabled by the [ApiController] attribute, but I'd encourage you to stick to the defaults unless you really need to change them. You can read more about disabling features in the documentation at <https://docs.microsoft.com/aspnet/core/web-api>.

**Listing 9.12 Customizing [ApiController] behaviors**

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddControllers()
            .ConfigureApiBehaviorOptions(options =>
        {
            options.SuppressModelStateInvalidFilter = true;           ←
        });
    }
    // ...
}
```

Control which conventions are applied by providing a configuration lambda.

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 topic 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")]
    public IEnumerable<string> ListCars()
    {
        return new string[]
        {
            "Nissan Micra", "Ford Focus"
        };
    }
}
```

The action is executed with a request to GET /api/cars.

The API model containing the data is an `IEnumerable<string>`.

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, or 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 type 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,<sup>a</sup> 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.

---

<sup>a</sup> For more on MIME types, see the Mozilla documentation: <http://mng.bz/gop8>.

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 listing 9.13) or via an `OkResult` or other `StatusCodesResult`, 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 9.10 shows that even though XML was requested, the API controller formatted the response as JSON.

**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.

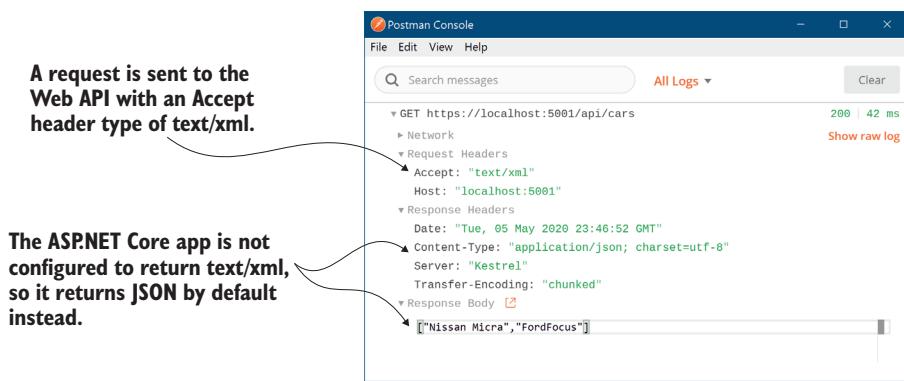


Figure 9.10 Even though the request was made with an `Accept` header of `text/xml`, the response returned was JSON, as the server was not configured to return XML.

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 it 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!

That all changed with the introduction of a new library in ASP.NET Core 3.0, `System.Text.Json`, which focuses on performance. In ASP.NET Core 3.0 onwards, ASP.NET Core uses `System.Text.Json` by default instead of `Newtonsoft.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:

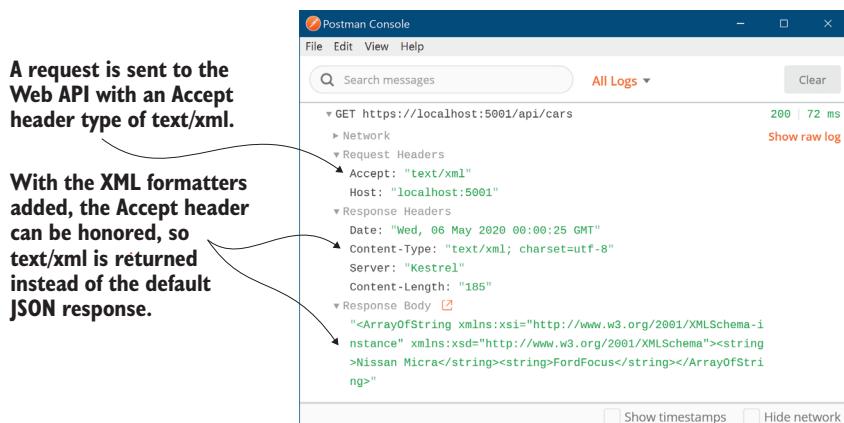
```
services.AddControllers()
    .AddNewtonsoftJson();
```

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 Microsoft's article "How to migrate from Newtonsoft.Json to System.Text.Json": <http://mng.bz/OmRJ>. For more advice on when to switch to the Newtonsoft.Json formatter, see the section "Add Newtonsoft.Json-based JSON format support" in Microsoft's "Format response data in ASP.NET Core Web API" documentation: <http://mng.bz/zx11>.

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,<sup>5</sup> use the following:

```
services.AddControllers()
    .AddXmlSerializerFormatters();
```

With this simple change, your API controllers can now format responses as XML. Running the same request as shown in figure 9.10 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 9.11.



**Figure 9.11** With the XML output formatters added, the text/xml Accept header is respected and the response can be serialized to XML.

<sup>5</sup> 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 <http://mng.bz/e5gG>.



```
{  
    options.RespectBrowserAcceptHeader = true;  
    options.OutputFormatters.RemoveType<StringOutputFormatter>(); <--  
}  
}  
  
False by default, a number of other  
properties are also available to be set.
```

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 Microsoft’s “Format response data in ASP.NET Core Web API” documentation: <http://mng.bz/zx11>.

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: you’ll learn the details needed to build complete apps, like adding users to your application, saving data to a database, and deploying 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.

## 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 that 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 `PageModels`, which contain both data and metadata about how to generate the response.

- Web APIs are associated with route templates by applying `RouteAttributes` 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, such as where you're using a common prefix on your routes.
- 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 you to choose 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, will automatically generate 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 onwards uses `System.Text.Json`, which is a strict, high performance library for JSON serialization and deserialization. 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'll 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'll 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'll 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'll only provide a brief introduction and point you to John Smith's excellent book Entity Framework Core in Action, second edition (Manning, 2021). I'll show you how to create a database and how to insert, update, and query simple data.

In chapters 13 through 15, we'll 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 apps 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'll 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.

# *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 look at 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.<sup>1</sup>

**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<sup>2</sup> principles have held up well.

On that basis, ASP.NET Core has *dependency injection* (sometimes called *dependency inversion*, *DI*, or *inversion of control*<sup>3</sup>) 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.

---

<sup>1</sup> Martin Fowler's website at <https://martinfowler.com> is a goldmine of best-practice goodness. One of the most applicable articles to this chapter can be found at [www.martinfowler.com/articles/injection.html](http://www.martinfowler.com/articles/injection.html).

<sup>2</sup> SOLID is a mnemonic for single responsibility, open-closed, Liskov substitution, interface segregation, and dependency inversion: [https://en.wikipedia.org/wiki/SOLID\\_\(object-oriented\\_design\)](https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)).

<sup>3</sup> 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 titled “Dependency Injection Is NOT the Same as the Dependency Inversion Principle”: <http://mng.bz/5jvB>.

### 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 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)
    {
        var emailSender = new EmailSender();
        emailSender.SendEmail(username);
        return Ok();
    }
}
```

The action method is called when a new user is created.

Creates a new instance of EmailSender

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 here for the real process of sending the email.

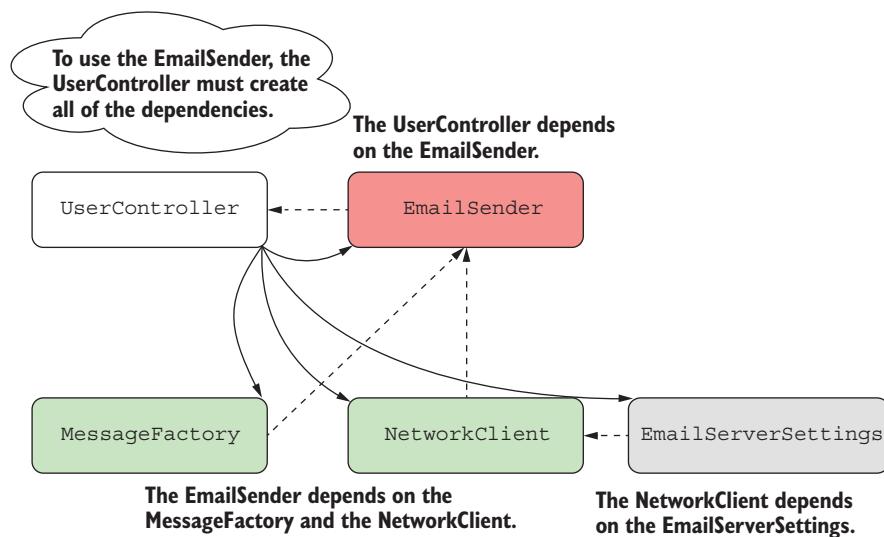
**NOTE** Although I'm using sending email as a simple example, in practice you might want to move this code out of your Razor Page and controller classes entirely. This type of asynchronous task is well suited to using message queues and a background process. For more details, see <http://mng.bz/pVWR>.

If the `EmailSender` class is as simple as the previous example and it has no dependencies, 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.



**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

Instances of the dependencies are provided in the constructor.

```
public class EmailSender
{
    private readonly NetworkClient _client;
    private readonly MessageFactory _factory;
    public EmailSender(MessageFactory factory, NetworkClient client)
    {
        _factory = factory;
        _client = client;
    }
    public void SendEmail(string username)
    {
        var email = _factory.Create(username);
        _client.SendEmail(email);
        Console.WriteLine($"Email sent to {username}!");
    }
}
```

The EmailSender now depends on two other classes.

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

You need a new Message-Factory.

The Network-Client also has dependencies.

```
public IActionResult RegisterUser(string username)
{
    var emailSender = new EmailSender(
        new MessageFactory(),
        new NetworkClient(
            new EmailServerSettings
            (
                host: "smtp.server.com",
                port: 25
            )));
}
```

To create EmailSender, you must create all of its dependencies.

You're already two layers deep, but there could feasibly be more.

```

    );
emailSender.SendEmail(username);
return Ok();
}

```

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:

- *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.

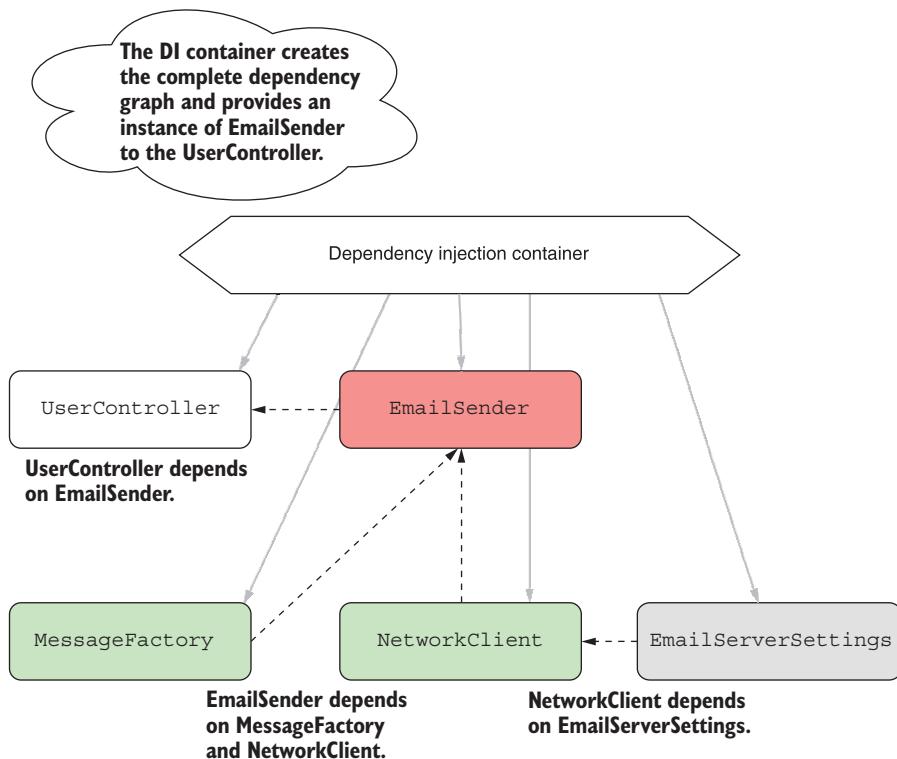
**TIP** Generally speaking, any dependencies in your code should be explicit, not implicit. Implicit dependencies are hard to reason about and difficult to test, so you should avoid them wherever you can. DI is useful for guiding you along this path.

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 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



**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.

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;
    public UserController(EmailSender emailSender)
    {
        _emailSender = emailSender;
    }

    [HttpPost("register")]
    public IActionResult RegisterUser(string username)
    {
        _emailSender.SendEmail(username);
        return Ok();
    }
}
```

Instead of creating the dependencies implicitly, they're injected via the constructor.

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

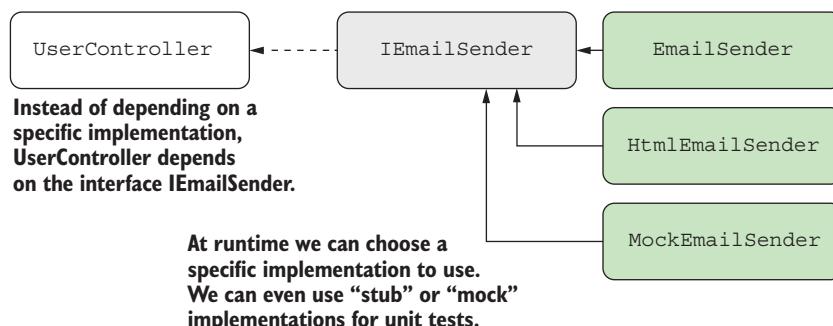
*Coupling* is an important concept in object-oriented programming. It refers to how a given class depends on other classes to perform its function. Loosely coupled code doesn't need to know a lot of details about a particular component to use it.

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 would be 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.



**Figure 10.3** By coding to interfaces instead of an explicit implementation, you can use different IEmailSender implementations in different scenarios, such as 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 in the following listing. That would allow you to use a different implementation during unit tests, such as a DummyEmailSender.

### Listing 10.5 Using interfaces with dependency injection

```
public class UserController : ControllerBase
{
    private readonly IEmailSender _emailSender;
    public UserController(IEmailSender emailSender)
    {
        _emailSender = emailSender;
    }

    [HttpPost("register")]
    public IActionResult RegisterUser(string username)
    {
        _emailSender.SendEmail(username);
        return Ok();
    }
}
```

You now depend on `IEmailSender` instead of the specific `EmailSender` implementation.

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.

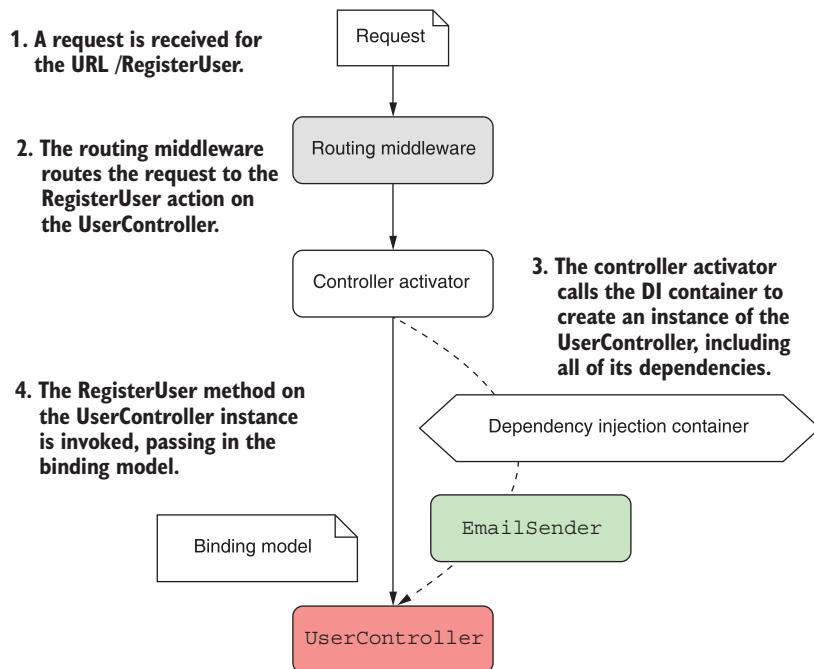


Figure 10.4 The ASP.NET Core hosting model uses the DI container to fulfill dependencies when creating controllers.

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.<sup>4</sup>

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

<sup>4</sup> You can read about the Service Locator antipattern in *Dependency Injection Principles, Practices, and Patterns* by Steven van Deursen and Mark Seemann (Manning, 2019): <http://mng.bz/6g4o>.

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`.

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();           ←
}
```

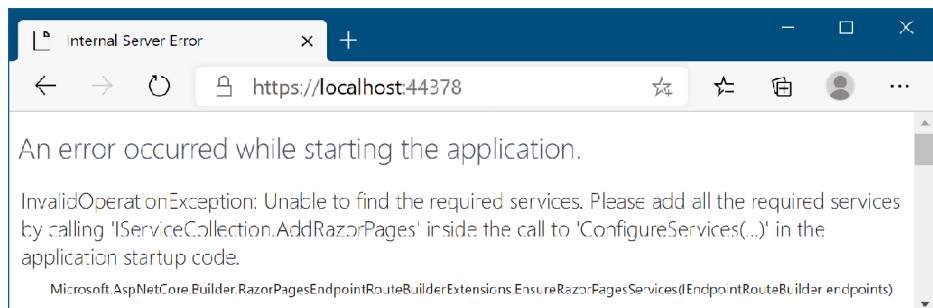
The `AddRazorPages` extension  
method adds all necessary services  
to the `IServiceCollection`.

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`.

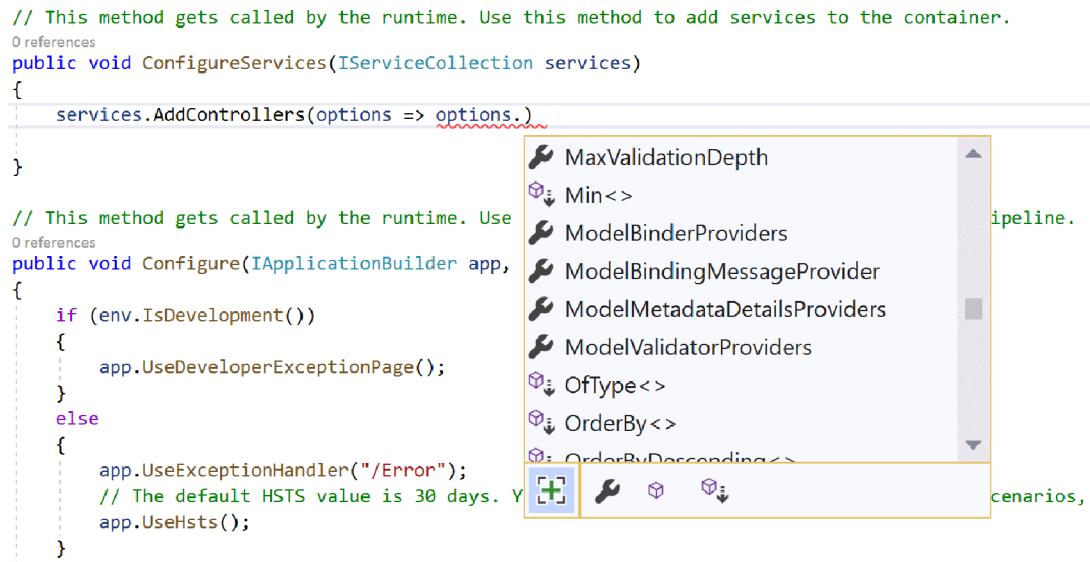
There's no way of knowing exactly which libraries will require you to add services to the container; it's generally a case of checking the documentation for any libraries you use. If you forget to add them, you may find the functionality doesn't work, or you might get a handy exception like the one shown in figure 10.5. Keep an eye out for these and be sure to register any services that you need.



**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.

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.



**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.

### 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, 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`).

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.

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 `HttpClient`, use an instance of `HttpClient`.
- When a service requires `MessageFactory`, use an instance of `MessageFactory`.

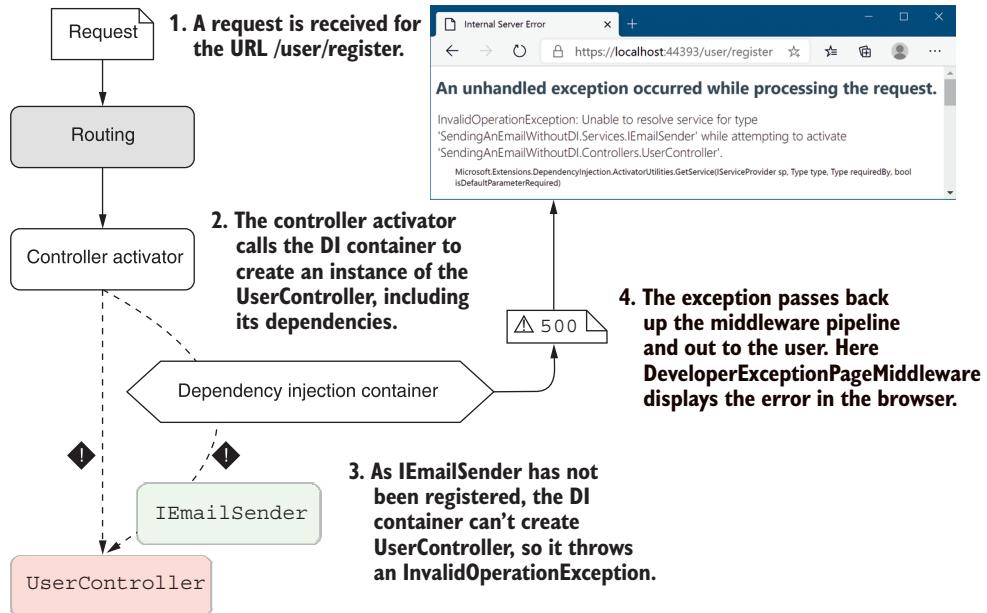


Figure 10.7 If you don't register all your required dependencies in `ConfigureServices`, you'll get an exception at runtime, telling you which service wasn't registered.

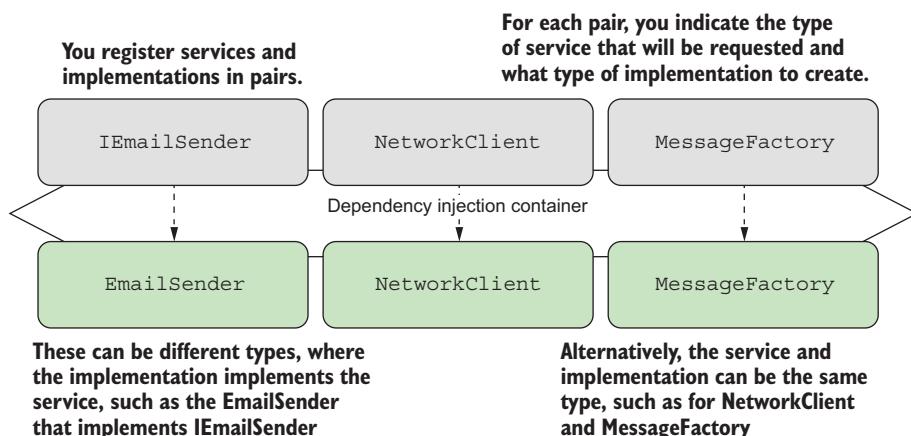


Figure 10.8 Configuring the DI container in your application involves telling it what type to use when a given service is requested; for example, “Use `EmailSender` when `IEmailSender` is required.”

**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. It's 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. It must be a concrete type, such as `EmailSender`. It 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. 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();           ← You're using API controllers, so
    services.AddScoped<IEmailSender, EmailSender>();   ← you must call AddControllers.
    services.AddScoped<NetworkClient>();           ← Whenever you require
    services.AddSingleton<MessageFactory>();          ← an IEmailSender, use
}                                         ← Whenever you require a
                                         ← Whenever you require a
                                         ← NetworkClient, use NetworkClient.
                                         ← MessageFactory, use MessageFactory.
```

That's all there is to dependency injection! It may seem a little bit like magic,<sup>5</sup> 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, and hence the slightly simpler signature.

<sup>5</sup> 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.

**NOTE** The `EmailSender` instance is *only* registered as an `IEmailSender`, so you can't retrieve it by requesting the specific `EmailSender` implementation; you *must* use the `IEmailSender` interface.

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.
- 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 they must be arguments 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, it 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 strings 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 preconstructed 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
        (
            host: "smtp.server.com",
            port: 25
        ));
}
```

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*. If you create an object and pass it to the container, it's *always* registered as a singleton. You can also register any class using the `AddSingleton<T>()` method, and the container will only use one instance throughout your application. I discuss singletons along with other *lifetimes* in detail in section 10.3. 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.

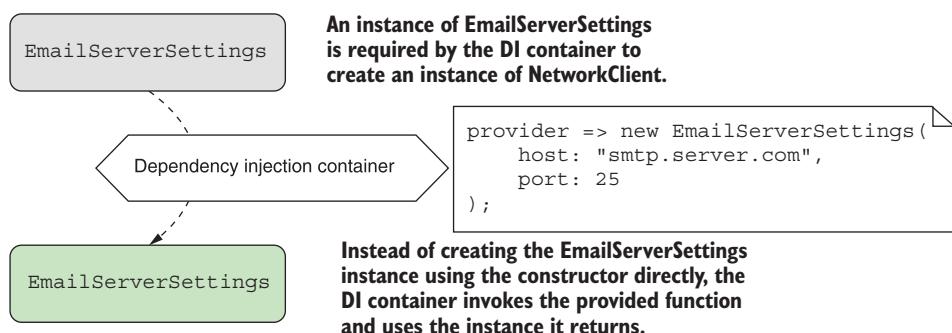


Figure 10.9 You can register a function with the DI container that will be invoked whenever a new instance of a service is required.

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(
        provider =>
        new EmailServerSettings
        (
            host: "smtp.server.com",
            port: 25
        ));
}
```

**The lambda is provided an instance of IService-Provider.**

**Because you're providing a function to create the object, you aren't restricted to a singleton.**

**The constructor is called every time an EmailServerSettings object 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.

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.

**(continued)**

To register these types, you must use a different overload of the Add\* methods. For example,

```
services.AddScoped(typeof(IRepository<T>), typeof(DbRepository<T>));
```

This ensures that whenever a service constructor requires `IRepository<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) ← Create an extension method on IServiceCollection by using the "this" keyword.

    {
        services.AddScoped<IEmailSender, EmailSender>();
        services.AddSingleton<NetworkClient>();
        services.AddScoped<MessageFactory>();
        services.AddSingleton(
            new EmailServerSettings
            (
                host: "smtp.server.com",
                port: 25
            ));
        return services; ← By convention, return the IServiceCollection to allow method chaining.
    }
}
```

← Cut and paste your registration code from `ConfigureServices`.

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

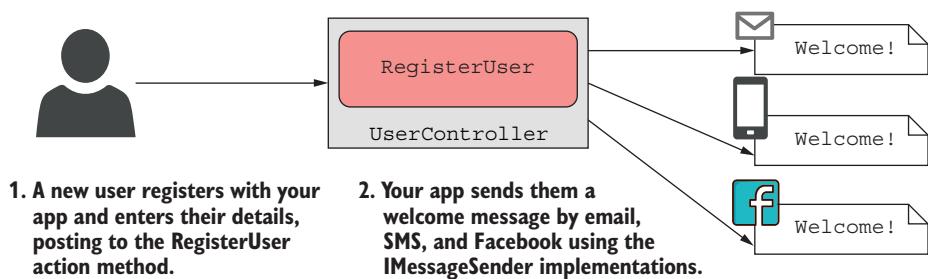
One of the advantages of coding to interfaces is that you can create multiple implementations of a service. For example, imagine you want to create a more generalized version of `IEmailSender` so that you can send messages via SMS or Facebook, as well as by email. You create an interface for it,

```
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 on whether you want to use all the implementations in your consumer or only one of them.

##### INJECTING MULTIPLE IMPLEMENTATIONS OF AN INTERFACE

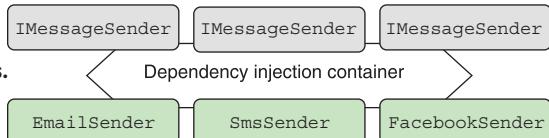
Imagine you want to send a message using each of the `IMessageSender` implementations whenever a new user registers, so that they get an email, an SMS, and a Facebook message, as shown in figure 10.10.



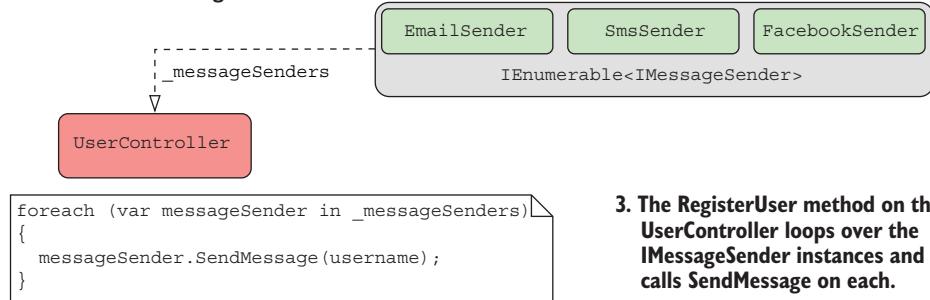
**Figure 10.10** When a user registers with your application, they call the `RegisterUser` method. This sends them an email, an SMS, and a Facebook message using the `IMessageSender` classes.

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.**



- 2. The DI container creates one of each `IMessageSender` implementation and injects them into the `UserController` as an `IEnumerable<IMessageSender>`.**



- 3. The `RegisterUser` method on the `UserController` loops over the `IMessageSender` instances and calls `SendMessage` on each.**

**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
{
}
  
```

**Requesting an `IEnumerable` will inject an array of `IMessageSender`.**

```
private readonly IEnumerable<IMessageSender> _messageSenders;
public UserController(
    IEnumerable<IMessageSender> messageSenders)
{
    _messageSenders = messageSenders;
}

[HttpPost("register")]
public IActionResult RegisterUser(string username)
{
    foreach (var messageSender in _messageSenders)
    {
        messageSender.SendMessage(username);
    }

    return Ok();
}
```

**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 how you can conditionally add `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>();      ←
    services.TryAddScoped<IMessageSender, SmsSender>();       ←
}
```

**EmailSender** is  
registered with  
the container.

**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 `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:

- Action methods
- Page handler methods
- 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 that 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;
    private readonly IPromotionService _promoService;
    public UserController(
        IMessageSender messageSender, IPromotionService promoService)
    {
        _messageSender = messageSender;
        _promoService = promoService;
    }

    [HttpPost("register")]
    public IActionResult RegisterUser(string username)
```

Both `IMessageSender` and `IPromotionService` are injected into the constructor every time.

```

    {
        _messageSender.SendMessage(username);
        return Ok();
    }

    [HttpPost("promote")]
    public IActionResult PromoteUser(string username, int level)
    {
        _promoService.PromoteUser(username, level);
        return Ok();
    }
}

```

The RegisterUser method only uses **IMessageSender**.

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(
        [FromServices] IMessageSender messageSender,
        string username)
    {
        messageSender.SendMessage(username);
        return Ok();
    }

    [HttpPost("promote")]
    public IActionResult PromoteUser(
        [FromServices] IPromotionService promoService,
        string username, int level)
    {
        promoService.PromoteUser(username, level);
        return Ok();
    }
}

```

The `[FromServices]` attribute ensures **IMessageSender** is resolved from the DI container.

**IMessageSender** is only available in **RegisterUser**.

**IPromotionService** is resolved from the DI container and injected as a parameter.

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 listing 10.15.

**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()
    {
    }

    public IActionResult OnPost(
        [FromServices] IPromotionService promoService,
        string username, int level)
    {
        promoService.PromoteUser(username, level);
        return RedirectToPage("success");
    }
}
```

The `OnGet` handler does not require any services.

`IPromotionService` is resolved from the DI container and injected as a parameter.

Only the `OnPost` page handler can use `IPromotionService`.

Generally speaking, if you find you need to use the `[FromServices]` attribute, 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
<h1>The page title</h1>
<footer>
    @htmlHelper.Copyright()
</footer>
```

Injects an instance of `HtmlGenerator` into the view, named `htmlHelper`

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, such as 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'll 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, I'll use a simple representative example in this section. 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);
    }
    public int RowCount { get; }
}
```

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.

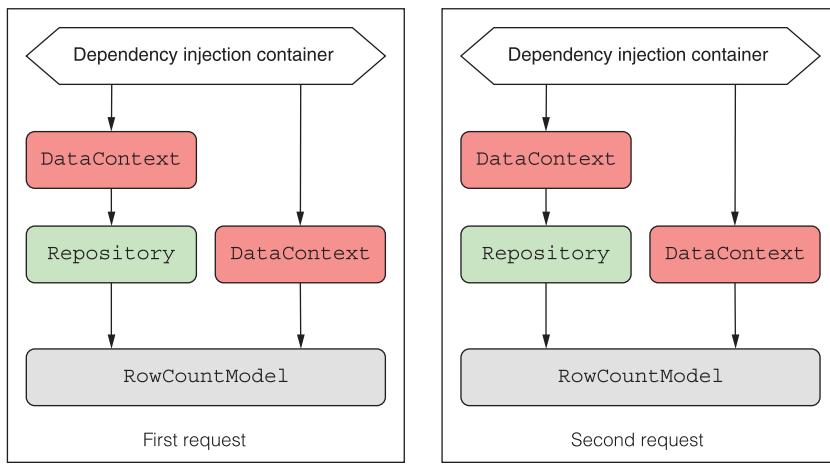
**Listing 10.18 Repository service that depends on an instance of DataContext**

```
public class Repository
{
    private readonly DataContext _dataContext;
    public Repository(DataContext dataContext)
    {
        _dataContext = dataContext;
    }
    public int RowCount => _dataContext.RowCount;
}
```

An instance of  
DataContext is  
provided using DI.

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.



**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`.

`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;
    private readonly DataContext _dataContext;
    public RowCountPageModel(
        Repository repository,
        DataContext dataContext)
    {
        _repository = repository;
        _dataContext = dataContext;
    }

    public void OnGet()
    {
        DataContextCount = _dataContext.RowCount;
        RepositoryCount = _repository.RowCount;
    }

    public int DataContextCount { get; set; }
    public int RepositoryCount { get; set; }
}
```

**DataContext and Repository are passed in using DI.**

**When invoked, the page handler retrieves and records RowCount from both dependencies.**

**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.

### 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 from 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.

LifetimeExamples	LifetimeExamples
<p><b>Random row count values</b></p> <p>The <code>DatabaseContext.RowCount</code> is 474,790,543</p> <p>The <code>Repository.RowCount</code> is 720,174,606</p>	<p><b>Random row count values</b></p> <p>The <code>DatabaseContext.RowCount</code> is 939,247,122</p> <p>The <code>Repository.RowCount</code> is 335,510,854</p>

**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, this 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.

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>();
```

LifetimeExamples		LifetimeExamples	
<p><b>Random row count values</b></p> <p>The <code>DbContext.RowCount</code> is 324,949,579</p> <p>The <code>Repository.RowCount</code> is 324,949,579</p>		<p><b>Random row count values</b></p> <p>The <code>DbContext.RowCount</code> is 661,002,787</p> <p>The <code>Repository.RowCount</code> is 661,002,787</p>	

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 RowCounts are identical within a request.

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 expensive to create, that 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`:

```
services.AddSingleton<DataContext>();
```

We can then call the RowCountModel Razor Page twice and observe the results 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.

LifetimeExamples	LifetimeExamples
<p><b>Random row count values</b></p> <p>The <code>DataContext.RowCount</code> is 765,012,313</p> <p>The <code>Repository.RowCount</code> is 765,012,313</p>	<p><b>Random row count values</b></p> <p>The <code>DataContext.RowCount</code> is 765,012,313</p> <p>The <code>Repository.RowCount</code> is 765,012,313</p>

**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 load the settings once at startup and reuse 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.

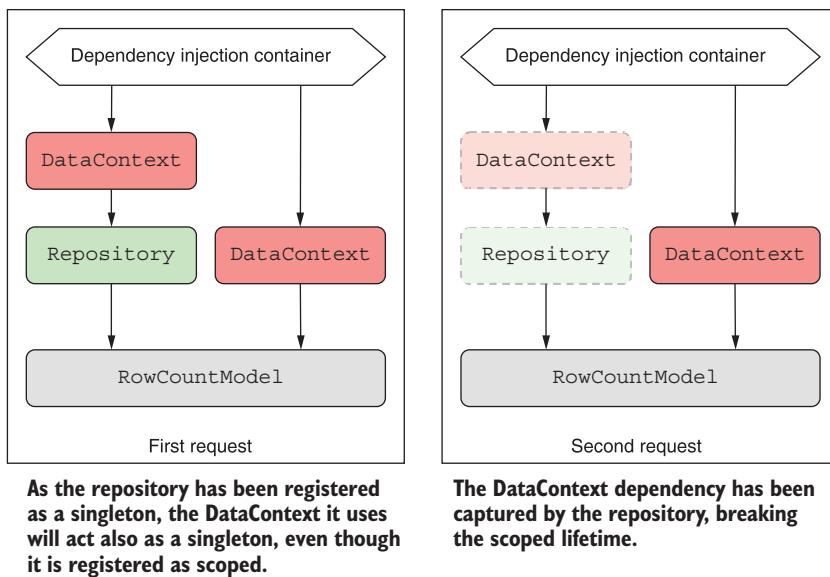
#### 10.3.4 Keeping an eye out for captured dependencies

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



**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.

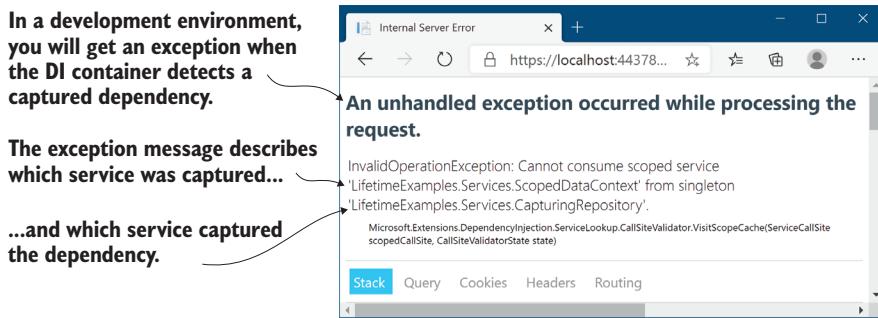
created for each scope, but the instance inside Repository hangs around for the lifetime of the app.

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.

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 listing 10.20.



**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.

#### 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)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            })
            .UseDefaultServiceProvider(options =>
            {
                options.ValidateScopes = true;
                options.ValidateOnBuild = true;
            });
    }
}
```

You can override the validation check with the `UseDefaultServiceProvider` extension.

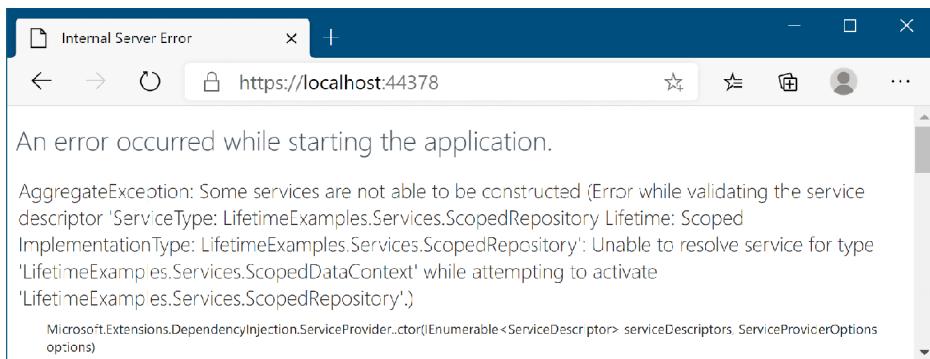
The default builder sets `ValidateScopes` to validate only in development environments.

Setting this to true will validate scopes in all environments. This has performance implications.

**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 it's 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<sup>6</sup>.

<sup>6</sup> Unfortunately, the container can't catch everything. For a list of caveats and exceptions, see this post from my blog: <http://mng.bz/QmwG>.



**Figure 10.18 When `ValidateOnBuild` is enabled, the DI container will check on app startup 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 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!

## 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` or 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 fulfill 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 that 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 fulfill 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 11.3 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 11.4. 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'll 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.

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

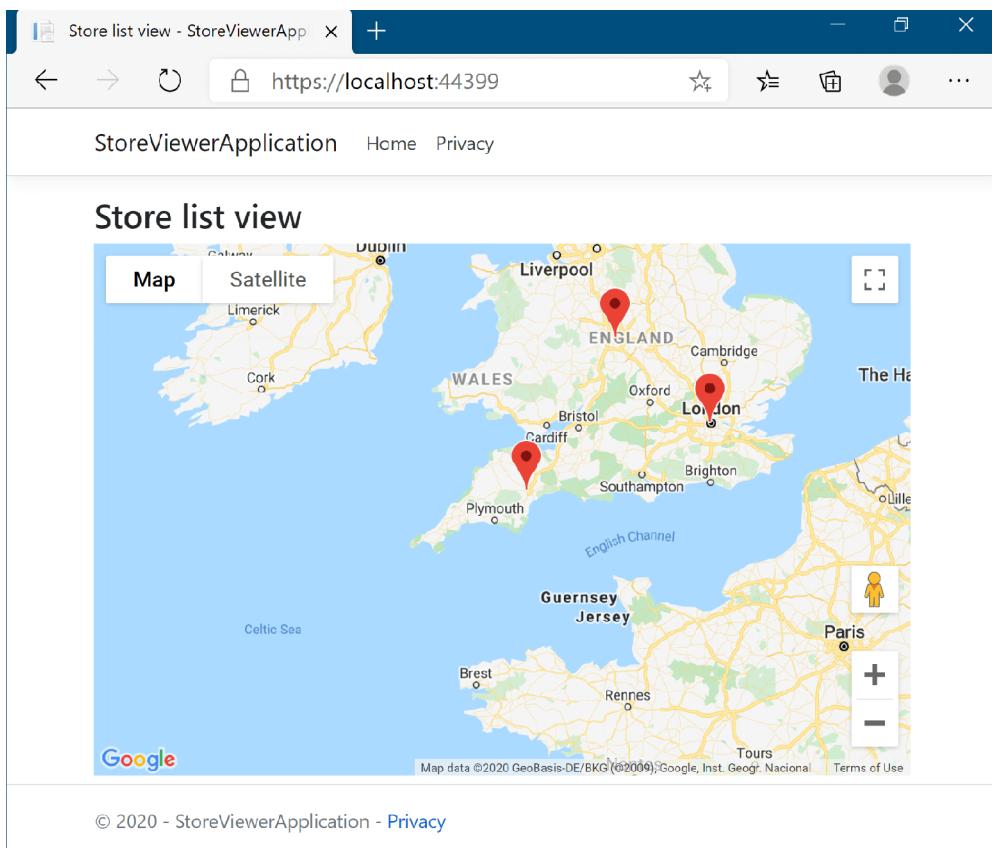


Figure 11.1 You can store the default map location, zoom level, and mapping API Key in configuration and load them at runtime. It's important to keep secrets like API keys in configuration and out of your code.

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 5.0 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 5.0 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();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

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 `Host.CreateDefaultBuilder` method**

```
public static IHostBuilder CreateDefaultBuilder(string[] args)
{
    var builder = new HostBuilder()
        .UseContentRoot(Directory.GetCurrentDirectory())
```

```

    .ConfigureHostConfiguration(config =>
    {
        // Configuration provider setup
    })
Configures application settings, the topic of this chapter

    .ConfigureAppConfiguration((hostingContext, config) =>
    {
        // Configuration provider setup
    })
    .ConfigureLogging((hostingContext, logging) =>
    {
        logging.AddConfiguration(
            hostingContext.Configuration.GetSection("Logging"));
        logging.AddConsole();
        logging.AddDebug();
    })
    .UseDefaultServiceProvider((context, options) =>
    {
        var isDevelopment = context.HostingEnvironment
            .IsDevelopment();
        options.ValidateScopes = isDevelopment;
        options.ValidateOnBuild = isDevelopment;
    });
}

Configures the DI container, optionally enabling verification settings

    return builder;      ← Returns HostBuilder for further configuration by calling extra methods before calling Build()
}

```

**Configures hosting settings such as determining the hosting environment**

**Sets up the logging infrastructure**

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, which 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'll 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.

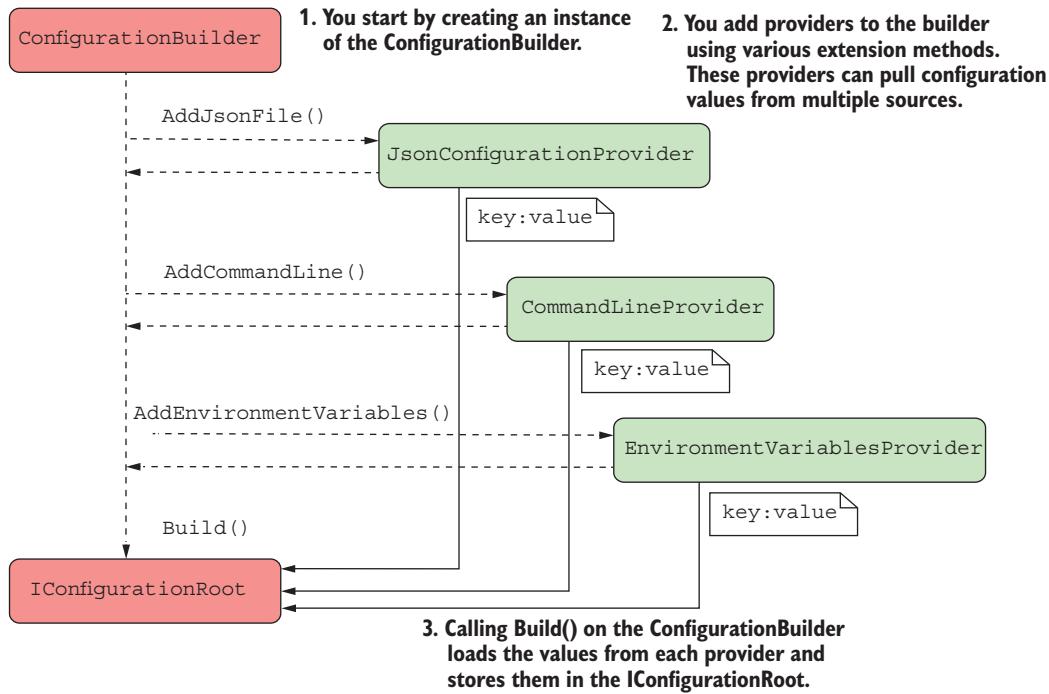
You describe your configuration setup by adding a number of `IConfigurationProviders` to the `ConfigurationBuilder` in `ConfigureAppConfiguration`. These describe how to load the key-value pairs from a particular source; for example, a JSON file or environment variables, as shown in figure 11.2. Calling `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.

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



**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`.

example, you could use the official Azure Key Vault provider NuGet package<sup>1</sup> or the YAML file provider I wrote.<sup>2</sup>

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. The following listing shows the default file generated by the ASP.NET Core 5.0 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"
    }
  }
}
```

<sup>1</sup> The Azure Key Vault provider is available on NuGet: <http://mng.bz/OEKj>.

<sup>2</sup> You can find my YAML provider on GitHub at <http://mng.bz/Yqdj>.

```

},
"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": {
    "DefaultZoomLevel": 9,           ← Values can be numbers in the JSON file, but they'll
    "DefaultLocation": {            ← be converted to strings when they're read.
      "latitude": 50.500,
      "longitude": -4.000
    }
  }
}

```

I've nested the new configuration 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, we'll 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`. It also 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. These are 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 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 following listing, 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 `HostBuilder`

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureAppConfiguration(addAppConfiguration)
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });

    public static void AddAppConfiguration(
        HostBuilderContext hostingContext,
        IConfigurationBuilder config)
    {
        // Adds the configuration setup function to HostBuilder
    }
}
```

**Adds the configuration setup function to HostBuilder**

**HostBuilder provides a hosting context and an instance of ConfigurationBuilder.**

```

{
    config.Sources.Clear();
    config.AddJsonFile("appsettings.json", optional: true);
}

```

**Clears the providers configured by default in CreateDefaultBuilder**

**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 a `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? `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 in which 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`.

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 listing 11.6 for brevity.

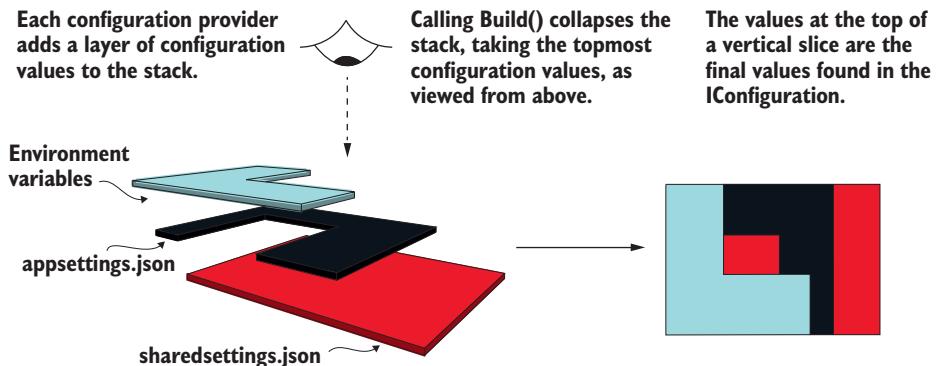


Figure 11.3 Each configuration provider adds a “layer” of values to `ConfigurationBuilder`. Calling `Build()` collapses that configuration. Later providers will overwrite configuration values with the same keys as earlier providers.

#### 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)
            .AddJsonFile("appsettings.json", optional: true)
            .AddEnvironmentVariables(); ← Adds the machine's
                                         environment variables as
                                         a configuration provider
    }
}
```

Loads configuration from a different JSON configuration file before the `appsettings.json` file

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.

Most of the settings in each provider are unique and are 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

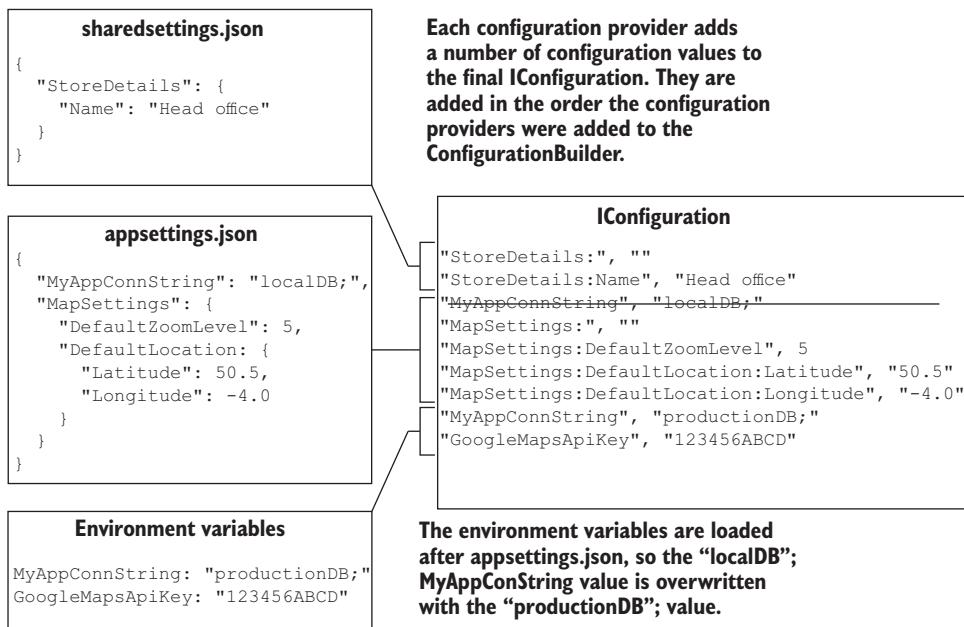


Figure 11.4 The final `IConfiguration` includes the values from each of the providers. Both `appsettings.json` and the environment variables include the `MyAppConnString` key. As the environment variables are added later, that configuration value is used.

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, a 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 because they're stored in plaintext. They're intended to help you avoid committing secrets to source control.

**TIP** Azure Key Vault<sup>3</sup> 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. Another popular option is Vault by Hashicorp ([www.vaultproject.io/](http://www.vaultproject.io/)), which can be run on your premises or in the cloud.

Whichever approach you use 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.

**NOTE** 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 the colon in environment variables. You must use the double underscore approach in these environments instead. The double underscore in environment variables will be converted into a colon when they're imported into the `IConfiguration` object. You should always use the colon 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 provider will read them at runtime, overriding the defaults specified in your `appsettings.json` files.<sup>4</sup>

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 `ConnectionString_DefaultConnection` environment variable, that would be added for *every* app you run locally. That sounds like more of a hassle than a benefit!

---

<sup>3</sup> 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 Microsoft's "Azure Key Vault Configuration Provider in ASP.NET Core" documentation at <http://mng.bz/BR7v>.

<sup>4</sup> For instructions on how to set environment variables for your operating system, see Microsoft's "Use multiple environments in ASP.NET Core" documentation at <http://mng.bz/d4OD>.

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.

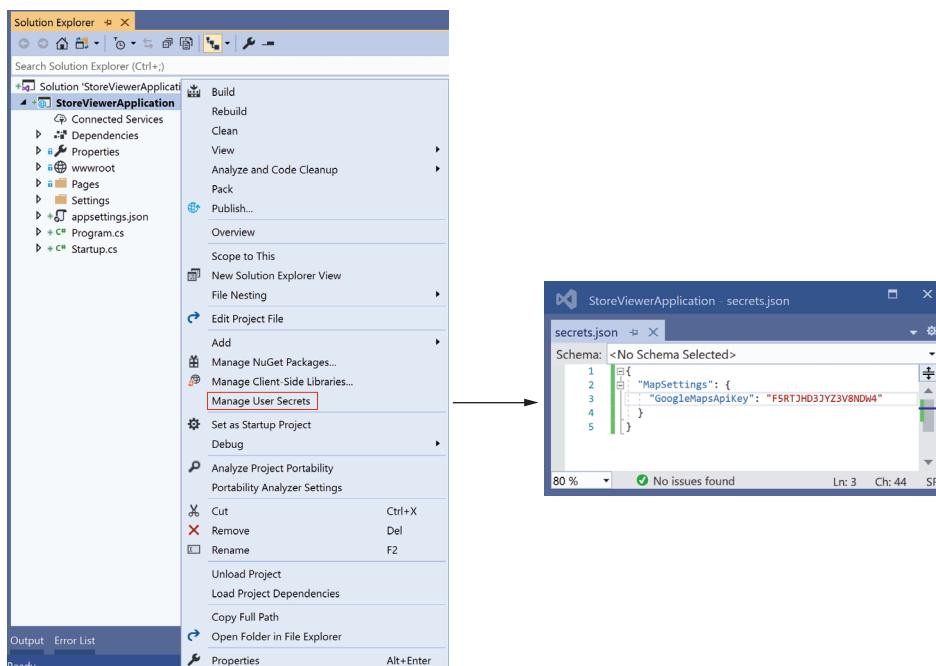
### STORING SECRETS WITH THE USER SECRETS MANAGER IN DEVELOPMENT

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 for each app to keep the secrets segregated.

**WARNING** The secrets aren't encrypted, so they 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 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.



**Figure 11.5** Select **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 they 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 for details.<sup>5</sup>

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 preceding snippet 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 UserSecretsId-Attribute. 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.

---

<sup>5</sup> The Secret Manager Tool stores the secrets.json file in the user profile. You can read more about this tool specifically in Microsoft's "Safe storage of app secrets in development in ASP.NET Core" documentation: <http://mng.bz/ryAg>. You can find more about .NET tools in general in Microsoft's "How to manage .NET tools" documentation: <http://mng.bz/VdmX>.

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, 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. 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:

- Error
- Warning
- Information
- Debug

Each of these settings is more verbose than the last, but it 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;           ←
        );
    }
}
```

**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 that 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": {
        "DefaultZoomLevel": 6,
        "DefaultLocation": {
            "latitude": 50.500,
            "longitude": -4.000
        }
    }
}
```

**Settings related to the store locator section of the app**

```

        }
    },
    "AppDisplaySettings": {
        "Title": "Acme Store Locator",
        "ShowCopyright": true
    }
}

```

**General settings related to displaying the app**

The simplest approach to making the home page settings available in the Index.cshtml Razor Page would be to inject `IConfiguration` into the `PageModel` and access the values using the dictionary syntax:

```

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)
    {
        AppDisplaySettings settings = options.Value;
        var title = settings.Title;
        bool showCopyright = settings.ShowCopyright;
    }
}

```

**You can inject a strongly typed options class using the `IOptions<T>` wrapper interface.**

**The Value property exposes the POCO settings object.**

**The settings object contains properties that are bound to configuration values at runtime.**

**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 the injection of configuration classes as explicit dependencies. Such settings 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're not restricted 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.

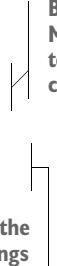
To help facilitate the binding of configuration values to your custom POCO options classes, ASP.NET Core introduces the `IOptions<T>` interface. This is a simple interface with a single property, `Value`, which contains your configured POCO options class at runtime. Options classes are set up in the `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>(
        Configuration.GetSection("MapSettings"));

    services.Configure<AppDisplaySettings>(
        Configuration.GetSection("AppDisplaySettings"));
}
```

Binds the `MapSettings` section to the POCO options class `MapSettings`



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, such as by calling `GetSection(nameof(MapSettings))`.

Each call to `Configure<T>` sets up the following series of actions internally:

- 1 Creates an instance of `ConfigureOptions<T>`, which indicates that `IOptions<T>` should be configured based on configuration.  
If `Configure<T>` is called multiple times, multiple `ConfigureOptions<T>` objects will be used, all of which can be applied to create the final object, in much the same way as `IConfiguration` is built from multiple layers.
- 2 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.
- 3 The `IOptions<T>` interface is registered 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, such as `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, `IOptionsSnapshot<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 from 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.

`IOptionsSnapshot<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)
    {
        AppDisplaySettings settings = options.Value;
        var title = settings.Title;
    }
}
```

`IOptionsSnapshot<T> will update if the underlying configuration values change.`

`The Value property exposes the POCO settings object, the same as for IOptions<T>.`

`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.

An important consideration when using the options pattern 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 already touched on some of the requirements for POCO classes to work with the `IOptions<T>` binder, 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 that

- 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

The following 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; }
    public int Integer { get; set; }
    public SubClass Object { get; set; }
    public SubClass ReadOnly { get; } = new SubClass();
    public Dictionary<string, SubClass> Dictionary { get; set; }
    public List<SubClass> List { get; set; }
    public IDictionary<string, SubClass> IDictionary { get; set; }
    public IEnumerable<SubClass> IEnumerable { get; set; }
    public ICollection<SubClass> IConvertible { get; }
        = new List<SubClass>();

    internal string NotPublic { get; set; }
    public SubClass SetOnly { set => _setOnly = value; }
    public SubClass NullReadOnly { get; } = null;
    public SubClass NullPrivateSetter { get; private set; } = null;
    public SubClass this[int i]
    {
        get => _indexerList[i];
        set => _indexerList[i] = value;
    }

    public List<SubClass> NullList { get; }
    public Dictionary<int, SubClass> IntegerKeys { get; set; }
    public IEnumerable<SubClass> ReadOnlyEnumerable { get; }
        = new List<SubClass>();
    public SubClass _setOnly = null;
    private readonly List<SubClass> _indexerList
        = new List<SubClass>();
    public class SubClass
    {
        public string Value { get; set; }
    }
}
```

**The binder will also bind collections, including interfaces; dictionaries must have string keys.**

**The binder can't bind nonpublic, set-only, null-read-only, or indexer properties.**

**These collection properties can't be bound.**

**The binder can bind simple and complex object types, and read-only properties with a default.**

**The backing fields for implementing SetOnly and Indexer properties—not bound directly**

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 keys, 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)`.

The Options pattern is used throughout ASP.NET Core, but not everyone is a fan. In the next section you'll see how to use strongly typed settings and the configuration binder *without* the Options pattern.

#### 11.4.4 Binding strongly typed settings without the `IOptions` interface

The `IOptions` interface is very much canonical in ASP.NET Core—it's used by the core ASP.NET Core libraries and has various convenience functions for binding strongly typed settings, as you've already seen.

In many cases, however, the `IOptions` interface doesn't give many benefits for *consumers* of the strongly typed settings objects. Services must take a dependency on the `IOptions` interface but then immediately extract the “real” object by calling `IOptions<T>.Value`. This can be especially annoying if you're building a reusable library that isn't inherently tied to ASP.NET Core, as you must expose the `IOptions<T>` interface in all your public APIs.

Luckily, the configuration binder that maps `IConfiguration` objects to strongly typed settings objects isn't inherently tied to `IOptions`. Listing 11.13 shows how you can manually bind a strongly typed settings object to a configuration section and register it with the DI container.

##### Listing 11.13 Configuring strongly typed settings without `IOptions` in `Startup.cs`

```
public IConfiguration Configuration { get; }
public void ConfigureServices(IServiceCollection services)
{
    var settings = new MapSettings();
    Create a new instance of
    the MapSettings object.
    services.Configure<MapSettings>(options => options.MapPath = "map");
}
```

```

        Configuration.GetSection("MapSettings").Bind(settings);
        services.AddSingleton(settings); ←
    }
}

```

**Register the settings object as a singleton.**

**Bind the MapSettings section in IConfiguration to the settings object.**

You can now inject the `MapSettings` object directly into your services without the additional ceremony required to use `IOptions<MapSettings>`.

```

public class MyMappingController
{
    private readonly MapSettings _settings;
    public MyMappingController(MapSettings settings)
    {
        _settings = settings;
    }
}

```

If you use this approach, you won't benefit from the ability to reload strongly typed settings without further work, or from some of the more advanced usages of `IOptions`, but in most cases that's not a big problem. I'm a fan of this approach generally, but as always, consider what you're losing before adopting it wholeheartedly.

**TIP** In chapter 19 I show one such scenario, where you configure an `IOptions` object using services in your DI container. For other advanced scenarios see Microsoft's "Options pattern in ASP.NET Core" documentation, <http://mng.bz/DR7y>, or see the various `IOptions` posts on my blog, such as this one: <http://mng.bz/l1Aj>.

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, and 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 various 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 11.14 Adding environment-specific `appsettings.json` files

```
public class Program
{
    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); ←
    }
}
```

Adds an optional environment-specific JSON file where the filename varies with the environment

The current `IHostEnvironment` is available on `HostBuilderContext`.

It's common to make the base `appsettings.json` compulsory.

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 11.15 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())
        {
            builder.AddUserSecrets<Startup>();
        }
    }
}
```

**Extension methods make checking the environment simple and explicit.**

**In Staging and Production, the User Secrets provider won't be used.**

It's also common to customize your application's middleware pipeline based on the environment. 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 11.16 Using the hosting environment to customize your middleware pipeline**

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

```
if (env.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
}
else
{
    app.UseExceptionHandler("/Error");
}

app.UseHttpsRedirection();
app.UseStaticFiles();

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

}
```

In development, DeveloperExceptionPageMiddleware is added to the pipeline.

In staging or production, the pipeline uses ExceptionHandlerMiddleware instead.

**NOTE** In listing 11.16 we injected `IWebHostEnvironment` instead of `IHostEnvironment`. This interface extends `IHostEnvironment` by adding the `WebRootPath` property—the path to the `wwwroot` folder in your application. We don't need that path here, but it's good to be aware of its existence!

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.

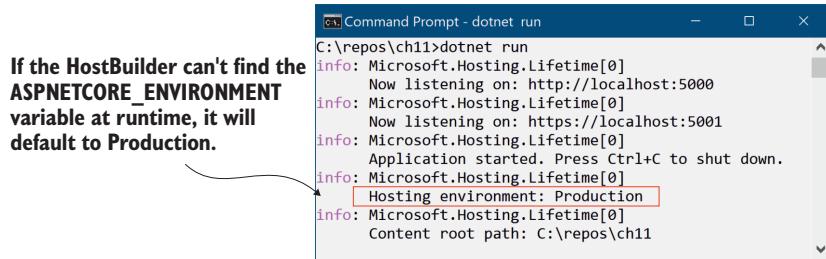
As useful as it is, setting `IHostEnvironment` with an environment variable can be a little cumbersome if you want to switch back and forth between different environments during testing. Personally, I'm always forgetting how to set environment variables on the various operating systems I use. The final 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. This 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.

**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.



**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.

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 I 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 it’s conventionally named the same as the project containing the `launchSettings.json` file.

#### Listing 11.17 A typical `launchSettings.json` file defining two profiles

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:53846",
      "sslPort": 44399
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "StoreViewerApplication": {
      "commandName": "Project",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

If true, launches the browser when you run the application

If true, launches the browser when you run the application

Defines settings for when running behind IIS or using the IIS Express profile

The IIS Express profile is used by default in Visual Studio on Windows.

Defines custom environment variables for the profile. Sets the environment to Development.

The “project” profile, equivalent to calling `dotnet run` on the project

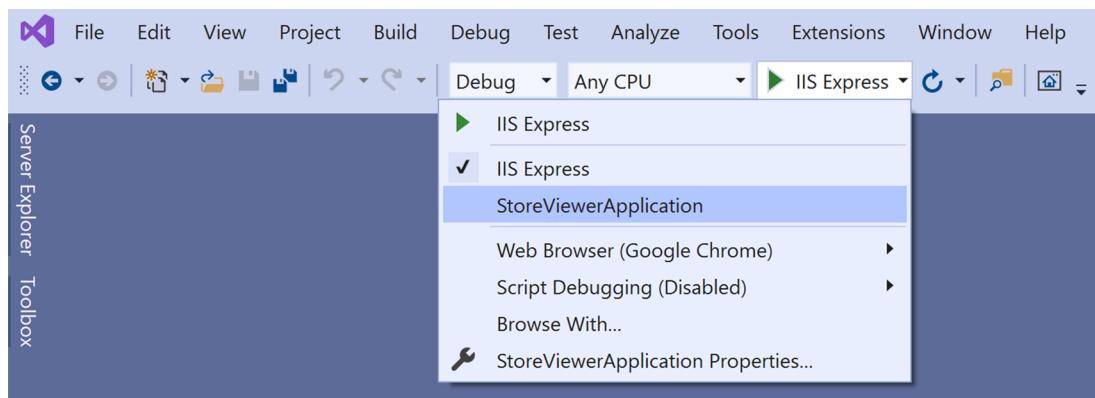
```
"environmentVariables": {  
    "ASPNETCORE_ENVIRONMENT": "Development"  
},  
"applicationUrl":  
    "https://localhost:5001;http://localhost:5000"  
}  
}  
}
```

Each profile can have different environment variables.

Defines the URLs the application will listen on in this profile

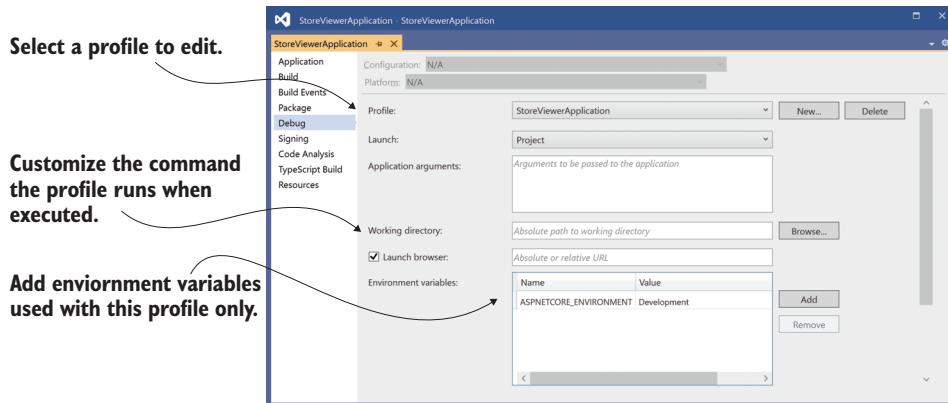
The advantage of using the `launchSettings.json` file locally is that it allows you to set “local” environment variables for a project. For example, in listing 11.17, the environment is set to the development environment. This lets you use different environment variables for each project, and even for each profile, and store them in source control.

You can choose a profile to use in Visual Studio by selecting from the drop-down 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 drop-down list. 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`.



**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 box.

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 like this:

```
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 they 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.

## 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 Key Vault.
- 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` file. 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`.

# *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 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, and 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 and .NET 5.0: 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 it 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 learn enough 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*, 2nd ed., 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's LocalDB feature. This is installed as part of Visual Studio 2019 (when you choose the ASP.NET and Web Development workload), and it provides a lightweight SQL Server engine.<sup>1</sup> 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, and you'll learn about 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, so if you choose to use EF Core in your

---

<sup>1</sup> You can read more about LocalDB in Microsoft's "SQL Server Express LocalDB" documentation at <http://mng.bz/5jEa>.

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.

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.

**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.

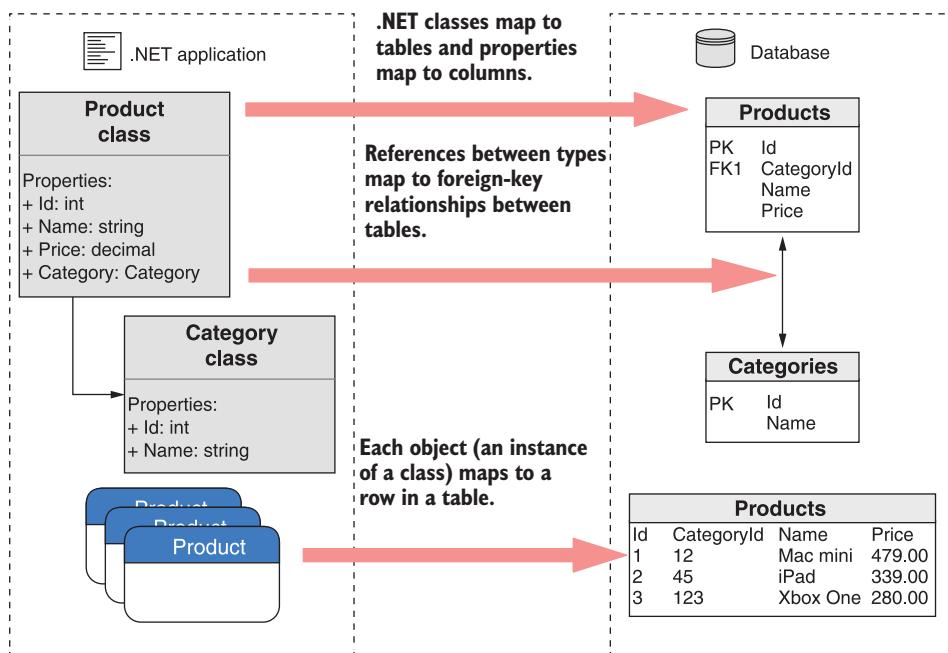


Figure 12.1 EF Core maps .NET classes and objects to database concepts such as tables and rows.

**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, often 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 retrieve 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 the database from you. 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 several concepts, so they inherently do more work than if you were to handcraft 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, 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 are 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 reimaging 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 those 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 5.0 targets .NET Standard, so it can be used in cross-platform apps that target .NET Core 3.0 or later. Since version 6.3, EF 6.x is *also* cross-platform, with some limitations when running on .NET 5.0, 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 5.0. 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, though this difference is smaller with EF Core 5.0 than ever before. EF Core now has many features that EF 6.x doesn’t have (batching statements, client-side key generation, in-memory database for testing). EF Core is still missing some features compared to EF 6.x, such as stored procedure mapping and Table-Per-Concrete-Type (TPC), but as EF Core is under active development, these features are on the backlog for implementation.<sup>2</sup> 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,

---

<sup>2</sup> For a detailed list of feature differences between EF 6.x and EF Core, see the documentation at <http://mng.bz/GxgA>.

and keep an eye on the guidance from the EF team here: <https://docs.microsoft.com/ef/efcore-and-ef6>.

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.

This is obviously a simple application, but it contains all the database interactions you need with its two *entities*: Recipe and Ingredient.

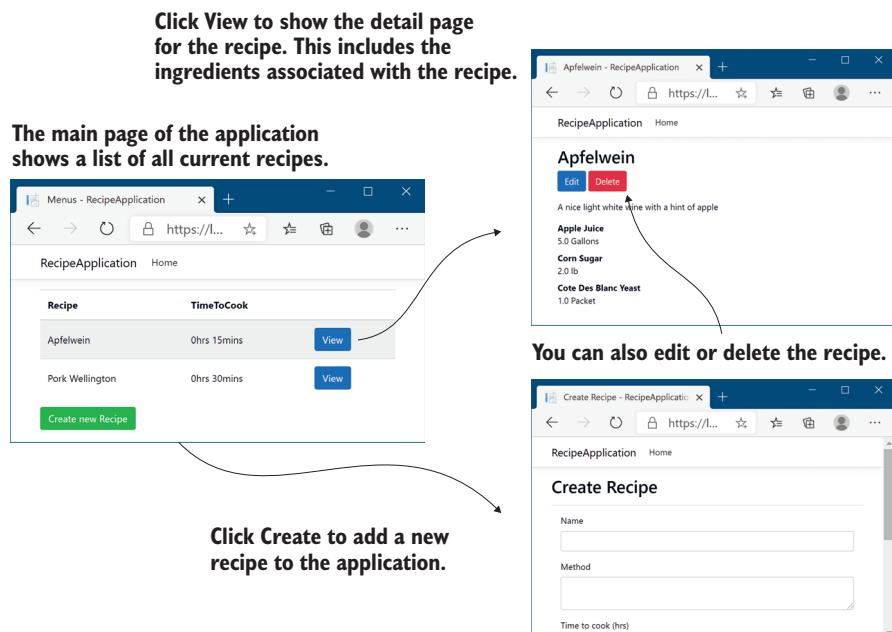


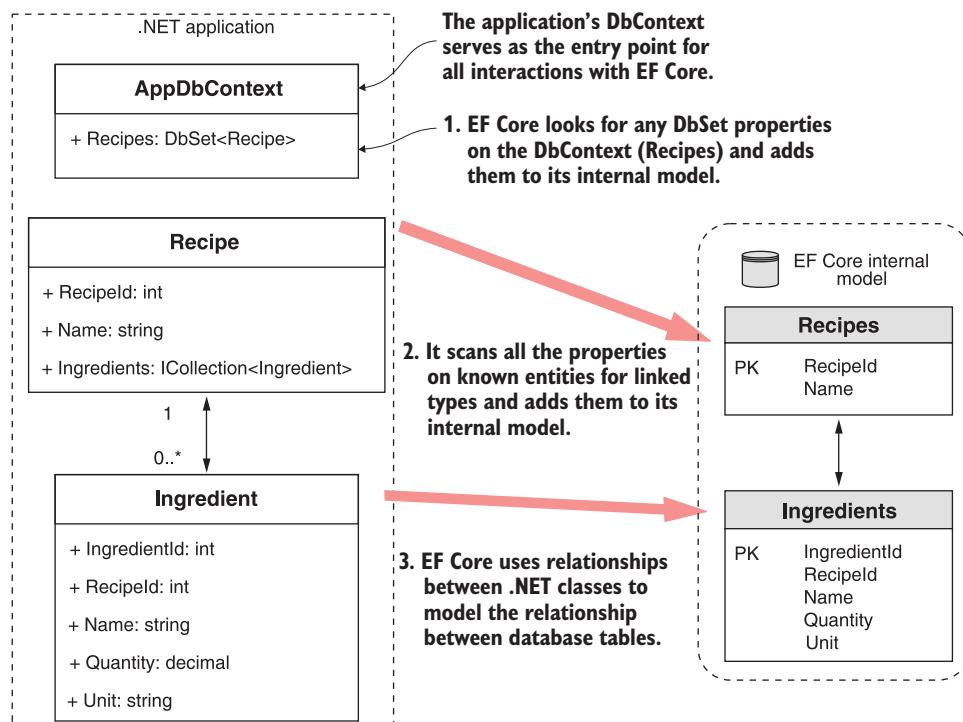
Figure 12.2 The cookery app lists recipes. You can view, update, and delete recipes, or create new ones.

**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.



**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<T>` 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, such as 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

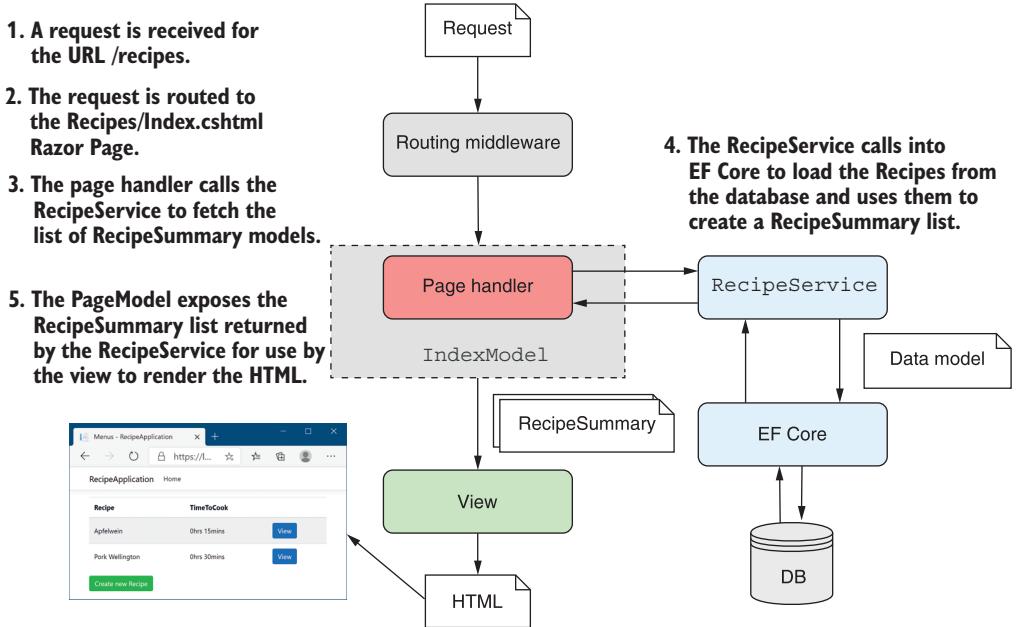
In this section, we'll focus on getting EF Core installed and configured in your ASP.NET Core recipe app. You'll learn how to install the required NuGet packages and how to build the data model for your application. As we're talking about EF Core in this chapter, I'm not going to go into how to create 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: at 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 LocalDB 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.

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.



**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.

- 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 need to install multiple 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>net5.0</TargetFramework>           <!-- The app targets .NET 5.0. -->
    </PropertyGroup>

    <ItemGroup>
        <PackageReference
            Include="Microsoft.EntityFrameworkCore.SqlServer"
            Version="5.0.0" />
        <PackageReference
            Include="Microsoft.EntityFrameworkCore.Design"
            Version="5.0.0" />
    </ItemGroup>
</Project>
```

The app targets .NET 5.0.

Install the appropriate NuGet package for your selected DB.

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 you to 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; } ←
}

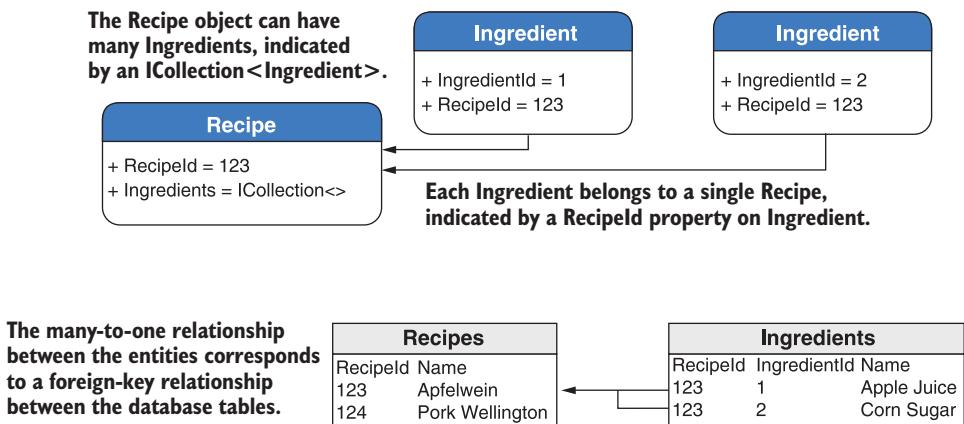
public class Ingredient
{
    public int IngredientId { get; set; }
    public int RecipeId { get; set; }
    public string Name { get; set; }
    public decimal Quantity { get; set; }
    public string Unit { get; set; }
}
```

A 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 an `Id` suffix 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.



**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)
        : base(options) { }
    public DbSet<Recipe> Recipes { get; set; } ←
}
```

You'll use the `Recipes` property to query the database.

The constructor `options` object, containing details such as the connection string

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 must navigate *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 in Microsoft's "Reverse Engineering" article in the EF Core documentation: <http://mng.bz/mgd4>.)

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 we'll 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
        .GetConnectionString("DefaultConnection");
```

The connection string is taken from configuration, from the `ConnectionString` section.

```

services.AddDbContext<AppDbContext>(
    options => options.UseSqlServer(connString));
    ↑
    | Specify the database provider
    | in the customization options
    | for the DbContext.
    |
}

// Add other services.
}

```

**Register your app's DbContext by using it as the generic parameter.**

**NOTE** If you're using a different database provider, such as 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 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<sup>3</sup> to keep track of which scripts have been applied and ensure your database schema is up to date. Alternatively, you can use external tools to 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 roll back 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 roll back the migration, the table will be recreated, but the data it previously contained will be gone forever!

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*—You can use PowerShell cmdlets 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 tool*—Cross-platform tooling that you can run from the command line and which extends the .NET SDK. You can install these tools globally for your machine by running `dotnet tool install --global dotnet-ef`.<sup>4</sup>

In this book, I'll be using the cross-platform .NET tools, but if you're familiar with EF 6.x or prefer to use the Visual Studio PMC, there are equivalent commands for all of the steps you're going to take.<sup>5</sup> You can check that the .NET tool installed correctly by running `dotnet ef`. This should produce a help screen like the one shown in figure 12.6.

---

<sup>3</sup> DbUp (<https://github.com/DbUp/DbUp>) and FluentMigrator (<https://github.com/fluentmigrator/fluentmigrator>) are open source projects.

<sup>4</sup> Alternatively, you can install the tools *locally* using a *tool manifest file*. To see how to use this approach, see my blog article: <https://andrewlock.net/new-in-net-core-3-local-tools/>.

<sup>5</sup> Documentation for PowerShell cmdlets can be found at <https://docs.microsoft.com/ef/core/miscellaneous/cli/powershell>.

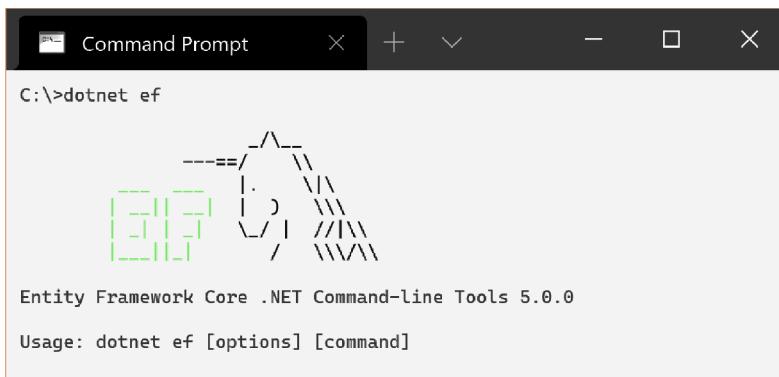


Figure 12.6 Running the `dotnet ef` command to check that the .NET EF Core tools are installed correctly

**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 `DbContext` (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 that 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 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 depends on how you've designed your application, how you'll update your production database, and your personal preference. I'll use the .NET tool for now, but I discuss some of these considerations in section 12.5.

You can apply migrations to a database by running

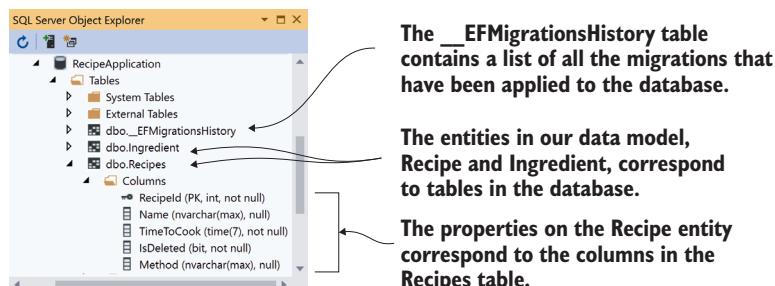
```
dotnet ef database update
```

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, it 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.

**NOTE** If you get an error message saying "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.



**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. 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. Change your entities to your desired state, generate a migration, and apply it to the database, as shown in figure 12.8.

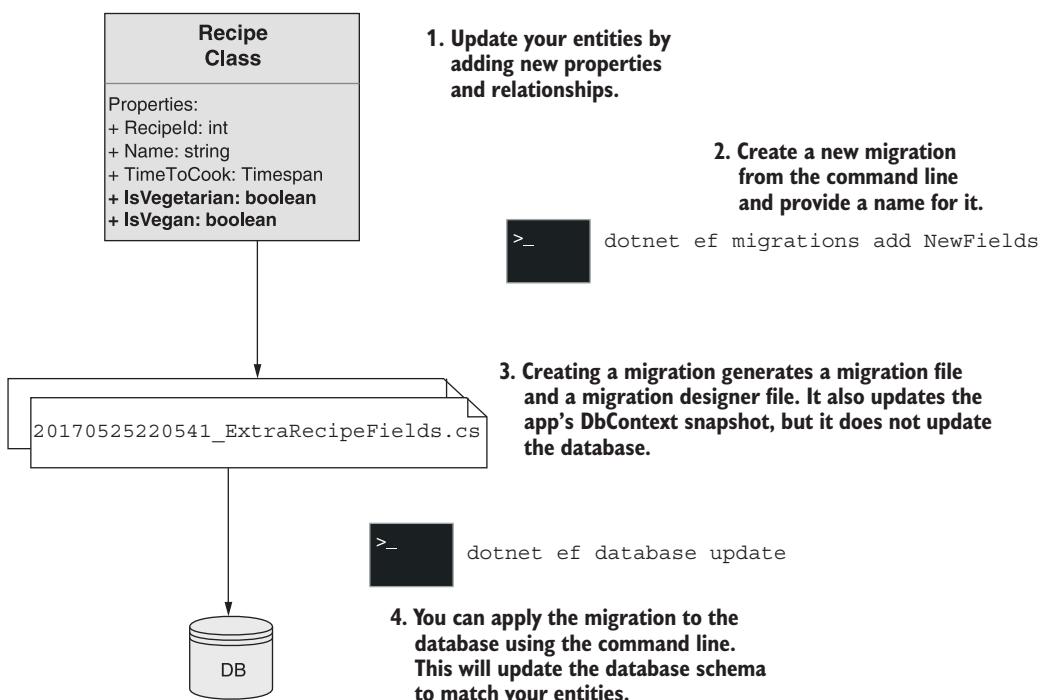


Figure 12.8 Creating a second migration and applying it to the database using the command-line tools.

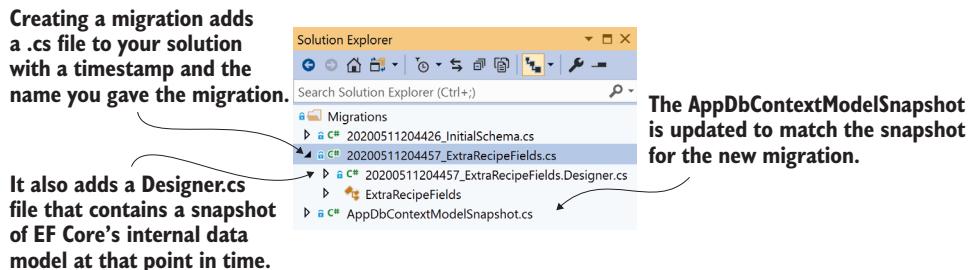
**Listing 12.5 Adding properties to the Recipe entity**

```
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.

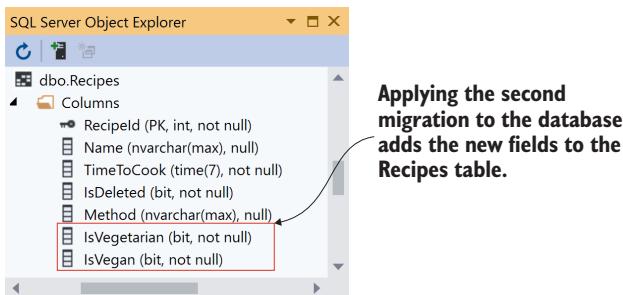


**Figure 12.9** Adding a second migration adds a new migration file and a migration `Designer.cs` file. It also updates `AppDbContextModelSnapshot` to match the new migration's `Designer.cs` file.

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 in your database to see which migrations are outstanding, and then it 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.



**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 will handle 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.

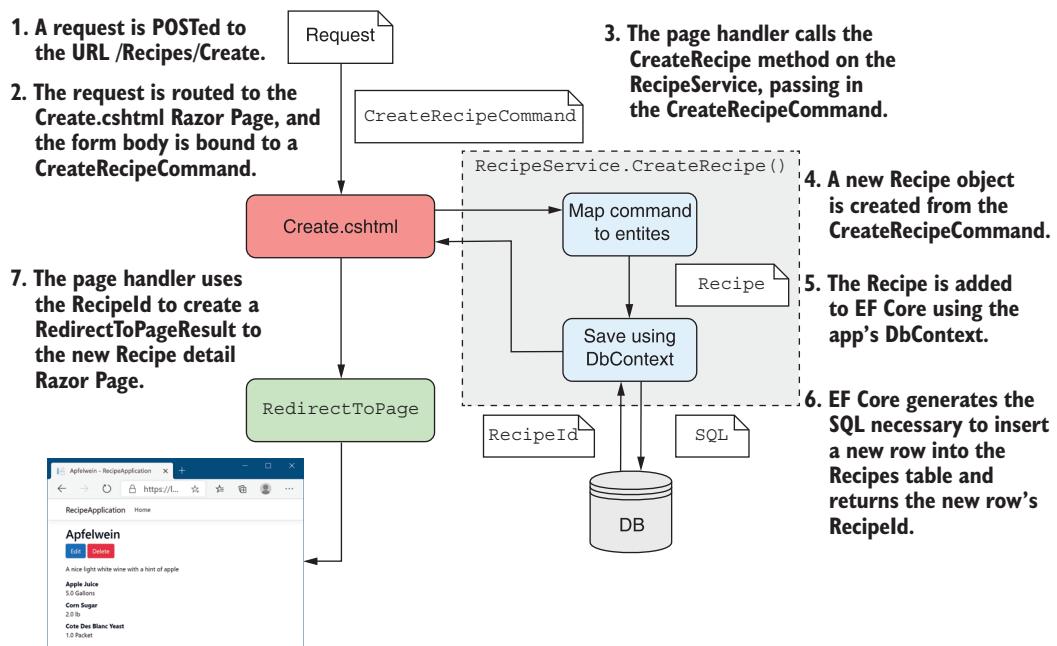
Our database doesn't have any data in it yet, so we'd better start by letting you create a recipe.

### 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.



**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 it 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 my blog post on the subject: <http://mng.bz/d48O>.

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;           ← An instance of the AppDbContext is
 public async Task<int> CreateRecipe(CreateRecipeCommand cmd)   ← injected in the class constructor using DI.
 {
    var recipe = new Recipe
    {
        Name = cmd.Name,
        TimeToCook = new TimeSpan(
            cmd.TimeToCookHrs, cmd.TimeToCookMins, 0),
        Method = cmd.Method,
        IsVegetarian = cmd.IsVegetarian,
        IsVegan = cmd.IsVegan,
        Ingredients = cmd.Ingredients?.Select(i =>
        new Ingredient
        {
            Name = i.Name,
            Quantity = i.Quantity,
            Unit = i.Unit,
        }).ToList()
    };
    _context.Add(recipe);
    await _context.SaveChangesAsync();
    return recipe.RecipeId;   ← Tell EF Core to write the entities
}                                         ← to the database. This uses the
                                         ← async version of the command.
                                         ← EF Core populates the RecipeId field
                                         ← on your new Recipe when it's saved.
```

← CreateRecipeCommand is passed in from the Razor Page handler.

← Create a Recipe by mapping from the command object to the Recipe entity.

← Map each CreateIngredientCommand onto an Ingredient entity.

Tell EF Core to track the new entities.

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

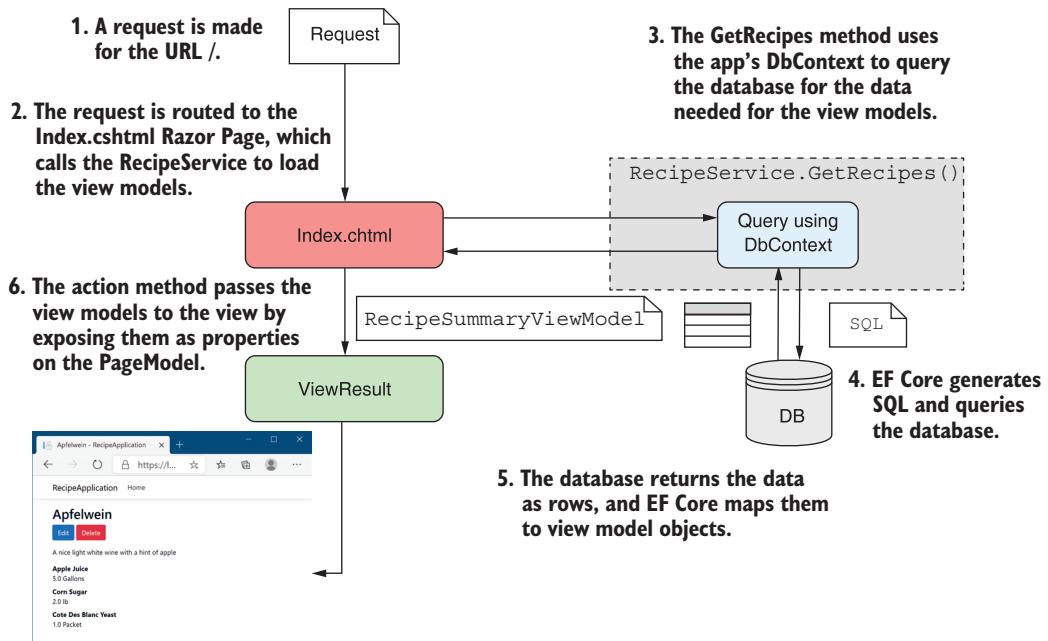
Now that you can create recipes, you need to write the code to view them. Luckily, loading data is simple in EF Core, relying heavily on LINQ methods to control which fields you need. For your app, you'll create a method on `RecipeService` that returns a summary view of a recipe, consisting of the `RecipeId`, `Name`, and `TimeToCook` as a `RecipeSummaryViewModel`, as shown in figure 12.12.

**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.

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()`, `ToByteArrayAsync()`, `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.



**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.

```
_context.Recipes.Where(r => !r.IsDeleted).ToListAsync()
```

AppDbContext DbSet	LINQ commands to modify data returned	Execute query command
Property access		

**Figure 12.13** The three parts of an EF Core database query

Listing 12.7 shows the code to fetch a list of `RecipeSummaryViewModels`, 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 in RecipeService

```
public async Task<ICollection<RecipeSummaryViewModel>> GetRecipes()
{
    return await _context.Recipes           ← A query starts from
        .Where(r => !r.IsDeleted)          a DbSet property.
```

**EF Core will only query the Recipe columns it needs to map the view model correctly.**

```
.Select(r => new RecipeSummaryViewModel
{
    Id = r.RecipeId,
    Name = r.Name,
    TimeToCook = $"{r.TimeToCook.TotalMinutes}mins"
})
.ToListAsync();
```

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 = $"{r.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 <http://mng.bz/zxP6>.

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 in RecipeService**

```

public async Task<RecipeDetailViewModel> GetRecipeDetail(int id)
{
    return await _context.Recipes
        .Where(x => x.RecipeId == id)
        .Select(x => new RecipeDetailViewModel
    {
        Id = x.RecipeId,
        Name = x.Name,
        Method = x.Method,
        Ingredients = x.Ingredients
        .Select(item => new RecipeDetailViewModel.Item
        {
            Name = item.Name,
            Quantity = $"{item.Quantity} {item.Unit}"
        })
    })
    .SingleOrDefaultAsync();
}

```

The id of the recipe to load is passed as a parameter.

As before, a query starts from a DbSet property.

Map the Recipe to a RecipeDetailViewModel.

Load and map linked Ingredients as part of the same query.

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 Ingredients 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.

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.<sup>6</sup>

<sup>6</sup> For a detailed discussion on handling relationship updates in EF Core, see *Entity Framework Core in Action*, 2nd ed., by Jon P. Smith (Manning, 2021), chapter 3, section 3.4: <http://mng.bz/w9D2>.

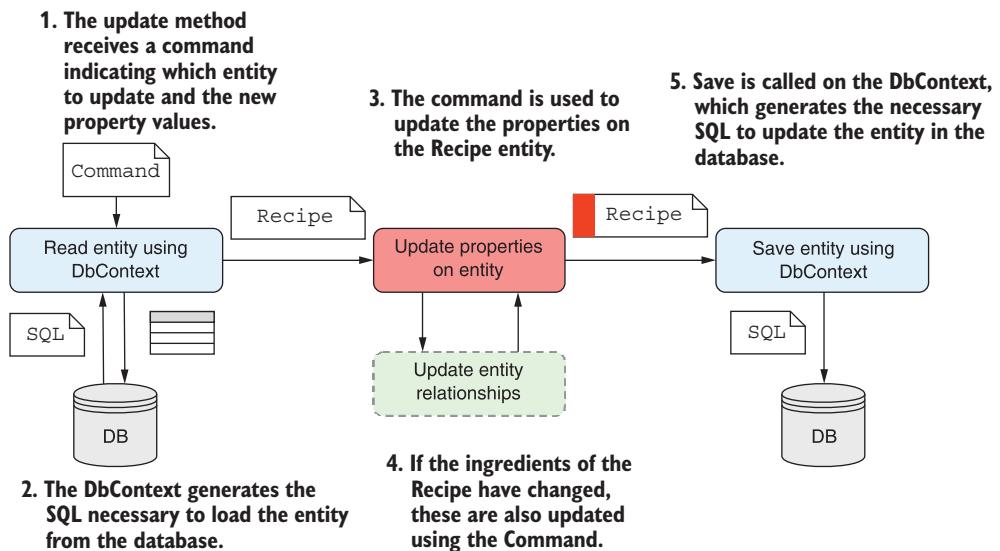


Figure 12.14 Updating an entity involves three steps: read the entity using EF Core, update the properties of the entity, and call `SaveChangesAsync()` on the `DbContext` to generate the SQL to update the correct rows in the database.

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` called `UpdateRecipe`. This method takes an `UpdateRecipeCommand` parameter and contains the code to change the `Recipe` entity.

**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 performs 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 in `RecipeService`

```
public async Task UpdateRecipe(UpdateRecipeCommand cmd)
{
    var recipe = await _context.Recipes.FindAsync(cmd.Id);
```

Find is exposed directly by `Recipes` and simplifies reading an entity by id.

```

Set the new values on the Recipe entity.    if(recipe == null) {
    throw new Exception("Unable to find the recipe");
}
UpdateRecipe(recipe, cmd);
await _context.SaveChangesAsync();
}

static void UpdateRecipe(Recipe recipe, UpdateRecipeCommand cmd)
{
    recipe.Name = cmd.Name;
    recipe.TimeToCook =
        new TimeSpan(cmd.TimeToCookHrs, cmd.TimeToCookMins, 0);
    recipe.Method = cmd.Method;
    recipe.IsVegetarian = cmd.IsVegetarian;
    recipe.IsVegan = cmd.IsVegan;
}

```

If an invalid id is provided, recipe will be null.

Execute the SQL to save the changes to the database.

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 have 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 in this request), 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.

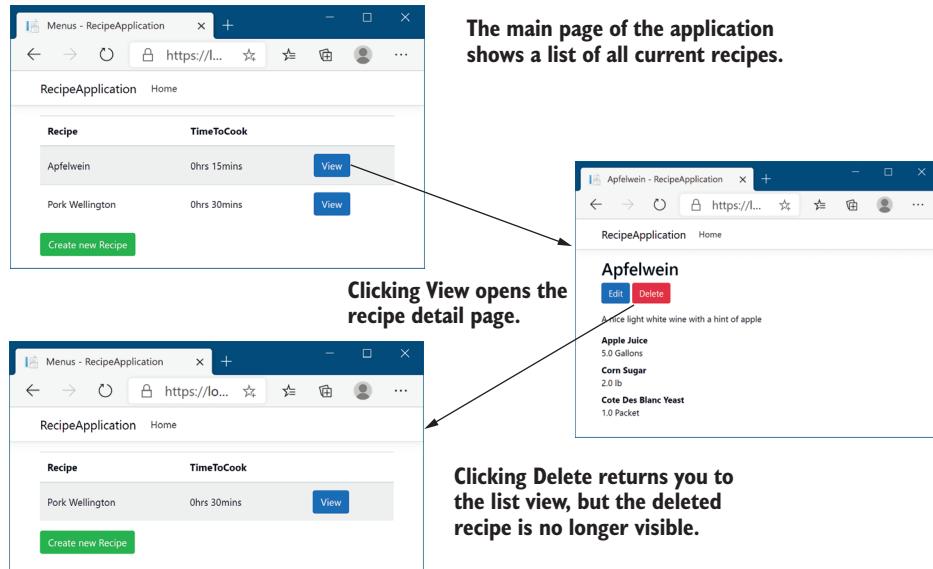
You may be wondering how EF Core knows which columns to update when you call `SaveChangesAsync()`. The simplest approach would be to update every column—if the field hasn't changed, then it doesn't matter if you write the same value again. But EF Core is a bit more clever than that.

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, and 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 chapter 3 of Jon P. Smith's *Entity Framework Core in Action*, 2nd ed., (Manning, 2021) for details: <http://mng.bz/q9PJ>.

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.



**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.

You *could* achieve this by deleting the recipe from the database, but the problem with data is that 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, deleting data suddenly 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 that 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);
    if(recipe is null) {
        throw new Exception("Unable to find the recipe");
    }
    recipe.IsDeleted = true;
    await _context.SaveChangesAsync();
}
```

The code is annotated with several callouts:

- A callout labeled "Mark the Recipe as deleted." points to the line `recipe.IsDeleted = true;`.
- A callout labeled "Fetch the Recipe entity by id." points to the line `var recipe = await _context.Recipes.FindAsync(recipeId);`.
- A callout labeled "If an invalid id is provided, recipe will be null." points to the line `if(recipe is null) {`.
- A callout labeled "Execute the SQL to save the changes to the database." points to the line `await _context.SaveChangesAsync();`.

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 learn more about 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 *Entity Framework Core in Action*, 2nd ed., by Jon P. Smith (Manning, 2021) for details (<http://mng.bz/7Vme>) or exploring the EF Core documentation site at <https://docs.microsoft.com/ef/core/>.

The following topics aren't essential for 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 tackling 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 to use `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 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 than 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.

## 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 it has a different feature set than 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 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 `DataAnnotations`, 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 that 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 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: reading the entity from the database, modifying the entity, and saving 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.

# 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 cross-cutting 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, and I cover that in section 13.3.

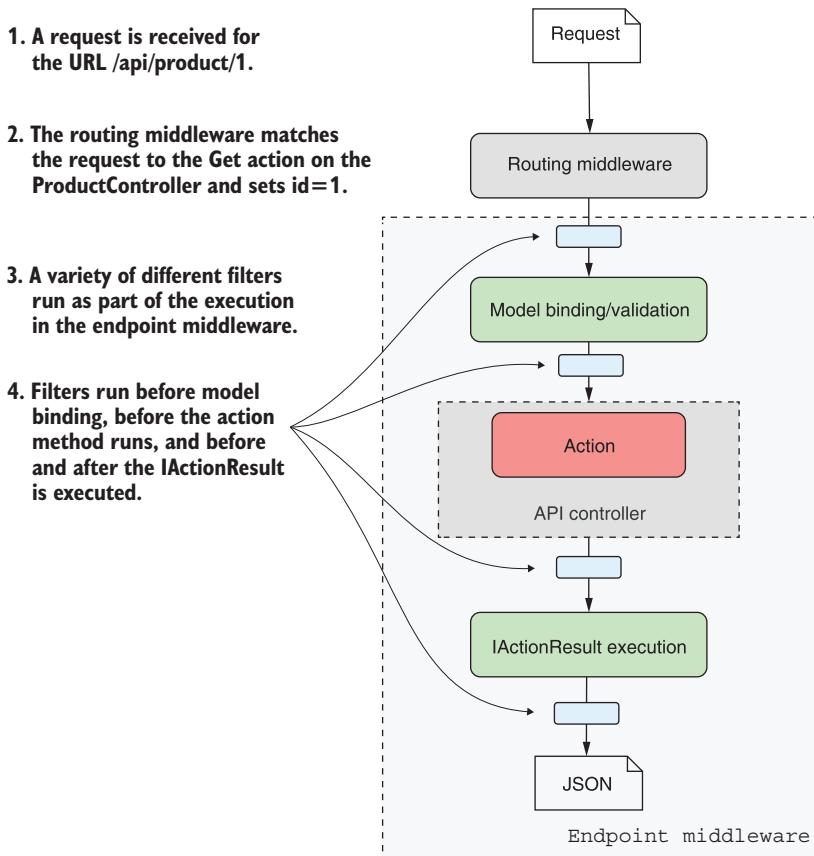
You typically add filters to the pipeline by implementing them as attributes added to your controller classes, action methods, and Razor Pages. Unfortunately, you can't easily use DI with attributes due to the limitations of C#. In section 13.4 I'll show you how to use the `ServiceFilterAttribute` and `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 filters, 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.



**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

- Ensure a user is logged in before an action method, model binding, or validation runs
- Customize the output format of particular action methods
- Handle model validation failures before an action method is invoked
- Catch exceptions from an action method and handle 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 cross-cutting 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. The 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 filters that apply to MVC requests, each of which runs at a different *stage* in the MVC framework, as shown in figure 13.2.

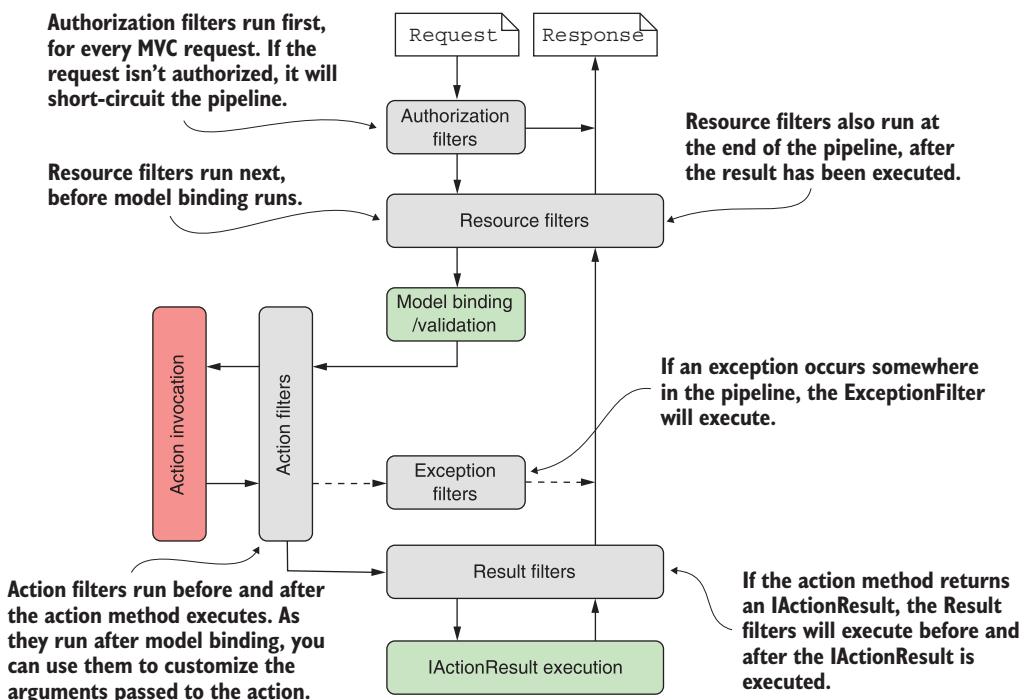


Figure 13.2 The MVC filter pipeline, including the five different filter stages. Some filter stages (resource, action, and result) run twice, before and after the remainder of the pipeline.

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 they’re 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 that let you run code at a particular point in a request’s lifecycle.

This section described how the filter pipeline works for MVC controllers, such as you would use to create APIs; 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.

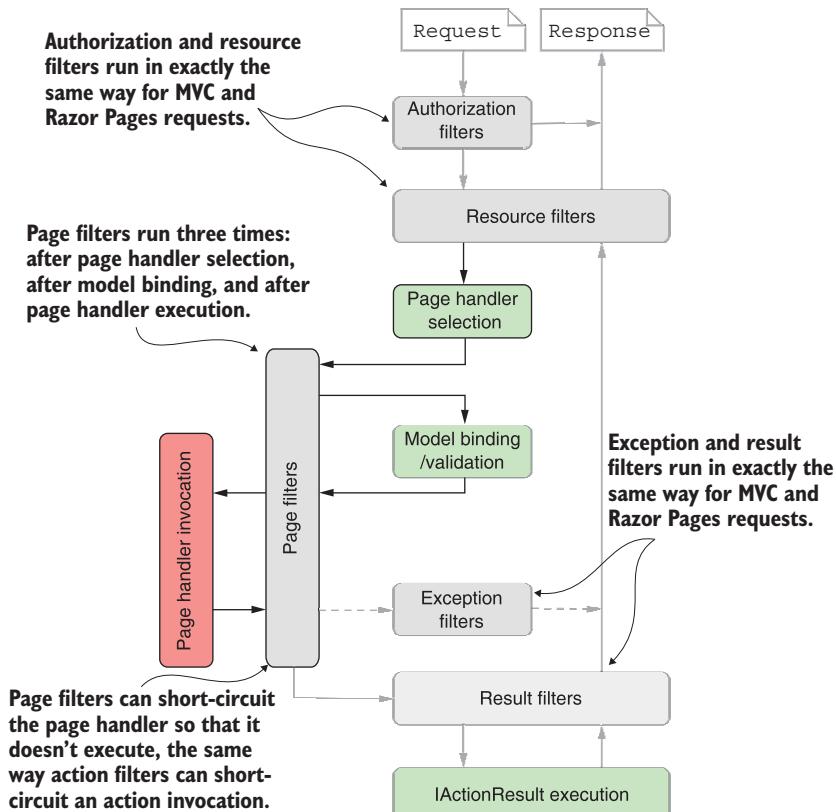


Figure 13.3 The Razor Pages filter pipeline, including the five different filter stages. Authorization, resource, exception, and result filters execute in exactly the same way as for the MVC pipeline. Page filters are specific to Razor Pages and execute in three places: after page handler selection, 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 cross-cutting application concerns, such as logging, performance profiling, and exception handling.* Filters also lend themselves to cross-cutting 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 and works at a “lower level,” so it 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, which operates on lower-level primitives like the `HttpContext`, so it has the wider reach. If the functionality you need has no MVC-specific requirements, 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 cross-cutting 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: <http://mng.bz/RXa0>.

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'll show you 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, 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(
        ResourceExecutingContext context) ←
    {
        Console.WriteLine("Executing!"); ←
    }
}
```

**Executed at the start of the pipeline,  
after authorization filters**

**The context contains the `HttpContext`,  
routing details, and information  
about the current action.**

```
public void OnResourceExecuted(
    ResourceExecutedContext context)
{
    Console.WriteLine("Executed" );
}
```

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 we create 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(
        ResourceExecutingContext context,
        ResourceExecutionDelegate next) {
        Console.WriteLine("Executing async!");
        ResourceExecutedContext executedContext = await next();
        Console.WriteLine("Executed async!");
    }
}
```

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]
    public IActionResult Index()
    {
        return Ok();
    }
    public IActionResult View()
    {
        return OK();
    }
}
```

**LogResourceFilter**  
will run as part of the pipeline when executing this action.

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]
public class RecipeController : ControllerBase
{
    public IActionResult Index()
    {
        return Ok();
    }
    public IActionResult View()
    {
        return Ok();
    }
}
```

The LogResourceFilter is added to every action on the controller.

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]
public class IndexModel : PageModel
{
    public void OnGet()
    {
    }
    public void OnPost()
    {
    }
}
```

The LogResourceFilter is added to the Razor Page's PageModel.

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 than 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 =>
        {
            options.Filters.Add(new LogResourceFilter());
            options.Filters.Add(typeof(LogResourceFilter));
            options.Filters.Add<LogResourceFilter>();           ←
        });
    }
}
```

**You can pass an instance of the filter directly...** → A callout points to the first line of the options lambda.

**Adds filters using the MvcOptions object** ← An arrow points from the 'options' variable back to the 'AddControllers' method.

**... or pass in the Type of the filter and let the framework create it.** ← An arrow points from the 'typeof(LogResourceFilter)' line to the 'Add<LogResourceFilter>' line.

**Alternatively, the framework can create a global filter using a generic type parameter.** ← An arrow points from the 'Add<LogResourceFilter>' line to the explanatory text.

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()
            .AddMvcOptions(options =>
        {
            options.Filters.Add(new LogResourceFilter());
            options.Filters.Add(typeof(LogResourceFilter));
            options.Filters.Add<LogResourceFilter>();
        });
    }
}
```

**This method doesn't let you pass a lambda to configure the MvcOptions.** → A callout points to the 'AddMvcOptions' line.

**You must use an extension method to add the filters to the MvcOptions object.** ← An arrow points from the 'AddMvcOptions' line to the explanatory text.

**You can configure the filters in any of the ways shown previously.** ← An arrow points from the explanatory text to the 'Add<LogResourceFilter>' line.

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 and 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.

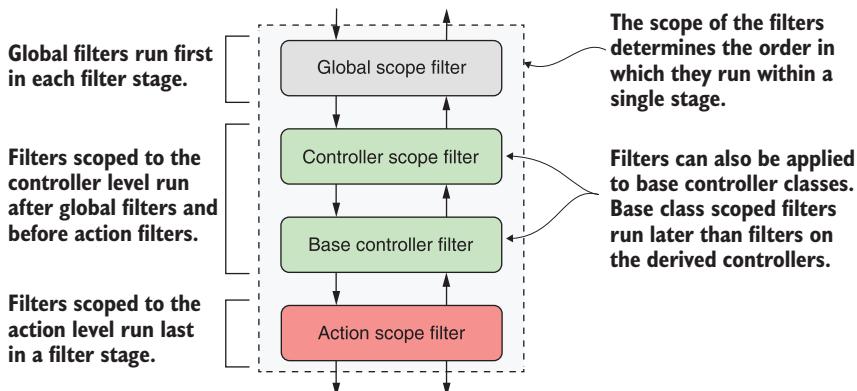


Figure 13.4 The default filter ordering within a given stage, based on the scope of the filters. For the `*Executing` method, globally scoped filters run first, followed by controller-scoped, and finally, action-scoped filters. For the `*Executed` method, the filters run in reverse order.

By default, filters execute from the broadest scope (global) to the narrowest (action) when running the `*Executing` method for each stage. The filters' `*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.

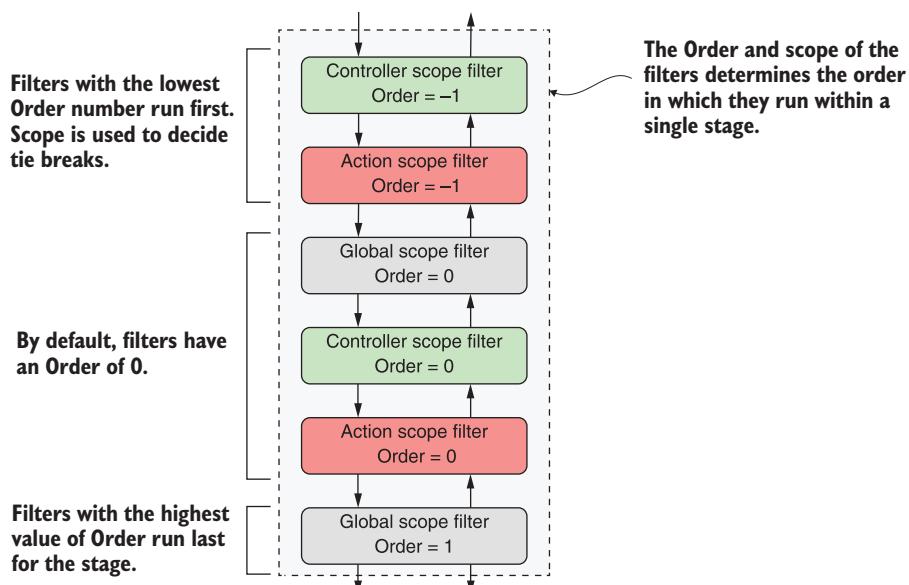
#### **OVERRIDING THE DEFAULT ORDER OF FILTER EXECUTION WITH `IORDEREDFILTER`**

Filters are great for extracting cross-cutting concerns from your controller actions and Razor Page, but if you have multiple filters applied to an action, 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 `IOrderedFilter`. 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 given 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.



**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 we'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, we'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;           ← This field would be passed in
    public RecipeService _service;                 as configuration and is used
    public RecipeApiController(RecipeService service)   to control access to actions.
    {
        _service = service;
    }

    [HttpGet("{id}")]
    public IActionResult Get(int id)
    {
        if (!.IsEnabled) { return BadRequest(); } ← If the API isn't
        try                                enabled, block
        {
            if (!_service.DoesRecipeExist(id))  further execution.
            {
                return NotFound();
            }
            var detail = _service.GetRecipeDetail(id); ← If the requested Recipe
        }                                         doesn't exist, return a
    }
}
```

**Fetch RecipeDetail ViewModel.**

```

        Response.GetTypedHeaders().LastModified =
            detail.LastModified;
        return Ok(detail);
    }
    catch (Exception ex)
    {
        return GetErrorResponse(ex);
    }
}

[HttpPost("{id}")]
public IActionResult Edit(
    int id, [FromBody] UpdateRecipeCommand command)
{
    if (!IsEnabled) { return BadRequest(); } ←
    try
    {
        if (!ModelState.IsValid)
        {
            return BadRequest(ModelState);
        }
        if (!_service.DoesRecipeExist(id))
        {
            return NotFound();
        }
        _service.UpdateRecipe(command);
        return Ok();
    }
    catch (Exception ex)
    {
        return GetErrorResponse(ex);
    }
}

private static IActionResult GetErrorResponse(Exception ex)
{
    var error = new ProblemDetails
    {
        Title = "An error occurred",
        Detail = context.Exception.Message,
        Status = 500,
        Type = "https://httpstatuses.com/500"
    };

    return new ObjectResult(error)
    {
        StatusCode = 500
    };
}
}

```

**Returns the view model with a 200 response**

**If the requested Recipe doesn't exist, return a 404 response.**

**Sets the Last-Modified response header to the value in the model**

**If an exception occurs, catch it and return the error in an expected format, as a 500 error.**

**If the API isn't enabled, block further execution.**

**Validate the binding model and return a 400 response if there are errors.**

**Update the Recipe from the command and return a 200 response.**

**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)]
public class RecipeApiController : ControllerBase
{
    public RecipeService _service;
    public RecipeApiController(RecipeService service)
    {
        _service = service;
    }

    [HttpGet("{id}"), EnsureRecipeExists, AddLastModifiedHeader]
    public IActionResult Get(int id)
    {
        var detail = _service.GetRecipeDetail(id);
        return Ok(detail);
    }

    [HttpPost("{id}"), EnsureRecipeExists]
    public IActionResult Edit(
        int id, [FromBody] UpdateRecipeCommand command)
    {
        _service.UpdateRecipe(command);
        return Ok();
    }
}
```

The filters encapsulate the majority of logic common to multiple action methods.

Placing filters at the action level limits them to a single action.

The intent of the action, return a Recipe view model, is much clearer.

Placing filters at the action level can control the order in which they execute.

The intent of the action, update a Recipe, is much clearer.

I think you’ll have to agree that the controller in listing 13.9 is much easier to read! In this section, you’ll refactor the controller bit by bit, removing cross-cutting code to get to something more manageable. All the filters we’ll create in this section will use the sync filter interfaces—I’ll leave it to you, as an exercise, to create their async counterparts. We’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)           ← The Get method has no
    {                                         [Authorize] attribute, so it
        // method body                      can be executed by anyone.
    }
    [Authorize]                                ← Adds the AuthorizeFilter to the
    public IActionResult Edit(                 filter pipeline using [Authorize]
        int id, [FromBody] UpdateRecipeCommand command)
    {
        // method body
    }
}
```

The `Get` method has no `[Authorize]` attribute, so it can be executed by anyone.

← Adds the `AuthorizeFilter` to the filter pipeline using `[Authorize]`

← 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:

- `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, 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. The resource filters run before model binding, so you can disable the model binding for a single action in this way.<sup>1</sup>

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.<sup>2</sup>

This piece of code is self-contained cross-cutting 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.

**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.

---

<sup>1</sup> For details on handling file uploads, see Microsoft's documentation: <http://mng.bz/4Z2D>.

<sup>2</sup> To read more about using feature toggles in your applications, see my series: “Adding feature flags to an ASP.NET Core app” blog entry: <http://mng.bz/2e40>.

**Listing 13.11 The FeatureEnabledAttribute resource filter**

```

public class FeatureEnabledAttribute : Attribute, IResourceFilter
{
    public bool IsEnabled { get; set; }
    public void OnResourceExecuting(
        ResourceExecutingContext context)
    {
        if (!IsEnabled)
        {
            context.Result = new BadRequestResult();
        }
    }
    public void OnResourceExecuted(
        ResourceExecutedContext context) { }
}

```

**Defines whether the feature is enabled**

**Executes before model binding, early in the filter pipeline**

**If the feature isn't enabled, short-circuits the pipeline by setting the context.Result property**

**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`, which runs before model binding, and `*Executed`, which 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.

**NOTE** 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.

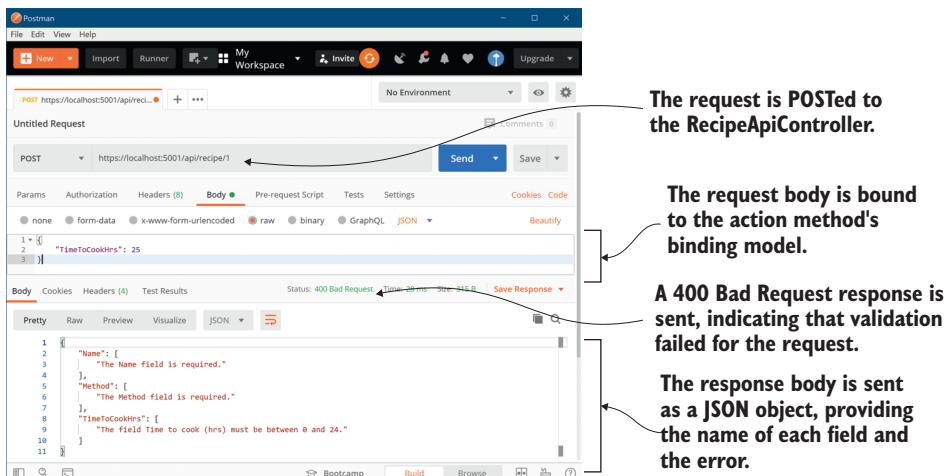
**TIP** Caching is a broad topic that aims to improve the performance of an application over the naive approach. But caching can also make debugging issues difficult and may even be undesirable in some situations. Consequently, I often apply `ResponseCacheFilter` to my action methods to set HTTP caching headers that *disable* caching! You can read about this and other approaches to caching in Microsoft's "Response caching in ASP.NET Core" documentation at <http://mng.bz/2eGd>.

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.

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



**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 Bad Request response with a list of the validation errors.

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

For convenience, you derive from the `ActionFilterAttribute` base class.

```
public class ValidateModelAttribute : ActionFilterAttribute
{
    public override void OnActionExecuting(
        ActionExecutingContext context)
    {
        if (!context.ModelState.IsValid)
        {
            context.Result =
                new BadRequestObjectResult(context.ModelState);
        }
    }
}
```

Model binding and validation have already run at this point, so you can check the state.

Overrides the Executing method to run the filter before the Action executes

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 this code from the start of both the action methods, as it will run automatically in the filter pipeline:

```
if (!ModelState.IsValid)
{
    return BadRequest(ModelState);
}
```

You'll use a similar approach to remove the duplicate code that checks whether the `id` provided as an argument to the action methods corresponds to an existing Recipe entity.

The following 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 EnsureRecipeExistsAttribute : ActionFilterAttribute
{
    public override void OnActionExecuting(
        ActionExecutingContext context)
    {
        var service = (RecipeService) context.HttpContext
            .RequestServices.GetService(typeof(RecipeService));
        var recipeId = (int) context.ActionArguments["id"];
        if (!service.DoesRecipeExist(recipeId))
        {
            context.Result = new NotFoundResult();
        }
    }
}
```

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.<sup>3</sup> In section 13.4 I'll show you 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 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 bypass the action method execution.

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, you can override the appropriate methods on your controller.

#### Listing 13.14 Overriding action filter methods directly on ControllerBase

```
public class HomeController : ControllerBase
{
    public override void OnActionExecuting(
        ActionExecutingContext context)
    {
    }
    public override void OnActionExecuted(
        ActionExecutedContext context)
    {
    }
}
```

Derives from the  
ControllerBase class

Runs before any other action filters  
for every action in the controller

Runs after all other action filters  
for every action in the controller

<sup>3</sup> For a detailed discussion of DI patterns and antipatterns, see chapter 5 of *Dependency Injection Principles, Practices, and Patterns* by Steven van Deursen and Mark Seemann (Manning, 2019) <http://mng.bz/RXOK>.

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 cross-cutting concerns, when you need behavior that's either specific to MVC or that 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 in your logs, such as passwords or credit card details.

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 that 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 `StatusCodeResults` 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 at these times:

- 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 that your resource and result filters don't throw exceptions. Similarly, they won't catch exceptions thrown when executing an `IActionResult`, such as 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
{
    public override void OnException(ExceptionContext context)
    {
        var error = new ProblemDetails
        {
            Title = "An error occurred",
            Detail = context.Exception.Message,
            Status = 500,
            Type = "https://httpstatuses.com/500"
        };

        context.Result = new ObjectResult(error)
        {
            StatusCode = 500
        };
        context.ExceptionHandled = true;
    }
}
```

**Building a problem details object to return in the response**

**ExceptionFilterAttribute is an abstract base class that implements `IExceptionFilter`.**

**There's only a single method to override for `IExceptionFilter`.**

**Creates an `ObjectResult` to serialize the `ProblemDetails` and to set the response status code**

**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, the next stage of the pipeline, after action filters, is 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, 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.<sup>4</sup>

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.

---

<sup>4</sup> 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 my blog entry on the topic: <http://mng.bz/1rYV>.

### Listing 13.16 Setting a response header in a result filter

```

Checks whether the action result returned
a 200 Ok result with a view model.          ResultFilterAttribute provides a useful
                                            base class you can override.

public class AddLastModifiedHeaderAttribute : ResultFilterAttribute
{
    public override void OnResultExecuting(
        ResultExecutingContext context)
    {
        if (context.Result is OkObjectResult result
            && result.Value is RecipeDetailViewModel detail)
        {
            var viewModelDate = detail.LastModified;
            context.HttpContext.Response
                .GetTypedHeaders().LastModified = viewModelDate;
        }
    }
}

```

**Checks whether the view model type is RecipeDetailViewModel ...**

**You could also override the Executed method, but the response would already be sent by then.**

**... 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, fetch the `LastModified` field from the view model and add a `Last-Modified` header to the response.

**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.

#### Running result filters after short-circuits with `IAlwaysRunResultFilter`

Result filters are designed to "wrap" the execution of an `IActionResult` returned by an action method or action filter so that you can customize how the action result is executed. However, this customization doesn't apply to `IActionResults` set when you short-circuit the filter pipeline by setting `context.Result` in an authorization filter, resource filter, or exception filter.

That's often not a problem, as many result filters are designed to handle "happy path" transformations. But sometimes you want to make sure a transformation is *always* applied to an `IActionResult`, regardless of whether it was returned by an action method or a short-circuiting filter.

For those cases, you can implement `IAlwaysRunResultFilter` or `IAsyncAlwaysRunResultFilter`. These interfaces extend (and are identical) to the standard result filter interfaces, so they run just like normal result filters in the filter pipeline. But these interfaces mark the filter to *also* run after an authorization filter, resource filter, or exception filter short-circuits the pipeline, where standard result filters won't run.

You can use `IAlwaysRunResultFilter` to ensure that certain action results are always updated. For example, the documentation shows how to use an `IAlwaysRunResultFilter` to convert a `415 StatusCodeResult` into a `422 StatusCodeResult`, regardless of the source of the action result. See the “`IAlwaysRunResultFilter` and `IAsyncAlwaysRunResultFilter`” section of Microsoft’s “Filters in ASP.NET Core” documentation: <http://mng.bz/JDo0>.

We've finished simplifying the `RecipeApiController` now. By extracting various pieces of functionality to filters, the original controller in listing 13.8 has been simplified to the version in listing 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 to 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 fulfill similar roles.

As is the case for 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**

```

Fetches an instance of RecipeService
from the DI container
public class PageEnsureRecipeExistsAttribute : Attribute, IPageFilter
{
    public void OnPageHandlerSelected(
        PageHandlerSelectedContext context)
    {}

    public void OnPageHandlerExecuting(
        PageHandlerExecutingContext context)
    {
        var service = (RecipeService) context.HttpContext
            .RequestServices.GetService(typeof(RecipeService));
        var recipeId = (int) context.HandlerArguments["id"];
        if (!service.DoesRecipeExist(recipeId))
        {
            context.Result = new NotFoundResult();
        }
    }

    public void OnPageHandlerExecuted(
        PageHandlerExecutedContext context)
    {}
}

```

**Implements IPageFilter and as an attribute so you can decorate the Razor Page PageModel.**

**Executed after handler selection, before model binding—not used in this example**

**Executed after model binding and validation, before page handler execution**

**Checks whether a Recipe entity with the given RecipId exists**

**If it doesn't exist, returns a 404 Not Found result and short-circuits the pipeline**

**Executed after page handler execution (or short-circuiting)—not used in this example**

**Retrieves the id parameter that will be passed to the page handler method when it executes**

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, 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, we'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 be 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 difficulty 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.

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.

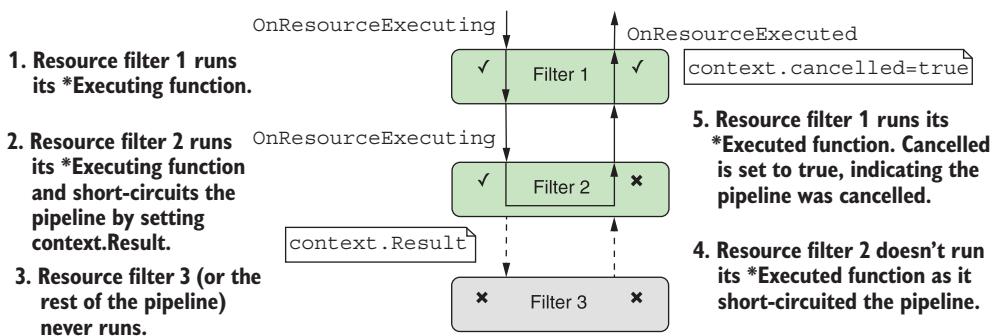


Figure 13.7 The effect of short-circuiting a resource filter on other resource filters in that stage. Later filters in the stage won't run at all, but earlier filters run their `OnResourceExecuted` function.

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 <code>context.Result</code>	Only runs <code>IAlwaysRunResultFilters</code> .
Resource filters	Set <code>context.Result</code>	Resource-filter <code>*Executed</code> functions from earlier filters run with <code>context.Cancelled = true</code> . Runs <code>IAlwaysRunResultFilters</code> before executing the <code>IActionResult</code> .
Action filters	Set <code>context.Result</code>	Only bypasses action method execution. Action filters earlier in the pipeline run their <code>*Executed</code> methods with <code>context.Cancelled = true</code> , then result filters, result execution, and resource filters' <code>*Executed</code> methods all run as normal.
Page filters	Set <code>context.Result</code> in <code>OnPageHandlerSelected</code>	Only bypasses page handler execution. Page filters earlier in the pipeline run their <code>*Executed</code> methods with <code>context.Cancelled = true</code> , then result filters, result execution, and resource filters' <code>*Executed</code> methods all run as normal.
Exception filters	Set <code>context.Result</code> and <code>Exception.Handled = true</code>	All resource-filter <code>*Executed</code> functions run. Runs <code>IAlwaysRunResultFilters</code> before executing the <code>IActionResult</code> .
Result filters	Set <code>context.Cancelled = true</code>	Result filters earlier in the pipeline run their <code>*Executed</code> functions with <code>context.Cancelled = true</code> . All resource-filter <code>*Executed</code> 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 {  
    private readonly RecipeService _service;  
    public EnsureRecipeExistsFilter(RecipeService service)  
    {  
        _service = service;  
    }  
}
```

Doesn't derive from an Attribute class

RecipeService is injected into the constructor.

```

public void OnActionExecuting(ActionExecutingContext context)
{
    var recipeId = (int) context.ActionArguments["id"];
    if (!_service.DoesRecipeExist(recipeId))
    {
        context.Result = new NotFoundResult();
    }
}

public void OnActionExecuted(ActionExecutedContext context) { }

The rest of
the method
remains the
same.

You must implement the
Executed action to satisfy
the interface.

→ public class EnsureRecipeExistsAttribute : TypeFilterAttribute
{
    public EnsureRecipeExistsAttribute()
        : base(typeof(EnsureRecipeExistsFilter)) {}
}

Derives from TypeFilter, which is used to
fill dependencies using the DI container

Passes the type
EnsureRecipeExistsFilt
er 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.

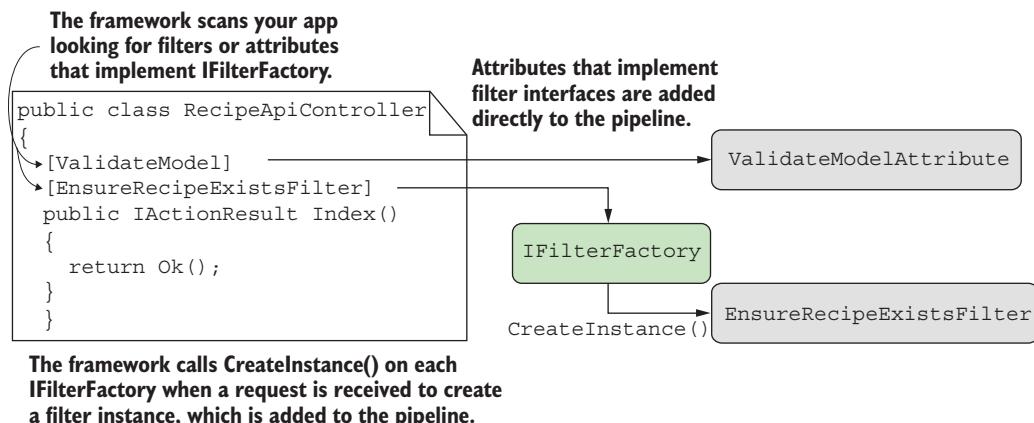


Figure 13.8 The framework scans your app on startup to find both filters and attributes that implement `IFilterFactory`. At runtime, the framework calls `CreateInstance()` to get an instance of the filter.

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.

## 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 an `IActionResult` 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 they 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.
- Result filters aren't executed when you short-circuit the pipeline using authorization, resource, or exception filters. You can ensure result filters also run for these short-circuit cases by implementing a result filter as `IAlwaysRunResultFilter` or `IAsyncAlwaysRunResultFilter`.
- 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, and on others you might see suggested articles. On an e-commerce app, you’d be able to place orders 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 how the pieces fit together, but you'll often need to add users and authentication to an existing app. In section 14.4 you'll see 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 how to provide them with permissions 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 how you can 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'll 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.

### 14.1.1 Understanding users and claims in ASP.NET Core

The concept of a user is baked in to ASP.NET Core. 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 `ClaimsPrincipal`, which has a collection of *claims* associated with it, as shown in figure 14.1.

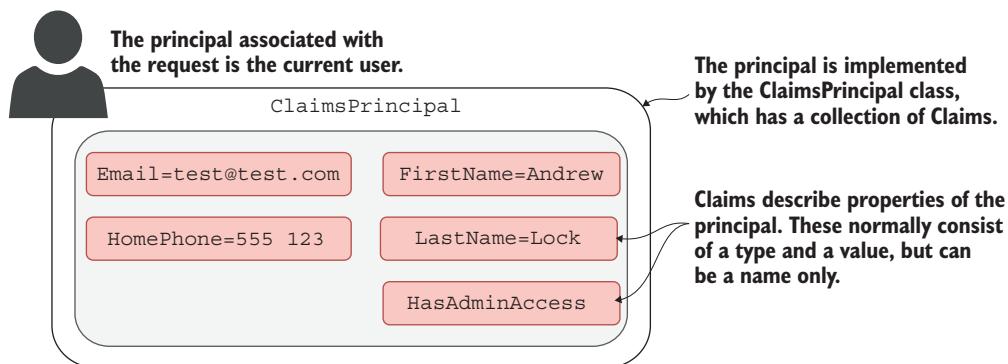


Figure 14.1 The principal is the current user, implemented as `ClaimsPrincipal`. It contains a collection of `Claims` that describe the user.

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; it 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:

- 1 The client sends an identifier and a secret to the app, which identify the current user. For example, you could send an email address (identifier) and a password (secret).
- 2 The app verifies that the identifier corresponds to a user known by the app and that the corresponding secret is correct.
- 3 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.

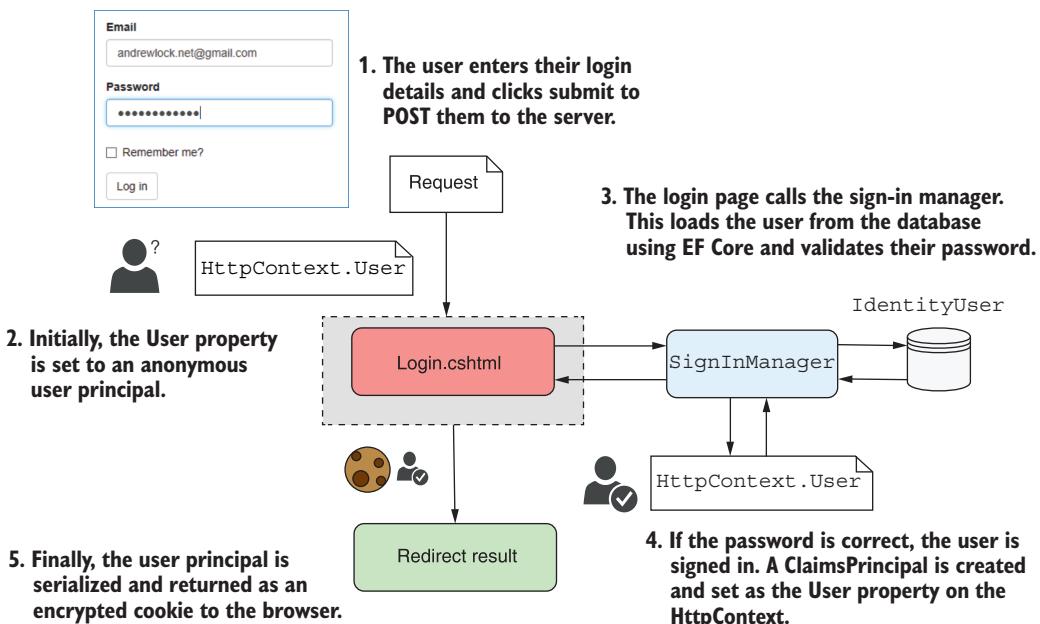


Figure 14.2 Signing in to an ASP.NET Core application. `SignInManager` is responsible for setting `HttpContext.User` to the new principal and serializing the principal to the encrypted cookie.

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.cshtml` 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 address. 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`.

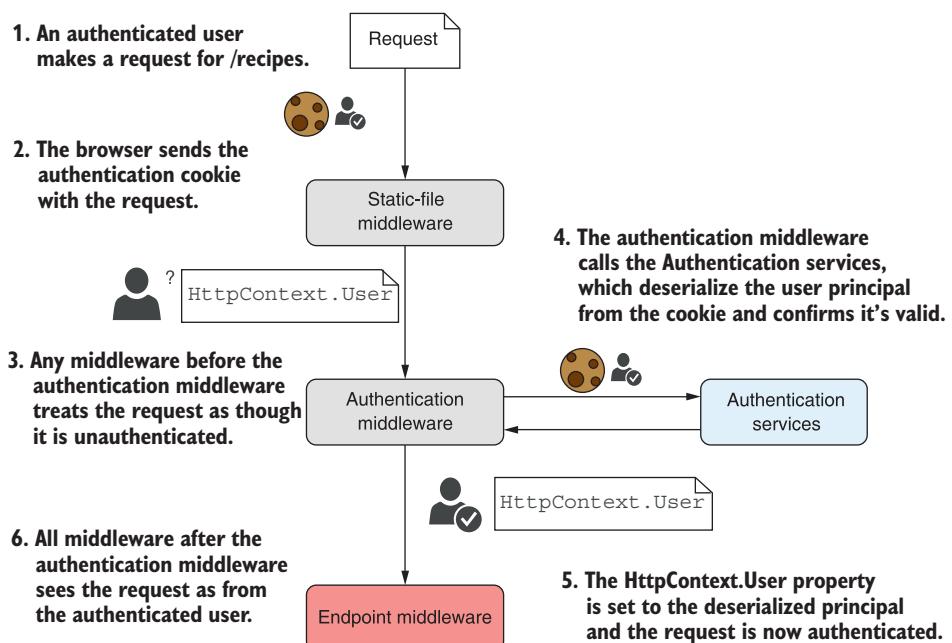


Figure 14.3 A subsequent request after signing in to an application. The cookie sent with the request contains the user principal, which is validated and used to authenticate the request.

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 the authentication services, which reads the cookie from the request, decrypts it, and deserializes it to obtain the `ClaimsPrincipal` created when the user logged in.

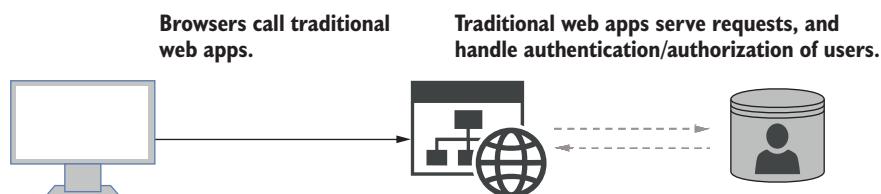
`AuthenticationMiddleware` sets the `HttpContext.User` principal to the new, authenticated principal. All subsequent middleware will now know the user principal for the request and can adjust their behavior accordingly (for example, displaying the user's name on the home page, or restricting access to 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'll take a look at authentication for Web API applications, used by client-side and mobile apps, and at how the authentication system changes for those scenarios.

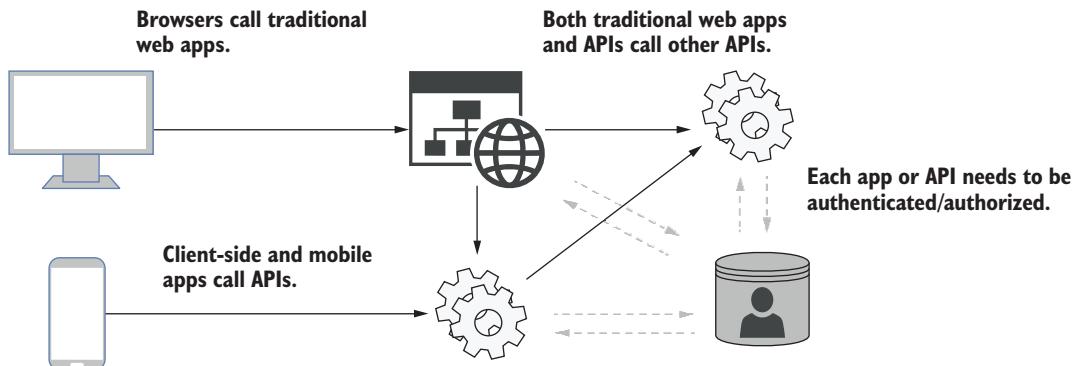
### 14.1.3 Authentication for APIs and distributed applications

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.



**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 toward 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.



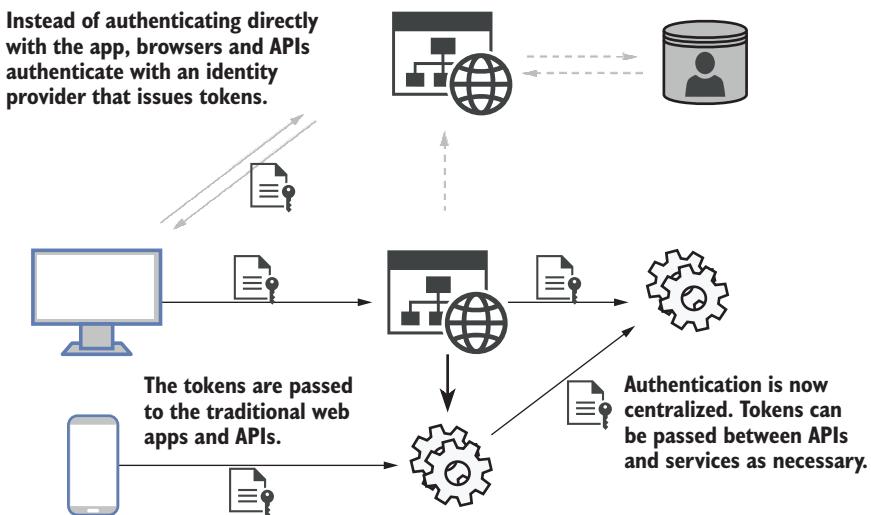
**Figure 14.5** Modern applications typically need to expose Web APIs for mobile and client-side apps, as well as potentially calling APIs on the backend. When all of these services need to authenticate and manage users, this becomes logically complicated.

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 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.



**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.

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 IdentityServer (<https://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 provide 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 Microsoft's "Authentication and authorization for SPAs" documentation at <http://mng.bz/w9Mq> and the IdentityServer docs: <https://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?

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 that 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, 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 it 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, 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 email messages.
Password validation and rules.	Customizing claims for users (adding new claims).

**Table 14.1 Which services are and aren't handled by ASP.NET Core Identity (continued)**

Managed by ASP.NET Core Identity	Requires implementing by the developer
Handling user account lockout (to prevent brute-force attacks). 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).	Configuring third-party identity providers.

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 you need to customize the HTML generated, 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, 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 .NET 5.0 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 box, shown in figure 14.7.

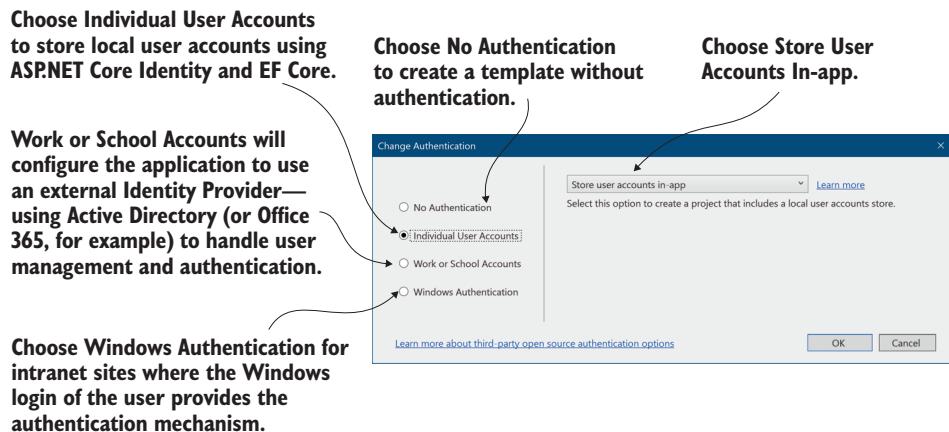


Figure 14.7 Choosing the authentication mode of the new ASP.NET Core application template in VS 2019

- 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.

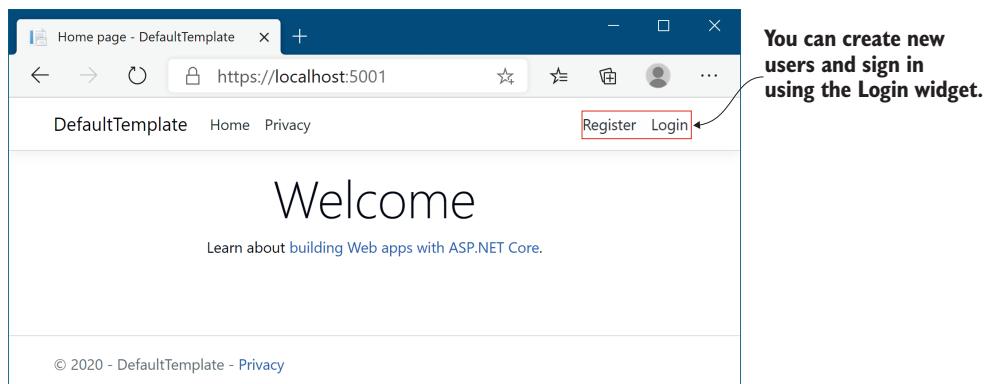


Figure 14.8 The default template with individual account authentication looks similar to the no authentication template, with the addition of a Login widget at the top right of the page.

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.

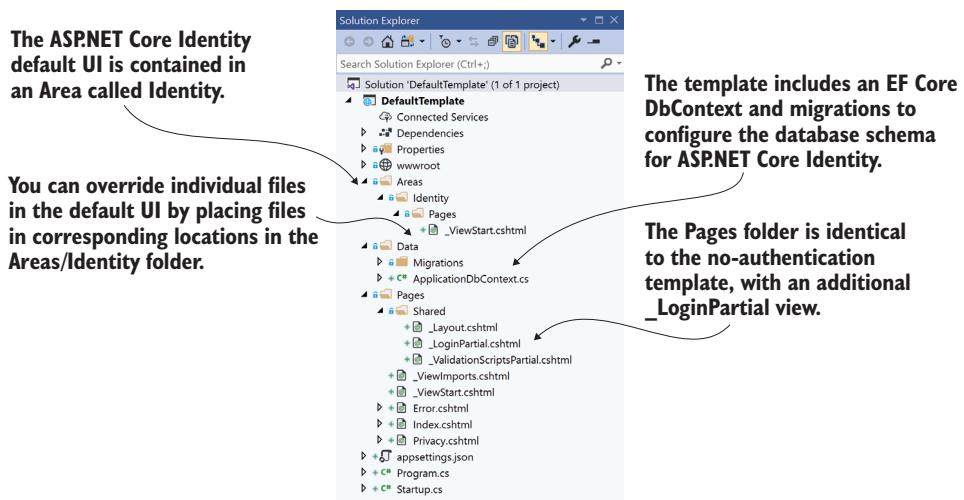


Figure 14.9 The project layout of the default template. Depending on your version of Visual Studio, the exact files may vary slightly.

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 subfolders in the main Pages folder instead. The one exception is the Identity UI, which uses a separate Identity area by default. For more details on areas, see Microsoft’s “Areas in ASP.NET Core” documentation: <http://mng.bz/7Vw9>.

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 to 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 it’s 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

Adds the Identity system, including the default UI, and configures the user type as IdentityUser

ASP.NET Core Identity uses EF Core, so it includes the standard EF Core configuration.

```
public void ConfigureServices(IServiceCollection services)  
{  
    services.AddDbContext<ApplicationDbContext>(options =>  
        options.UseSqlServer(  
            Configuration.GetConnectionString("DefaultConnection")));  
    services.AddDefaultIdentity<IdentityUser>(options =>
```

```

        options.SignIn.RequireConfirmedAccount = true);
    .AddEntityFrameworkStores<ApplicationContext>();
    services.AddRazorPages();
}

Configures Identity to store its data in EF Core
Requires users to confirm their accounts (typically by email) before they log in

```

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();           ← Middleware placed before UseAuthentication will see all requests as anonymous.
    app.UseRouting();              ← The routing middleware determines which page is requested based on the request URL.
    app.UseAuthentication();       ← UseAuthentication should be placed after UseRouting.
    app.UseAuthorization();        ← UseEndpoints should be last, after the user principal is set and authorization has been applied.
}

UseAuthorization should be placed after UseAuthentication so it can access the user principal.

```

If you don’t use this specific order of middleware, you can run into strange bugs where users aren’t authenticated correctly, or where 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 that 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 this command to update the database with the migrations:

```
dotnet ef database update
```

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.<sup>1</sup>

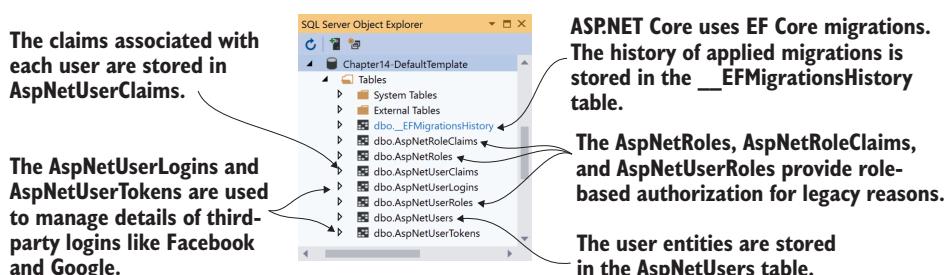


Figure 14.10 The database schema used by ASP.NET Core Identity

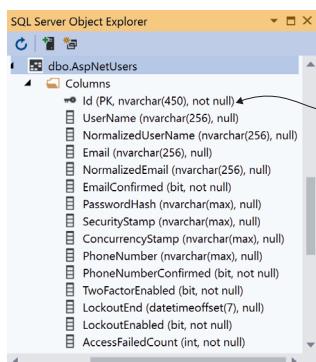
<sup>1</sup> If you're using MS SQL Server (or LocalDB), you can use the SQL Server Object Explorer in Visual Studio to browse tables and objects in your database. See Microsoft's "How to: Connect to a Database and Browse Existing Objects" for details: <http://mng.bz/mg8r>.

**TIP** If you see an error after running the `dotnet ef` command, ensure you have the .NET tool installed by following the instructions provided in section 12.3.1. Also ensure you run the command from the *project* folder, not the *solution* folder.

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.
- *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.



The screenshot shows the SQL Server Object Explorer interface. A tree view on the left lists 'dbo.AspNetUsers' under 'Tables'. A callout points to the 'Id' column, which is highlighted in blue and identified as a primary key (PK) with a data type of nvarchar(450) and a constraint of not null. Another callout provides detailed information about the 'Id' column: 'By default, ASP.NET Core Identity uses Guids for the user Id, stored as strings in the database.'

Below the table structure, another callout highlights the entire list of columns: 'The table contains all the necessary fields for authenticating a user, email and phone number confirmation, two-factor authentication, and account lockout.'

Column Name	Type	Properties
Id	nvarchar(max)	PK, not null
UserName	nvarchar(256)	
NormalizedUserName	nvarchar(256)	
Email	nvarchar(256)	
NormalizedEmail	nvarchar(256)	
EmailConfirmed	bit	not null
PasswordHash	nvarchar(max)	
SecurityStamp	nvarchar(max)	
ConcurrencyStamp	nvarchar(max)	
PhoneNumber	nvarchar(max)	
PhoneNumberConfirmed	bit	not null
TwoFactorEnabled	bit	not null
LockoutEnd	datetimeoffset(7)	not null
LockoutEnabled	bit	not null
AccessFailedCount	int	not null

**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.

**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 the “Change the primary key type” section of Microsoft’s “Identity model customization in ASP.NET Core” documentation: <http://mng.bz/5jdB>.

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'll 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: <http://mng.bz/Xd61>.

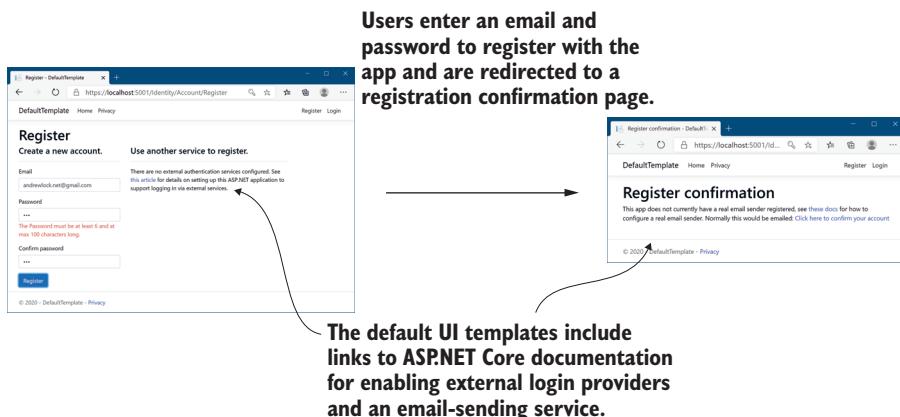
It's important to understand the difference between the `IdentityUser` entity (stored in the `AspNetUsers` table) and the `ClaimsPrincipal`, which is exposed on `HttpContext.User`. When a user first logs in, an `IdentityUser` is loaded from the database. This entity is combined with additional claims for the user from the `AspNetUserClaims` table to create a `ClaimsPrincipal`. It's this `ClaimsPrincipal` that 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 that they should confirm their email. No email service is enabled by default, as this is dependent of you configuring an external email service. You can read how to enable email sending in Microsoft's “Account confirmation and password recovery in ASP.NET Core” documentation at



**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.

<http://mng.bz/6gBo>. Once you configure this, users will automatically receive an email with a link to confirm their account.

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 ConfigureServices in Startup.cs

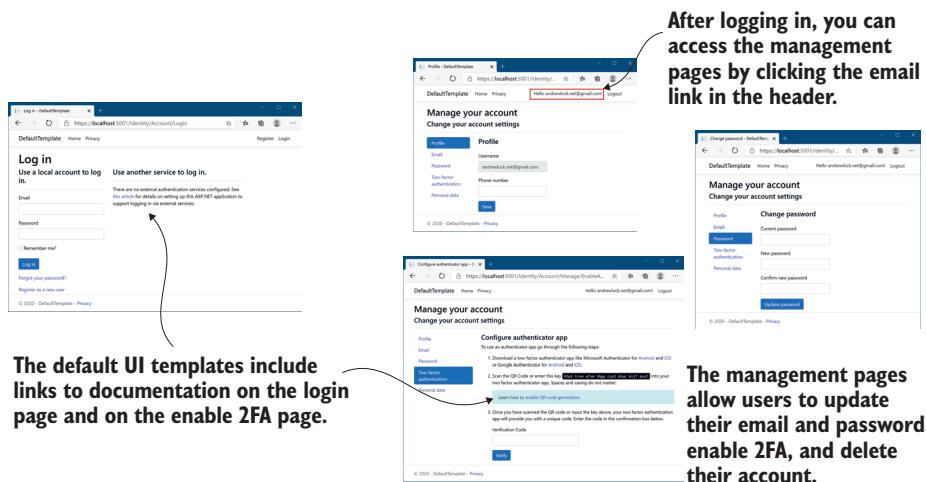
```
services.AddDefaultIdentity<IdentityUser>(options =>
{
    options.SignIn.RequireConfirmedAccount = true;           ← Require users to confirm
    options.Lockout.AllowedForNewUsers = true;              their account by email
    options.Password.RequiredLength = 12;                   before they can log in.
    options.Password.RequireNonAlphanumeric = false;
    options.Password.RequireDigit = false;
})
.AddEntityFrameworkStores<AppDbContext>();
```

Enables user-lockout, to prevent brute-force attacks against user passwords

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 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 it's one of the reasons you may need to customize the default UI templates, as you'll see in section 14.5.

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



**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).

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.<sup>2</sup>

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 Microsoft's "Account confirmation and password recovery in ASP.NET Core" documentation: <http://mng.bz/vzy7>.
- Add a QR code generator for the enable 2FA page, as described in Microsoft's "Enable QR Code generation for TOTP authenticator apps in ASP.NET Core" documentation: <http://mng.bz/4Zmw>.
- Customize the register and login 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 Microsoft's "Scaffold Identity in ASP.NET Core projects" documentation: <http://mng.bz/QmMG>.

<sup>2</sup> You can improve the 2FA authenticator page by enabling QR code generation, as described in Microsoft's "Enable QR Code generation for TOTP authenticator apps in ASP.NET Core" document: <http://mng.bz/nM5a>.

- 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 Microsoft's "Overview of ASP.NET Core authentication" at <http://mng.bz/XdGv> 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.

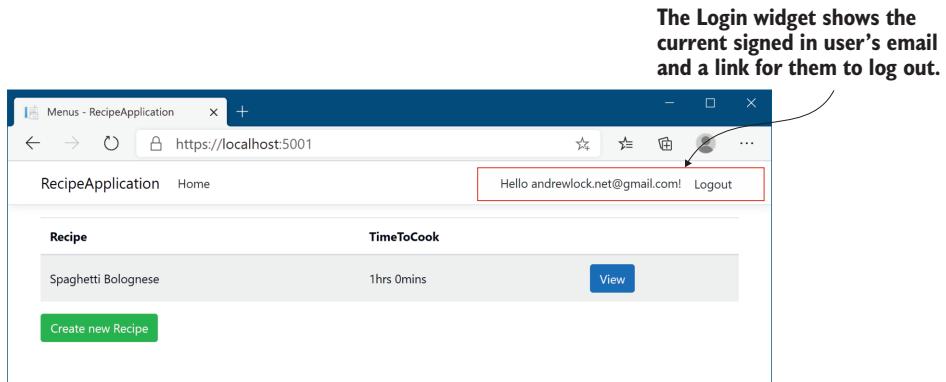


Figure 14.14 The recipe app after adding authentication, showing the login widget.

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 do the following:

- 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 file to include these two packages:

```
<PackageReference
    Include="Microsoft.AspNetCore.Identity.EntityFrameworkCore"
    Version="5.0.0" />
<PackageReference
    Include="Microsoft.AspNetCore.Identity.UI" Version="5.0.0" />
```

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 =>
        options.UseSqlServer(
            Configuration.GetConnectionString("DefaultConnection")));
    services.AddDefaultIdentity<ApplicationUser>(options =>
        options.SignIn.RequireConfirmedAccount = true)
        .AddEntityFrameworkStores<AppDbContext>(); ←
    services.AddRazorPages();
    services.AddScoped<RecipeService>();
}
```

The existing service configuration is unchanged.

Adds the Identity services to the DI container and uses a custom user type, ApplicationUser

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 14.5, 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();           ← StaticFileMiddleware will never see requests  
as authenticated, even after you sign in.

    app.UseRouting();
    app.UseAuthentication();        ← Adds AuthenticationMiddleware  
after UseRouting() and before  
UseAuthorization
    app.UseAuthorization();

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

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 line:

```
public class ApplicationUser : IdentityUser { }
```

It's not strictly necessary to create a custom user type in this case (for example, the default templates use the raw `IdentityUser`), but I find it's easier to add the derived type now rather than try to retrofit it later if you need to add extra properties to your user 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

**Updates to inherit from the Identity context, instead of directly from DbContext**

```
public class AppDbContext : IdentityDbContext<ApplicationUser>
{
    public AppDbContext(DbContextOptions<AppDbContext> options)
        : base(options)
    { }

    public DbSet<Recipe> Recipes { get; set; }
}
```

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;
@inject SignInManager< ApplicationUser > SignInManager
@inject UserManager< ApplicationUser > UserManager




@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>

```

Update to your project's namespace  
that contains ApplicationUser
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'll 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 are 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.<sup>3</sup>

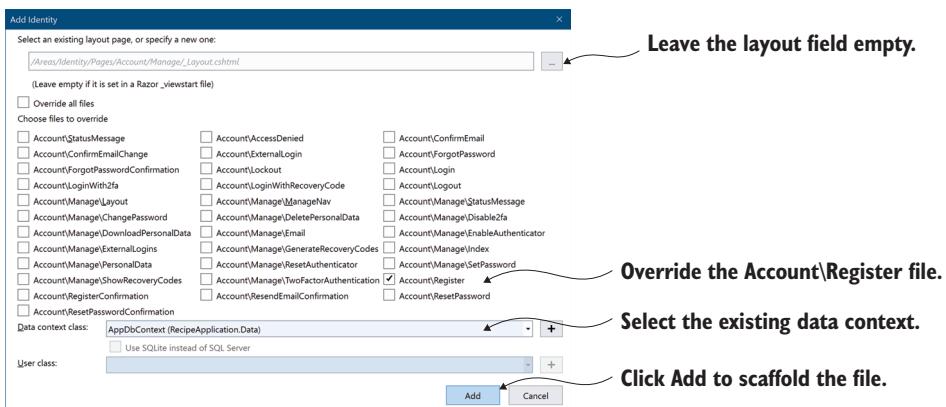
- 1 Add the `Microsoft.VisualStudio.Web.CodeGeneration.Design` and `Microsoft.EntityFrameworkCore.Tools` NuGet packages to your project file, if they're not already added. Visual Studio uses these packages to scaffold your application correctly, and without them you may get an error running the scaffolder.

```
<PackageReference Version="5.0.0"
    Include="Microsoft.VisualStudio.Web.CodeGeneration.Design" />
<PackageReference Version="5.0.0"
    Include="Microsoft.EntityFrameworkCore.Tools" />
```

- 2 Ensure your project builds—if it doesn't build, the scaffolder will fail before adding your new pages.

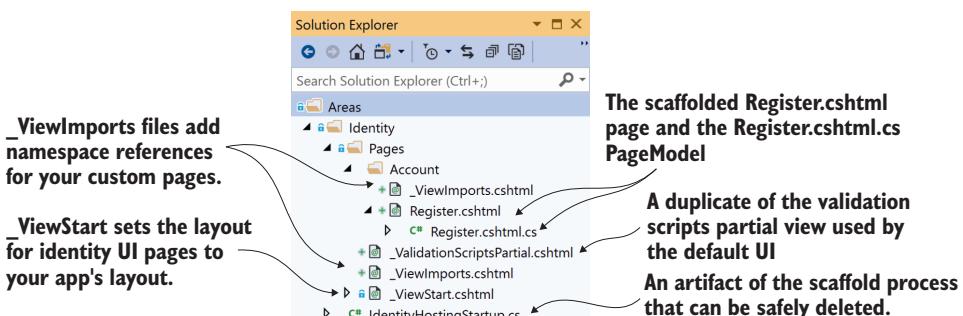
<sup>3</sup> Install the necessary .NET CLI tools and packages as described in Microsoft's "Scaffold Identity in ASP.NET Core projects" documentation: <http://mng.bz/yYGB>. Then run `dotnet aspnet-codegenerator identity -dc RecipeApplication.Data.AppDbContext --files "Account.Register"`.

- 3 Right-click your project and choose Add > New Scaffolded Item.
- 4 In the selection dialog box, choose Identity from the category, and click Add.
- 5 In the Add Identity dialog box, 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.



**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.



**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 that 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 the 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.cshtml 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'll show you the first approach, automatically adding new claims to a user when they're created. The latter approach is 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. Microsoft's "Add, download, and delete custom user data to Identity in an ASP.NET Core project" documentation describes the steps required to achieve that: <http://mng.bz/aoe7>.

Let's say you want to add a new `Claim` to a user, called `FullName`. A typical approach would be as follows:

- 1 Scaffold the `Register.cshtml` Razor Page, as you did in section 14.5.
- 2 Add a "Name" field to the `InputModel` in the `Register.cshtml.cs` `PageModel`.
- 3 Add a "Name" input field to the `Register.cshtml` Razor view template.
- 4 Create the new  `ApplicationUser` entity as before in the `OnPost()` page handler by calling `CreateAsync` on `UserManager< ApplicationUser >`.
- 5 Add a new `Claim` to the user by calling `UserManager.AddClaimAsync()`.
- 6 Continue the method as before, sending a confirmation email or signing the user in if email confirmation is not required.

Steps 1–3 are fairly self-explanatory and just require updating the existing templates with the new field. Steps 4–6 all take place in Register.cshtml.cs in the `OnPost()` page handler, which is summarized in the following listing. In practice, the page handler has more error checking and boilerplate; our focus here is on the additional lines that add the extra `Claim` to the `ApplicationUser`.

**Listing 14.8 Adding a custom claim to a new user in the Register.cshtml.cs page**

```

public async Task<IActionResult> OnPostAsync(string returnUrl = null)
{
    if (ModelState.IsValid)
    {
        var user = new ApplicationUser {
            UserName = Input.Email, Email = Input.Email };
        var result = await _userManager.CreateAsync(
            user, Input.Password);
        if (result.Succeeded)
        {
            var claim = new Claim("FullName", Input.Name);
            await _userManager.AddClaimAsync(user, claim);

            var code = await _userManager
                .GenerateEmailConfirmationTokenAsync(user);
            await _emailSender.SendEmailAsync(
                Input.Email, "Confirm your email", code );
            await _signInManager.SignInAsync(user);
            return LocalRedirect(returnUrl);
        }
        foreach (var error in result.Errors)
        {
            ModelState.AddModelError(
                string.Empty, error.Description);
        }
    }
    return Page();
}

```

**TIP** Listing 14.8 shows how you can add extra claims at registration time, but you will often need to add additional data later, such as permission-related claims or other information. You will need to create additional endpoints and pages for adding this data, securing the pages as appropriate (so that users can't update their own permissions, for example).

This is all that's required to *add* the new claim, but you're not *using* it anywhere currently. What if you want to display it? Well, you've added a claim to the `ClaimsPrincipal`, which was assigned to the `HttpContext.User` property when you called `SignInAsync`. That means you can retrieve the claims anywhere you have access to the `ClaimsPrincipal`—including in your page handlers and in view templates. For example, you could display the user's `FullName` claim anywhere in a Razor template with the following statement:

```
@User.Claims.FirstOrDefault(x=>x.Type == "FullName")?.Value
```

This finds the first claim on the current user principal with a Type of "FullName" and prints the assigned value (or if the claim is not found, it prints nothing). The Identity system even includes a handy extension method that tidies up this LINQ expression (found in the `System.Security.Claims` namespace):

```
@User.FindFirstValue("FullName")
```

With that last tidbit, we've reached the end of this chapter on ASP.NET Core Identity. I hope you've come to appreciate the amount of effort using Identity can save you, especially when you make use of the default Identity UI package.

Adding user accounts and authentication to an app is typically the first step to customizing your app further. Once you have authentication, you can have authorization, which lets you lock down certain actions in your app, based on the current user. In the next chapter you'll learn about the ASP.NET Core authorization system and how you can use it to customize your apps; in particular, the recipe application, which is coming along nicely!

## Summary

- Authentication is the process of determining who you are, and authorization is the process of determining what you're allowed to do. You need to authenticate users before you can apply authorization.
- Every request in ASP.NET Core is associated with a user, also known as a principal. By default, without authentication, this is an anonymous user. You can use the claims principal to behave differently depending on who made a request.
- The current principal for a request is exposed on `HttpContext.User`. You can access this value from your Razor Pages and views to find out properties of the user such as their, ID, name, or email.
- Every user has a collection of claims. These claims are single pieces of information about the user. Claims could be properties of the physical user, such as `Name` and `Email`, or they could be related to things the user has, such as `HasAdminAccess` or `IsVipCustomer`.
- Earlier versions of ASP.NET used roles instead of claims. You can still use roles if you need to, but you should use claims where possible.
- Authentication in ASP.NET Core is provided by `AuthenticationMiddleware` and a number of authentication services. These services are responsible for setting the current principal when a user logs in, saving it to a cookie, and loading the principal from the cookie on subsequent requests.
- The `AuthenticationMiddleware` is added by calling `UseAuthentication()` in your middleware pipeline. This must be placed after the call to `UseRouting()` and before `UseAuthorization()` and `UseEndpoints()`.
- ASP.NET Core includes support for consuming bearer tokens for authenticating API calls and includes helper libraries for configuring `IdentityServer`. For

more details see Microsoft's "Authentication and authorization for SPAs" documentation: <http://mng.bz/go0V>.

- ASP.NET Core Identity handles low-level services needed for storing users in a database, ensuring their passwords are stored safely, and for logging users in and out. You must provide the UI for the functionality yourself and wire it up to the Identity subsystem.
- The Microsoft.AspNetCore.Identity.UI package provides a default UI for the Identity system and includes email confirmation, 2FA, and external login provider support. You need to do some additional configuration to enable these features.
- The default template for Web Application with Individual Account Authentication uses ASP.NET Core Identity to store users in the database with EF Core. It includes all the boilerplate code required to wire the UI up to the Identity system.
- You can use the `UserManager<T>` class to create new user accounts, load them from the database, and change their passwords. `SignInManager<T>` is used to sign a user in and out by assigning the principal for the request and by setting an authentication cookie. The default UI uses these classes for you, to facilitate user registration and login.
- You can update an EF Core `DbContext` to support Identity by deriving from `IdentityDbContext<TUser>`, where `TUser` is a class that derives from `IdentityUser`.
- You can add additional claims to a user using the `UserManager<TUser>.AddClaimAsync(TUser user, Claim claim)` method. These claims are added to the `HttpContext.User` object when the user logs in to your app.
- Claims consist of a type and a value. Both values are strings. You can use standard values for types exposed on the `ClaimTypes` class, such as `ClaimTypes.GivenName` and `ClaimTypes.FirstName`, or you can use a custom string, such as "FullName".

# 15 Authorization: Securing your application

## This chapter covers

- Using authorization to control who can use your app
- Using claims-based authorization with policies
- Creating custom policies to handle complex requirements
- Authorizing a request depending upon the resource being accessed
- Hiding elements from a Razor template that the user is unauthorized to access

In chapter 14 I showed you how to add users to an ASP.NET Core application by adding *authentication*. With authentication, users can register and log in to your app using an email address and password. Whenever you add authentication to an app, you inevitably find you want to be able to restrict what some users can do. The process of determining whether a user can perform a given action on your app is called *authorization*.

On an e-commerce site, for example, you may have admin users who are allowed to add new products and change prices, sales users who are allowed to view completed orders, and customer users who are only allowed to place orders and buy products.

In this chapter I'll show you how to use authorization in an app to control what your users can do. In section 15.1 I'll introduce authorization and put it in the context of a real-life scenario you've probably experienced: an airport. I'll describe the sequence of events, from checking in, to passing through security, to entering an airport lounge, and you'll see how these relate to the authorization concepts in this chapter.

In section 15.2 I'll show how authorization fits into an ASP.NET Core web application and how it relates to the `ClaimsPrincipal` class you saw in the previous chapter. You'll see how to enforce the simplest level of authorization in an ASP.NET Core app, ensuring that only authenticated users can execute a Razor Page or MVC action.

We'll extend that approach in section 15.3 by adding in the concept of policies. These let you set specific requirements for a given authenticated user, requiring that they have specific pieces of information in order to execute an action or Razor Page.

You'll use policies extensively in the ASP.NET Core authorization system, so in section 15.4 we'll explore how to handle more complex scenarios. You'll learn about authorization requirements and handlers, and how you can combine them to create specific policies that you can apply to your Razor Pages and actions.

Sometimes, whether a user is authorized depends on which resource or document they're attempting to access. A resource is anything that you're trying to protect, so it could be a document or a post in a social media app. For example, you may allow users to create documents, or to read documents from other users, but only to edit documents that they created themselves. This type of authorization, where you need the details of the document to determine if the user is authorized, is called resource-based authorization, and it's the focus of section 15.5.

In the final section of this chapter, I'll show how you can extend the resource-based authorization approach to your Razor view templates. This lets you modify the UI to hide elements that users aren't authorized to interact with. In particular, you'll see how to hide the Edit button when a user isn't authorized to edit the entity.

We'll start by looking more closely at the concept of authorization, how it differs from authentication, and how it relates to real-life concepts you might see in an airport.

## 15.1 Introduction to authorization

In this section I provide an introduction to authorization and discuss how it compares to authentication. I use the real-life example of an airport as a case study to illustrate how claims-based authorization works.

For people who are new to web apps and security, authentication and authorization can sometimes be a little daunting. It certainly doesn't help that the words look so similar! The two concepts are often used together, but they're definitely distinct:

- *Authentication*—The process of determining who made a request
- *Authorization*—The process of determining whether the requested action is allowed

Typically, authentication occurs first, so that you know who is making a request to your app. For traditional web apps, your app authenticates a request by checking the encrypted cookie that was set when the user logged in (as you saw in the previous chapter). Web APIs typically use a header instead of a cookie for authentication, but the process is the same.

Once a request is authenticated and you know who is making the request, you can determine whether they're allowed to execute an action on your server. This process is called authorization and is the focus of this chapter.

Before we dive into code and start looking at authorization in ASP.NET Core, I'll put these concepts into a real-life scenario you're hopefully familiar with: checking in at an airport. To enter an airport and board a plane, you must pass through several steps: an initial step to prove who you are (authentication); and subsequent steps that check whether you're allowed to proceed (authorization). In simplified form, these might look like this:

- 1 Show your passport at the check-in desk. Receive a boarding pass.
- 2 Show your boarding pass to enter security. Pass through security.
- 3 Show your frequent flyer card to enter the airline lounge. Enter the lounge.
- 4 Show your boarding pass to board the flight. Enter the airplane.

Obviously, these steps, also shown in figure 15.1, will vary somewhat in real life (I don't have a frequent flyer card!), but we'll go with them for now. Let's explore each step a little further.

When you arrive at the airport, the first thing you do is go to the check-in counter. Here, you can purchase a plane ticket, but to do so, you need to prove who you are by providing a passport; you *authenticate* yourself. If you've forgotten your passport, you can't authenticate, and you can't go any further.

Once you've purchased your ticket, you're issued a boarding pass, which says which flight you're on. We'll assume it also includes a `BoardingPassNumber`. You can think of this number as an additional *claim* associated with your identity.

**DEFINITION** A *claim* is a piece of information about a user that consists of a *type* and an optional *value*.

The next step is security. The security guards will ask you to present your boarding pass for inspection, which they'll use to check that you have a flight and so are allowed deeper into the airport. This is an authorization process: you must have the required claim (a `BoardingPassNumber`) to proceed.

If you don't have a valid `BoardingPassNumber`, there are two possibilities for what happens next:

- *If you haven't yet purchased a ticket*—You'll be directed back to the check-in desk, where you can authenticate and purchase a ticket. At that point, you can try to enter security again.

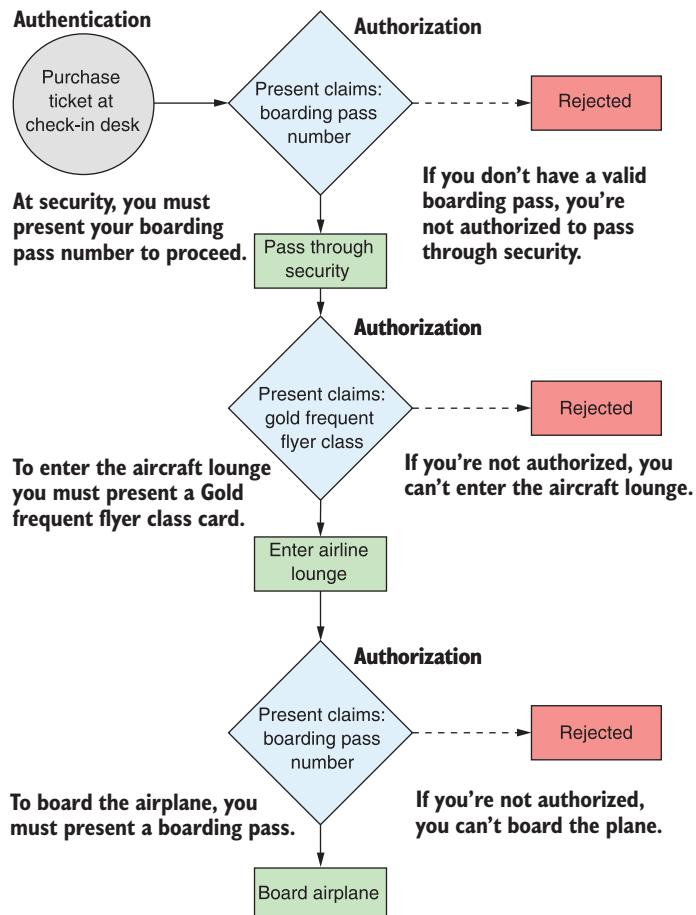


Figure 15.1 When boarding a plane at an airport, you pass through several authorization steps. At each authorization step, you must present a claim in the form of a boarding pass or a frequent flyer card. If you're not authorized, access will be denied.

- *If you have an invalid ticket*—You won't be allowed through security, and there's nothing else you can do. If, for example, you show up with a boarding pass a week late for your flight, they probably won't let you through. (Ask me how I know!)

Once you're through security, you need to wait for your flight to start boarding, but unfortunately there aren't any seats free. Typical! Luckily, you're a regular flyer, and you've notched up enough miles to achieve a Gold frequent flyer status, so you can use the airline lounge.

You head to the lounge, where you’re asked to present your Gold Frequent Flyer card to the attendant, and they let you in. This is another example of authorization. You must have a `FrequentFlyerClass` claim with a value of `Gold` to proceed.

**NOTE** You’ve used authorization twice so far in this scenario. Each time, you presented a claim to proceed. In the first case, the presence of any `BoardingPassNumber` was sufficient, whereas for the `FrequentFlyerClass` claim, you needed the specific value of `Gold`.

When you’re boarding the airplane, you have one final authorization step, in which you must present the `BoardingPassNumber` claim again. You presented this claim earlier, but boarding the aircraft is a distinct action from entering security, so you have to present it again.

This whole scenario has lots of parallels with requests to a web app:

- Both processes start with authentication.
- You have to prove *who* you are in order to retrieve the *claims* you need for authorization.
- You use authorization to protect sensitive actions like entering security and the airline lounge.

I’ll reuse this airport scenario throughout the chapter to build a simple web application that simulates the steps you take in an airport. We’ve covered the concept of authorization in general, so in the next section we’ll look at how authorization works in ASP.NET Core. We’ll start with the most basic level of authorization, ensuring only authenticated users can execute an action, and look at what happens when you try to execute such an action.

## 15.2 Authorization in ASP.NET Core

In this section you’ll see how the authorization principles described in the previous section apply to an ASP.NET Core application. You’ll learn about the role of the `[Authorize]` attribute and `AuthorizationMiddleware` in authorizing requests to Razor Pages and MVC actions. Finally, you’ll learn about the process of preventing unauthenticated users from executing endpoints, and what happens when users are unauthorized.

The ASP.NET Core framework has authorization built in, so you can use it anywhere in your app, but it’s most common in ASP.NET Core 5.0 to apply authorization via the `AuthorizationMiddleware`. The `AuthorizationMiddleware` should be placed *after* both the routing middleware and the authentication middleware, but *before* the endpoint middleware, as shown in figure 15.2.

**NOTE** In ASP.NET Core, an *endpoint* refers to the handler selected by the routing middleware, which will generate a response when executed. It is typically a Razor Page or a Web API action method.

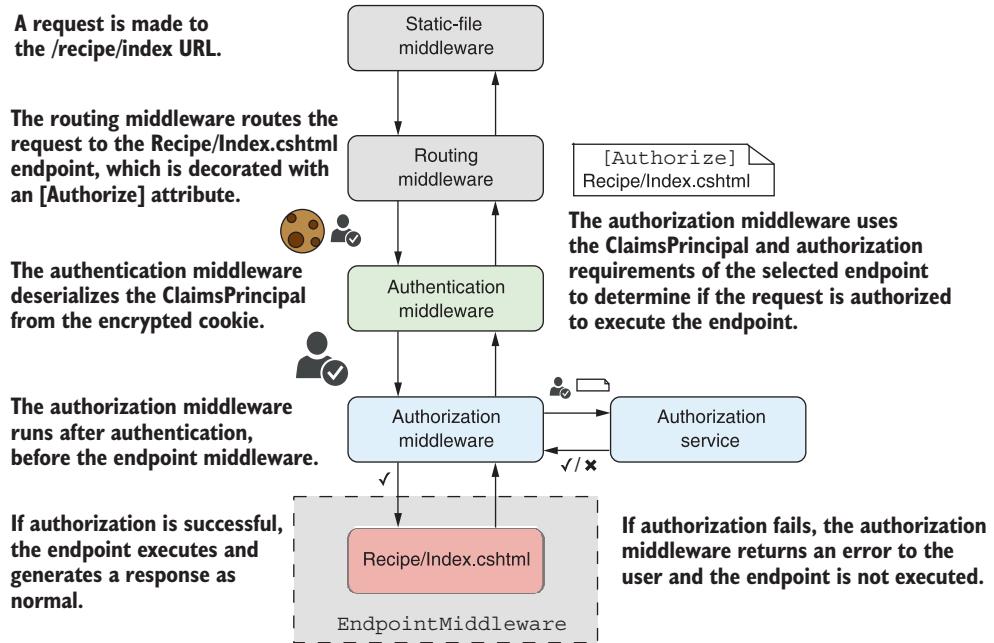


Figure 15.2 Authorization occurs after an endpoint has been selected and after the request is authenticated, but before the action method or Razor Page endpoint is executed.

With this configuration, the `RoutingMiddleware` selects an endpoint to execute based on the request's URL, such as a Razor Page, as you saw in chapter 5. Metadata about the selected endpoint is available to all middleware that occurs after the `routing middleware`. This metadata includes details about any authorization requirements for the endpoint, and it's typically attached by decorating an action or Razor Page with an `[Authorize]` attribute.

The `AuthenticationMiddleware` deserializes the encrypted cookie (or bearer token for APIs) associated with the request to create a `ClaimsPrincipal`. This object is set as the `HttpContext.User` for the request, so all subsequent middleware can access this value. It contains all the `Claims` that were added to the cookie when the user authenticated.

Now we come to the `AuthorizationMiddleware`. This middleware checks if the selected endpoint has any authorization requirements, based on the metadata provided by the `RoutingMiddleware`. If the endpoint has authorization requirements, the `AuthorizationMiddleware` uses the `HttpContext.User` to determine if the current request is authorized to execute the endpoint.

If the request is authorized, the next middleware in the pipeline executes as normal. If the request is not authorized, the `AuthorizationMiddleware` short-circuits the middleware pipeline, and the endpoint middleware is never executed.

**NOTE** The order of middleware in your pipeline is very important. The call to `UseAuthorization()` must come after `UseRouting()` and `UseAuthentication()`, but before `UseEndpoints()`.

### Changes to authorization in ASP.NET Core 3.0

The authorization system changed significantly in ASP.NET Core 3.0. Prior to this release, the `AuthorizationMiddleware` did not exist. Instead, the `[Authorize]` attribute executed the authorization logic as part of the MVC filter pipeline.

In practice, from the point of view of using authorization in your actions and Razor Pages, there is no real difference from a developer's point of view. Why change it then?

The new design, using the `AuthorizationMiddleware` in conjunction with endpoint routing (introduced at the same time), enables additional scenarios. The changes make it easier to apply authorization to non-MVC/Razor Page endpoints. You'll see how to create these types of endpoints in chapter 19. You can also read more about the authorization changes in the "Authorization" section of Microsoft's "Migrate from ASP.NET Core 2.2 to 3.0" documentation: <http://mng.bz/1rvj>.

The `AuthorizationMiddleware` is responsible for applying authorization requirements and ensuring that only authorized users can execute protected endpoints. In section 15.2.1 you'll learn how to apply the simplest authorization requirement, and in section 15.2.2 you'll see how the framework responds when a user is not authorized to execute an endpoint.

#### 15.2.1 Preventing anonymous users from accessing your application

When you think about authorization, you typically think about checking whether a particular user has permission to execute an endpoint. In ASP.NET Core you normally achieve this by checking whether a user has a given claim.

There's an even more basic level of authorization we haven't considered yet—only allowing authenticated users to execute an endpoint. This is even simpler than the claims scenario (which we'll come to later) as there are only two possibilities:

- *The user is authenticated*—The action executes as normal.
- *The user is unauthenticated*—The user can't execute the endpoint.

You can achieve this basic level of authorization by using the `[Authorize]` attribute, which you saw in chapter 13 when we discussed authorization filters. You can apply this attribute to your actions and Razor Pages, as shown in the following listing, to restrict them to authenticated (logged-in) users only. If an unauthenticated user tries to execute an action or Razor Page protected with the `[Authorize]` attribute, they'll be redirected to the login page.

**Listing 15.1 Applying [Authorize] to an action**

```
public class RecipeApiController : ControllerBase
{
    public IActionResult List()
    {
        return Ok();
    }

    [Authorize]
    public IActionResult View()
    {
        return Ok();
    }
}
```

The code listing shows two actions: `List()` and `View()`. An annotation for `List()` states: "This action can be executed by anyone, even when not logged in." An annotation for `[Authorize]` above `View()` states: "Applies [Authorize] to individual actions, whole controllers, or Razor Pages". Another annotation for `View()` states: "This action can only be executed by authenticated users."

Applying the `[Authorize]` attribute to an endpoint attaches metadata to it, indicating only authenticated users may access the endpoint. As you saw in figure 15.2, this metadata is made available to the `AuthorizationMiddleware` when an endpoint is selected by the `RoutingMiddleware`.

You can apply the `[Authorize]` attribute at the action scope, controller scope, Razor Page scope, or globally, as you saw in chapter 13. Any action or Razor Page that has the `[Authorize]` attribute applied in this way can be executed only by an authenticated user. Unauthenticated users will be redirected to the login page.

**TIP** There are several different ways to apply the `[Authorize]` attribute globally. You can read about the different options, and when to choose which option, on my blog: <http://mng.bz/opQp>.

Sometimes, especially when you apply the `[Authorize]` attribute globally, you might need to poke holes in this authorization requirement. If you apply the `[Authorize]` attribute globally, then any unauthenticated request will be redirected to the login page for your app. But if the `[Authorize]` attribute is global, then when the login page tries to load, you'll be unauthenticated and redirected to the login page again. And now you're stuck in an infinite redirect loop.

To get around this, you can designate specific endpoints to ignore the `[Authorize]` attribute by applying the `[AllowAnonymous]` attribute to an action or Razor Page, as shown next. This allows unauthenticated users to execute the action, so you can avoid the redirect loop that would otherwise result.

**Listing 15.2 Applying [AllowAnonymous] to allow unauthenticated access**

```
[Authorize]
public class AccountController : ControllerBase
{
    public IActionResult ManageAccount()
    {
        return Ok();
    }
}
```

The code listing shows one action: `ManageAccount()`. An annotation for `[Authorize]` above the action states: "Applied at the controller scope, so the user must be authenticated for all actions on the controller." Another annotation for `ManageAccount()` states: "Only authenticated users may execute ManageAccount."

```
[AllowAnonymous]
public IActionResult Login()
{
    return Ok();
}
```

[AllowAnonymous] overrides [Authorize] to allow unauthenticated users.

Login can be executed by anonymous users.

**WARNING** If you apply the `[Authorize]` attribute globally, be sure to add the `[AllowAnonymous]` attribute to your login actions, error actions, password reset actions, and any other actions that you need unauthenticated users to execute. If you’re using the default Identity UI described in chapter 14, this is already configured for you.

If an unauthenticated user attempts to execute an action protected by the `[Authorize]` attribute, traditional web apps will redirect them to the login page. But what about Web APIs? And what about more complex scenarios, where a user is logged in but doesn’t have the necessary claims to execute an action? In section 15.2.2 we’ll look at how the ASP.NET Core authentication services handle all of this for you.

### 15.2.2 Handling unauthorized requests

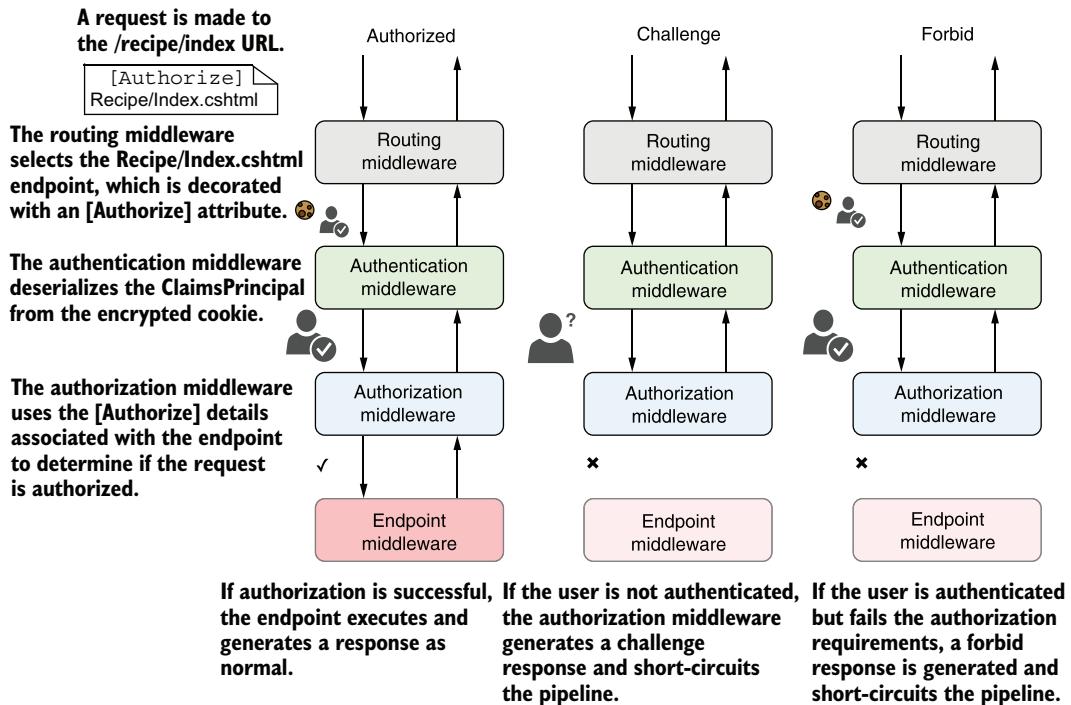
In the previous section you saw how to apply the `[Authorize]` attribute to an action to ensure only authenticated users can execute it. In section 15.3 we’ll look at more complex examples that require you to also have a specific claim. In both cases, you must meet one or more authorization requirements (for example, you must be authenticated) to execute the action.

If the user meets the authorization requirements, then the request passes unimpeded through the `AuthorizationMiddleware`, and the endpoint is executed in the `EndpointMiddleware`. If they don’t meet the requirements for the selected endpoint, the `AuthorizationMiddleware` will short-circuit the request. Depending on why the request failed authorization, the `AuthorizationMiddleware` generates one of two different types of responses, as shown in figure 15.3:

- *Challenge*—This response indicates the user was not authorized to execute the action because they weren’t yet logged in.
- *Forbid*—This response indicates that the user was logged in but didn’t meet the requirements to execute the action. They didn’t have a required claim, for example.

**NOTE** If you apply the `[Authorize]` attribute in basic form, as you did in section 15.2.1, you will only generate challenge responses. In this case, a challenge response will be generated for unauthenticated users, but authenticated users will always be authorized.

The exact HTTP response generated by a challenge or forbid response typically depends on the type of application you’re building and so the type of authentication your application uses: a traditional web application with Razor Pages, or an API application.



**Figure 15.3** The three types of response to an authorization attempt. In the left example, the request contains an authentication cookie, so the user is authenticated in the AuthenticationMiddleware. The AuthorizationMiddleware confirms the authenticated user can access the selected endpoint, so the endpoint is executed. In the center example, the request is not authenticated, so the AuthorizationMiddleware generates a challenge response. In the right example, the request is authenticated, but the user does not have permission to execute the endpoint, so a forbid response is generated.

For traditional web apps using cookie authentication, such as when you use ASP.NET Core Identity, as in chapter 14, the challenge and forbid responses generate an HTTP redirect to a page in your application.

A challenge response indicates the user isn't yet authenticated, so they're redirected to the login page for the app. After logging in, they can attempt to execute the protected resource again.

A forbid response means the request was from a user that *already* logged in, but they're still not allowed to execute the action. Consequently, the user is redirected to a "forbidden" or "access denied" web page, as shown in figure 15.4, which informs them they can't execute the action or Razor Page.

The preceding behavior is standard for traditional web apps, but Web APIs typically use a different approach to authentication, as you saw in chapter 14. Instead of logging in and using the API directly, you'd typically log in to a third-party application

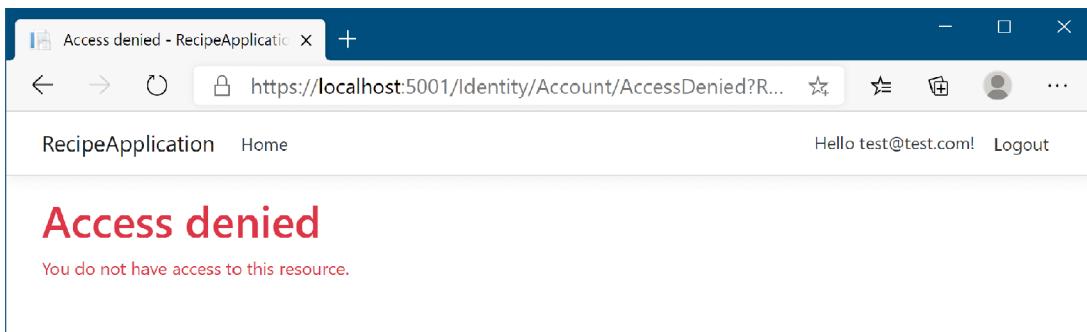


Figure 15.4 A forbid response in traditional web apps using cookie authentication. If you don't have permission to execute a Razor Page and you're already logged in, you'll be redirected to an "access denied" page.

that provides a token to the client-side SPA or mobile app. The client-side app sends this token when it makes a request to your Web API.

Authenticating a request for a Web API using tokens is essentially identical to a traditional web app that uses cookies; AuthenticationMiddleware deserializes the cookie or token to create the `ClaimsPrincipal`. The difference is in how a Web API handles authorization failures.

When a Web API app generates a challenge response, it returns a 401 Unauthorized error response to the caller. Similarly, when the app generates a forbid response, it returns a 403 Forbidden response. The traditional web app essentially handled these errors by automatically redirecting unauthorized users to the login or "access denied" page, but the Web API doesn't do this. It's up to the client-side SPA or mobile app to detect these errors and handle them as appropriate.

**TIP** The difference in authorization behavior is one of the reasons I generally recommend creating separate apps for your APIs and Razor pages apps—it's possible to have both in the same app, but the configuration is more complex.

The different behavior between traditional web apps and SPAs can be confusing initially, but you generally don't need to worry about that too much in practice. Whether you're building a Web API or a traditional MVC web app, the authorization code in your app looks the same in both cases. Apply `[Authorize]` attributes to your endpoints, and let the framework take care of the differences for you.

**NOTE** In chapter 14 you saw how to configure ASP.NET Core Identity in a Razor Pages app. This chapter assumes you're building a Razor Pages app too, but the chapter is equally applicable if you're building a Web API. Authorization policies are applied in the same way, whichever style of app you're building. It's only the final response of unauthorized requests that differs.

You've seen how to apply the most basic authorization requirement—restricting an endpoint to authenticated users only—but most apps need something more subtle than this all-or-nothing approach.

Consider the airport scenario from section 15.1. Being authenticated (having a passport) isn't enough to get you through security. Instead, you also need a specific claim: `BoardingPassNumber`. In the next section we'll look at how you can implement a similar requirement in ASP.NET Core.

### 15.3 Using policies for claims-based authorization

In the previous section, you saw how to require that users be logged in to access an endpoint. In this section you'll see how to apply additional requirements. You'll learn to use authorization policies to perform claims-based authorization to require that a logged in user have the required claims to execute a given endpoint.

In chapter 14 you saw that authentication in ASP.NET Core centers around a `ClaimsPrincipal` object, which represents the user. This object has a collection of claims that contain pieces of information about the user, such as their name, email, and date of birth.

You can use these to customize the app for each user, by displaying a welcome message addressing the user by name, for example, but you can also use claims for authorization. For example, you might only authorize a user if they have a specific claim (such as `BoardingPassNumber`) or if a claim has a specific value (`FrequentFlyerClass` claim with the value `Gold`).

In ASP.NET Core the rules that define whether a user is authorized are encapsulated in a *policy*.

**DEFINITION** A *policy* defines the requirements you must meet for a request to be authorized.

Policies can be applied to an action using the `[Authorize]` attribute, similar to the way you saw in section 15.2.1. This listing shows a Razor Page `PageModel` that represents the first authorization step in the airport scenario. The `AirportSecurity.cshtml` Razor Page is protected by an `[Authorize]` attribute, but you've also provided a policy name: "CanEnterSecurity".

#### Listing 15.3 Applying an authorization policy to a Razor Page

```
[Authorize("CanEnterSecurity")]
public class AirportSecurityModel : PageModel
{
    public void OnGet()      ← Only users that satisfy the
                           "CanEnterSecurity" policy
                           can execute the Razor Page.
}
```

← Applying the "CanEnterSecurity" policy using [Authorize]

If a user attempts to execute the `AirportSecurity.cshtml` Razor Page, the authorization middleware will verify whether the user satisfies the policy's requirements (we'll look at the policy itself shortly). This gives one of three possible outcomes:

- *The user satisfies the policy*—The middleware pipeline continues, and the Endpoint-Middleware executes the Razor Page as normal.
- *The user is unauthenticated*—The user is redirected to the login page.
- *The user is authenticated but doesn't satisfy the policy*—The user is redirected to a “forbidden” or “access denied” page.

These three outcomes correlate with the real-life outcomes you might expect when trying to pass through security at the airport:

- *You have a valid boarding pass*—You can enter security as normal.
- *You don't have a boarding pass*—You're redirected to purchase a ticket.
- *Your boarding pass is invalid (you turned up a day late, for example)*—You're blocked from entering.

Listing 15.3 shows how you can apply a policy to a Razor Page using the `[Authorize]` attribute, but you still need to define the `CanEnterSecurity` policy.

You add policies to an ASP.NET Core application in the `ConfigureServices` method of `Startup.cs`, as shown in listing 15.4. First you add the authorization services using `AddAuthorization()`, and then you can add policies by calling `AddPolicy()` on the `AuthorizationOptions` object. You define the policy itself by calling methods on a provided `AuthorizationPolicyBuilder` (called `policyBuilder` here).

#### **Listing 15.4 Adding an authorization policy using `AuthorizationPolicyBuilder`**

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthorization(options =>
    {
        options.AddPolicy(
            "CanEnterSecurity",
            policyBuilder => policyBuilder
                .RequireClaim("BoardingPassNumber"));
    });
    // Additional service configuration
}
```

The code is annotated with several callouts:

- An arrow points to the first line of the `options` block with the text "Adds a new policy".
- An arrow points to the `options.AddAuthorization()` call with the text "Calls AddAuthorization to configure AuthorizationOptions".
- An arrow points to the `options.AddPolicy()` call with the text "Provides a name for the policy".
- An arrow points to the `policyBuilder` parameter of the `options.AddPolicy()` call with the text "Defines the policy requirements using AuthorizationPolicyBuilder".

When you call `AddPolicy` you provide a name for the policy, which should match the value you use in your `[Authorize]` attributes, and you define the requirements of the policy. In this example, you have a single simple requirement: the user must have a claim of type `BoardingPassNumber`. If a user has this claim, whatever its value, the policy will be satisfied and the user will be authorized.

**REMEMBER** A *claim* is information about the user, as a key-value pair. A policy defines the requirements for successful authorization. A policy can require

that a user have a given claim, as well as specify more complex requirements, as you'll see shortly.

`AuthorizationPolicyBuilder` contains several methods for creating simple policies like this, as shown in table 15.1. For example, an overload of the `RequireClaim()` method lets you specify a specific value that a claim must have. The following would let you create a policy where the "BoardingPassNumber" claim must have a value of "A1234":

```
policyBuilder => policyBuilder.RequireClaim("BoardingPassNumber", "A1234");
```

**Table 15.1 Simple policy builder methods on `AuthorizationPolicyBuilder`**

Method	Policy behavior
<code>RequireAuthenticatedUser()</code>	The required user must be authenticated. Creates a policy similar to the default <code>[Authorize]</code> attribute, where you don't set a policy.
<code>RequireClaim(claim, values)</code>	The user must have the specified claim. If provided, the claim must be one of the specified values.
<code>RequireUsername(username)</code>	The user must have the specified username.
<code>RequireAssertion(function)</code>	Executes the provided lambda function, which returns a <code>bool</code> , indicating whether the policy was satisfied.

### Role-based authorization vs. claims-based authorization

If you look at all of the methods available on the `AuthorizationPolicyBuilder` type using IntelliSense, you might notice that there's a method I didn't mention in table 15.1, `RequireRole()`. This is a remnant of the role-based approach to authorization used in previous versions of ASP.NET, and I don't recommend using it.

Before Microsoft adopted the claims-based authorization used by ASP.NET Core and recent versions of ASP.NET, role-based authorization was the norm. Users were assigned to one or more roles, such as `Administrator` or `Manager`, and authorization involved checking whether the current user was in the required role.

This role-based approach to authorization is possible in ASP.NET Core, but it's primarily used for legacy compatibility reasons. Claims-based authorization is the suggested approach. Unless you're porting a legacy app that uses roles, I suggest you embrace claims-based authorization and leave those roles behind.

You can use these methods to build simple policies that can handle basic situations, but often you'll need something more complicated. What if you wanted to create a policy that enforces that only users over the age of 18 can execute an endpoint?

The `DateOfBirth` claim provides the information you need, but there's not a *single* correct value, so you couldn't use the `RequireClaim()` method. You *could* use the

`RequireAssertion()` method and provide a function that calculates the age from the `DateOfBirth` claim, but that could get messy pretty quickly.

For more complex policies that can't be easily defined using the `RequireClaim()` method, I recommend you take a different approach and create a custom policy, as you'll see in the following section.

## 15.4 Creating custom policies for authorization

You've already seen how to create a policy by requiring a specific claim, or requiring a specific claim with a specific value, but often the requirements will be more complex than that. In this section you'll learn how to create custom authorization requirements and handlers. You'll also see how to configure authorization requirements where there are multiple ways to satisfy a policy, any of which are valid.

Let's return to the airport example. You've already configured the policy for passing through security, and now you're going to configure the policy that controls whether you're authorized to enter the airline lounge.

As you saw in figure 15.1, you're allowed to enter the lounge if you have a `FrequentFlyerClass` claim with a value of `Gold`. If this was the only requirement, you could use `AuthorizationPolicyBuilder` to create a policy like this:

```
options.AddPolicy("CanAccessLounge", policyBuilder =>
    policyBuilder.RequireClaim("FrequentFlyerClass", "Gold");
```

But what if the requirements are more complicated than this? For example, suppose you can enter the lounge if you're at least 18 years old (as calculated from the `DateOfBirth` claim) and you're one of the following:

- You're a gold-class frequent flyer (have a `FrequentFlyerClass` claim with value `"Gold"`)
- You're an employee of the airline (have an `EmployeeNumber` claim)

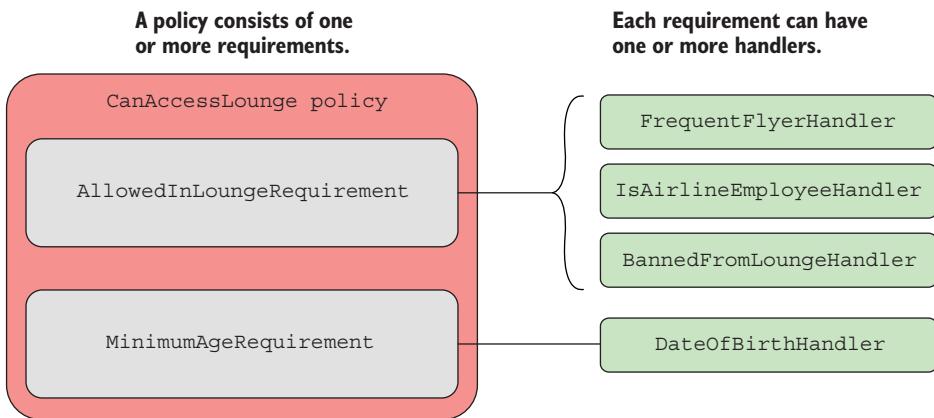
If you've ever been banned from the lounge (you have an `IsBannedFromLounge` claim), you won't be allowed in, even if you satisfy the other requirements.

There's no way of achieving this complex set of requirements with the basic usage of `AuthorizationPolicyBuilder` you've seen so far. Luckily, these methods are a wrapper around a set of building blocks that you can combine to achieve the desired policy.

### 15.4.1 Requirements and handlers: The building blocks of a policy

Every policy in ASP.NET Core consists of one or more *requirements*, and every requirement can have one or more *handlers*. For the airport lounge example, you have a single policy ("`CanAccessLounge`"), two requirements (`MinimumAgeRequirement` and `AllowedInLoungeRequirement`), and several handlers, as shown in figure 15.5.

For a policy to be satisfied, a user must fulfill *all* the requirements. If the user fails *any* of the requirements, the authorize middleware won't allow the protected endpoint



**Figure 15.5** A policy can have many requirements, and every requirement can have many handlers. By combining multiple requirements in a policy, and by providing multiple handler implementations, you can create complex authorization policies that meet any of your business requirements.

to be executed. In this example, a user must be allowed to access the lounge *and* must be over 18 years old.

Each requirement can have one or more handlers, which will confirm that the requirement has been satisfied. For example, as shown in figure 15.5, `AllowedInLoungeRequirement` has two handlers that can satisfy the requirement:

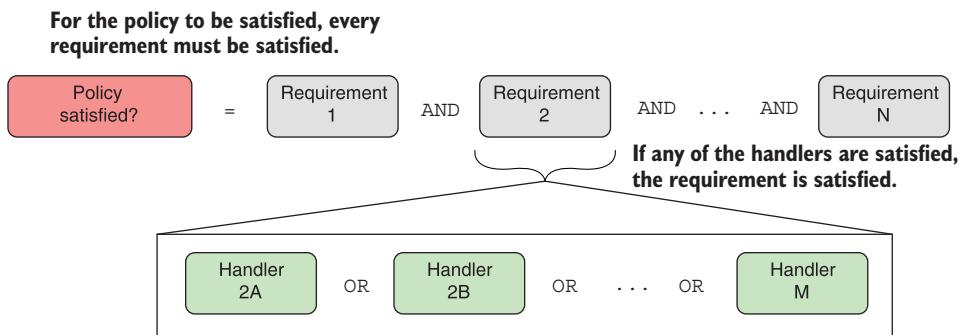
- `FrequentFlyerHandler`
- `IsAirlineEmployeeHandler`

If the user satisfies either of these handlers, then `AllowedInLoungeRequirement` is satisfied. You don't need all handlers for a requirement to be satisfied, you just need one.

**NOTE** Figure 15.5 shows a third handler, `BannedFromLoungeHandler`, which I'll cover in section 15.4.2. It's slightly different in that it can only *fail* a requirement, not *satisfy* it.

You can use requirements and handlers to achieve most any combination of behavior you need for a policy. By combining handlers for a requirement, you can validate conditions using a logical OR: if any of the handlers are satisfied, the requirement is satisfied. By combining requirements, you create a logical AND: all the requirements must be satisfied for the policy to be satisfied, as shown in figure 15.6.

**TIP** You can also add multiple policies to a Razor Page or action method by applying the `[Authorize]` attribute multiple times; for example, `[Authorize("Policy1"), Authorize("Policy2")]`. All policies must be satisfied for the request to be authorized.



**Figure 15.6** For a policy to be satisfied, every requirement must be satisfied. A requirement is satisfied if any of the handlers are satisfied.

I've highlighted requirements and handlers that will make up your "CanAccessLounge" policy, so in the next section you'll build each of the components and apply them to the airport sample app.

#### 15.4.2 Creating a policy with a custom requirement and handler

You've seen all the pieces that make up a custom authorization policy, so in this section we'll explore the implementation of the "CanAccessLounge" policy.

##### CREATING AN IAUTHORIZATIONREQUIREMENT TO REPRESENT A REQUIREMENT

As you've seen, a custom policy can have multiple requirements, but what *is* a requirement in code terms? Authorization requirements in ASP.NET Core are any class that implements the `IAuthorizationRequirement` interface. This is a blank, marker interface, which you can apply to any class to indicate that it represents a requirement.

If the interface doesn't have any members, you might be wondering what the requirement class needs to look like. Typically, they're simple POCO classes. The following listing shows `AllowedInLoungeRequirement`, which is about as simple as a requirement can get. It has no properties or methods; it implements the required `IAuthorizationRequirement` interface.

##### Listing 15.5 AllowedInLoungeRequirement

```
public class AllowedInLoungeRequirement : IAuthorizationRequirement { }      The interface identifies the class as an authorization requirement.
```

This is the simplest form of requirement, but it's also common for them to have one or two properties that make the requirement more generalized. For example, instead of creating the highly specific `MustBe18YearsOldRequirement`, you could instead create a parameterized `MinimumAgeRequirement`, as shown in the following listing. By providing the minimum age as a parameter to the requirement, you can reuse the requirement for other policies with different minimum age requirements.

**Listing 15.6 The parameterized MinimumAgeRequirement**

```
public class MinimumAgeRequirement : IAuthorizationRequirement
{
    public MinimumAgeRequirement(int minimumAge)
    {
        MinimumAge = minimumAge;
    }
    public int MinimumAge { get; } ←
}
Handlers can use the exposed
minimum age to determine whether
the requirement is satisfied. ←
The minimum
age is provided
when the
requirement is
created. ←
The interface
identifies the class
as an authorization
requirement. ←
```

The requirements are the easy part. They represent each of the components of the policy that must be satisfied for the policy to be satisfied overall.

**CREATING A POLICY WITH MULTIPLE REQUIREMENTS**

You've created the two requirements, so now you can configure the "CanAccessLounge" policy to use them. You configure your policies as you did before, in the `ConfigureServices` method of `Startup.cs`. Listing 15.7 shows how to do this by creating an instance of each requirement and passing them to `AuthorizationPolicyBuilder`. The authorization handlers will use these requirement objects when attempting to authorize the policy.

**Listing 15.7 Creating an authorization policy with multiple requirements**

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthorization(options =>
    {
        options.AddPolicy(
            "CanEnterSecurity",
            policyBuilder => policyBuilder
                .RequireClaim(Claims.BoardingPassNumber));
        options.AddPolicy(
            "CanAccessLounge",
            policyBuilder => policyBuilder.AddRequirements(
                new MinimumAgeRequirement(18),
                new AllowedInLoungeRequirement()));
    });
    // Additional service configuration
}
```

Adds a new policy for the airport lounge, called CanAccessLounge

Adds the previous simple policy for passing through security

Adds an instance of each IAuthorizationRequirement object

You now have a policy called "CanAccessLounge" with two requirements, so you can apply it to a Razor Page or action method using the `[Authorize]` attribute, in exactly the same way you did for the "CanEnterSecurity" policy:

```
[Authorize("CanAccessLounge")]
public class AirportLoungeModel : PageModel
{
```

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

When a request is routed to the `AirportLounge.cshtml` Razor Page, the authorize middleware executes the authorization policy and each of the requirements is inspected. But you saw earlier that the requirements are purely data; they indicate what needs to be fulfilled, but they don't describe how that has to happen. For that, you need to write some handlers.

### CREATING AUTHORIZATION HANDLERS TO SATISFY YOUR REQUIREMENTS

Authorization handlers contain the logic of how a specific `IAuthorizationRequirement` can be satisfied. When executed, a handler can do one of three things:

- Mark the requirement handling as a success
- Not do anything
- Explicitly fail the requirement

Handlers should implement `AuthorizationHandler<T>`, where `T` is the type of requirement they handle. For example, the following listing shows a handler for `AllowedInLoungeRequirement` that checks whether the user has a claim called `FrequentFlyerClass` with a value of `Gold`.

#### **Listing 15.8 FrequentFlyerHandler for AllowedInLoungeRequirement**

You must override the abstract `HandleRequirementAsync` method.

```
public class FrequentFlyerHandler :  
    AuthorizationHandler<AllowedInLoungeRequirement>  
{  
    protected override Task HandleRequirementAsync(  
        AuthorizationHandlerContext context,  
        AllowedInLoungeRequirement requirement)  
    {  
        if (context.User.HasClaim("FrequentFlyerClass", "Gold"))  
        {  
            context.Succeed(requirement);  
        }  
        return Task.CompletedTask;  
    }  
}
```

The context contains details such as the `ClaimsPrincipal` user object.

The handler implements `AuthorizationHandler<T>`.

The requirement instance to handle

If the user had the necessary claim, then mark the requirement as satisfied by calling `Succeed`.

If the requirement wasn't satisfied, do nothing.

This handler is functionally equivalent to the simple `RequireClaim()` handler you saw at the start of section 15.4, but using the requirement and handler approach instead.

When a request is routed to the `AirportLounge.cshtml` Razor Page, the authorization middleware sees the `[Authorize]` attribute on the endpoint with the "CanAccessLounge" policy. It loops through all the requirements in the policy, and all the handlers for each requirement, calling the `HandleRequirementAsync` method for each.

The authorization middleware passes the current `AuthorizationHandlerContext` and the requirement to be checked to each handler. The current `ClaimsPrincipal`

being authorized is exposed on the context as the `User` property. In listing 15.8, `FrequentFlyerHandler` uses the context to check for a claim called `FrequentFlyerClass` with the `Gold` value, and if it exists, indicates that the user is allowed to enter the airline lounge by calling `Succeed()`.

**NOTE** Handlers mark a requirement as being successfully satisfied by calling `context.Succeed()` and passing the requirement as an argument.

It's important to note the behavior when the user *doesn't* have the claim. `FrequentFlyerHandler` doesn't do anything if this is the case (it returns a completed Task to satisfy the method signature).

**NOTE** Remember, if any of the handlers associated with a requirement pass, then the requirement is a success. Only *one* of the handlers must succeed for the requirement to be satisfied.

This behavior, whereby you either call `context.Succeed()` or do nothing, is typical for authorization handlers. The following listing shows the implementation of `IsAirlineEmployeeHandler`, which uses a similar claim check to determine whether the requirement is satisfied.

#### Listing 15.9 IsAirlineEmployeeHandler

```
public class IsAirlineEmployeeHandler : AuthorizationHandler<AllowedInLoungeRequirement>
{
    protected override Task HandleRequirementAsync(
        AuthorizationHandlerContext context,
        AllowedInLoungeRequirement requirement)
    {
        if(context.User.HasClaim(c => c.Type == "EmployeeNumber"))
        {
            context.Succeed(requirement);
        }
        return Task.CompletedTask;
    }
}
```

The code is annotated with several callouts:

- A callout on the first line of the class definition points to the generic parameter `<T>` with the text "The handler implements `AuthorizationHandler<T>`".
- A callout on the `HandleRequirementAsync` method header points to the method with the text "You must override the abstract `HandleRequirementAsync` method.".
- A callout on the `if` statement points to the condition with the text "Checks whether the user has the EmployeeNumber claim".
- A callout on the `return Task.CompletedTask;` line points to the text "If the requirement wasn't satisfied, do nothing.".
- A callout on the `If the user has the necessary claim, mark the requirement as satisfied by calling Succeed.` line points to the `context.Succeed(requirement);` line.

**TIP** It's possible to write very generic handlers that can be used with multiple requirements, but I suggest sticking to handling a single requirement only. If you need to extract some common functionality, move it to an external service and call that from both handlers.

This pattern of authorization handler is common,<sup>1</sup> but in some cases, instead of checking for a *success* condition, you might want to check for a *failure* condition. In the

<sup>1</sup> I'll leave the implementation of `MinimumAgeHandler` for `MinimumAgeRequirement` as an exercise. You can find an example in the code samples for the chapter.

airport example, you don't want to authorize someone who was previously banned from the lounge, even if they would otherwise be allowed to enter.

You can handle this scenario by using the `context.Fail()` method exposed on the context, as shown in the following listing. Calling `Fail()` in a handler will always cause the requirement, and hence the whole policy, to fail. You should only use it when you want to guarantee failure, even if other handlers indicate success.

#### Listing 15.10 Calling `context.Fail()` in a handler to fail the requirement

```
public class BannedFromLoungeHandler : AuthorizationHandler<AllowedInLoungeRequirement>
{
    protected override Task HandleRequirementAsync(
        AuthorizationHandlerContext context,
        AllowedInLoungeRequirement requirement)
    {
        if(context.User.HasClaim(c => c.Type == "IsBanned"))
        {
            context.Fail(); ←
        }
        return Task.CompletedTask; ←
    }
}
```

The code is annotated with several callouts:

- A callout from the line `AuthorizationHandler<T>` points to the line `← The handler implements AuthorizationHandler<T>.`
- A callout from the line `AuthorizationHandlerContext context,` points to the line `← You must override the abstract HandleRequirementAsync method.`
- A callout from the line `if(context.User.HasClaim(c => c.Type == "IsBanned"))` points to the line `← Checks whether the user has the IsBanned claim`.
- A callout from the line `context.Fail();` points to the line `← If the user has the claim, fail the requirement by calling Fail. The whole policy will fail.`
- A callout from the line `return Task.CompletedTask;` points to the line `← If the claim wasn't found, do nothing.`

In most cases, your handlers will either call `Succeed()` or will do nothing, but the `Fail()` method is useful when you need a kill-switch to guarantee that a requirement won't be satisfied.

**NOTE** Whether a handler calls `Succeed()`, `Fail()`, or neither, the authorization system will always execute all of the handlers for a requirement, and all the requirements for a policy, so you can be sure your handlers will always be called.

The final step to complete your authorization implementation for the app is to register the authorization handlers with the DI container, as shown in the following listing.

#### Listing 15.11 Registering the authorization handlers with the DI container

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthorization(options =>
    {
        options.AddPolicy(
            "CanEnterSecurity",
            policyBuilder => policyBuilder
                .RequireClaim(Claims.BoardingPassNumber));
        options.AddPolicy(
            "CanAccessLounge",
```

```
policyBuilder => policyBuilder.AddRequirements(
    new MinimumAgeRequirement(18),
    new AllowedInLoungeRequirement()
);
}) ;
services.AddSingleton<IAuthorizationHandler, MinimumAgeHandler>();
services.AddSingleton<IAuthorizationHandler, FrequentFlyerHandler>();
services
    .AddSingleton<IAuthorizationHandler, BannedFromLoungeHandler>();
services
    .AddSingleton<IAuthorizationHandler, IsAirlineEmployeeHandler>();
// Additional service configuration
}
```

For this app, the handlers don't have any constructor dependencies, so I've registered them as singletons with the container. If your handlers have scoped or transient dependencies (the EF Core `DbContext`, for example), you might want to register them as scoped instead, as appropriate.

**NOTE** Services are registered with a lifetime of either transient, scoped, or singleton, as discussed in chapter 10.

You can combine the concepts of policies, requirements, and handlers in many ways to achieve your goals for authorization in your application. The example in this section, although contrived, demonstrates each of the components you need to apply authorization declaratively at the action method or Razor Page level, by creating policies and applying the `[Authorize]` attribute as appropriate.

As well as applying the `[Authorize]` attribute explicitly to actions and Razor Pages, you can also configure it globally, so that a policy is applied to every Razor Page or controller in your application. Additionally, for Razor Pages you can apply different authorization policies to different folders. You can read more about applying authorization policies using conventions in Microsoft's "Razor Pages authorization conventions in ASP.NET Core" documentation: <http://mng.bz/nMm2>.

There's one area, however, where the `[Authorize]` attribute falls short: resource-based authorization. The `[Authorize]` attribute attaches metadata to an endpoint, so the authorization middleware can authorize the user *before* an endpoint is executed, but what if you need to authorize the action *during* the action method or Razor Page handler?

This is common when you're applying authorization at the document or resource level. If users are only allowed to edit documents they created, then you need to load the document before you can tell whether they're allowed to edit it! This isn't easy with the declarative `[Authorize]` attribute approach, so you must use an alternative, imperative approach. In the next section, you'll see how to apply this resource-based authorization in a Razor Page handler.

## 15.5 Controlling access with resource-based authorization

In this section you'll learn about resource-based authorization. This is used when you need to know details about the resource being protected to determine if a user is authorized. You'll learn how to apply authorization policies manually using the `IAuthorizationService`, and how to create resource-based `AuthorizationHandlers`.

Resource-based authorization is a common problem for applications, especially when you have users who can create or edit some sort of document. Consider the recipe application you built in the previous three chapters. This app lets users create, view, and edit recipes.

Up to this point, everyone can create new recipes, and anyone can edit any recipe, even if they haven't logged in. Now you want to add some additional behavior:

- Only authenticated users should be able to create new recipes.
- You can only edit the recipes you created.

You've already seen how to achieve the first of these requirements: decorate the `Create.cshtml` Razor Page with an `[Authorize]` attribute and don't specify a policy, as shown in this listing. This will force the user to authenticate before they can create a new recipe.

### Listing 15.12 Adding `AuthorizeAttribute` to the `Create.cshtml` Razor Page

```
[Authorize]
public class CreateModel : PageModel
{
    [BindProperty]
    public CreateRecipeCommand Input { get; set; }

    public void OnGet()
    {
        Input = new CreateRecipeCommand();
    }

    public async Task<IActionResult> OnPost()
    {
        // Method body not shown for brevity
    }
}
```

**Users must be authenticated to execute the `Create.cshtml` Razor Page.**

**All page handlers are protected. You can only apply `[Authorize]` to the `PageModel`, not handlers.**

**TIP** As with all filters, you can only apply the `[Authorize]` attribute to the Razor Page, not to individual page handlers. The attribute applies to all page handlers in the Razor Page.

Adding the `[Authorize]` attribute fulfills your first requirement, but unfortunately, with the techniques you've seen so far, you have no way to fulfill the second. You could apply a policy that either permits or denies a user the ability to edit *all* recipes, but there's currently no easy way to restrict this so that a user can only edit *their own* recipes.

In order to find out who created the Recipe, you must first load it from the database. Only then can you attempt to authorize the user, taking the specific recipe (resource) into account. The following listing shows a partially implemented page handler for how this might look, where authorization occurs partway through the method, after the Recipe object has been loaded.

**Listing 15.13 The Edit.cshtml page must load the Recipe before authorizing the request**

```

    The id of
    the recipe
    to edit is
    provided
    by model
    binding.

    public IActionResult OnGet(int id)
    {
        var recipe = _service.GetRecipe(id);
        var createdBy = recipe.CreatedById;
        // Authorize user based on createdBy
        if(isAuthorized)
        {
            return View(recipe);
        }
    }

```

You must load the Recipe from the database before you know who created it.

The action method can only continue if the user was authorized.

You must authorize the current user to verify they're allowed to edit this specific Recipe.

You need access to the resource (in this case, the Recipe entity) to perform the authorization, so the declarative [Authorize] attribute can't help you. In section 15.5.1 you'll see the approach you need to take to handle these situations and to apply authorization inside the action method or Razor Page.

**WARNING** Be careful when exposing the integer ID of your entities in the URL, as in listing 15.13. Users will be able to edit every entity by modifying the ID in the URL to access a different entity. Be sure to apply authorization checks, or you could expose a security vulnerability called *insecure direct object reference (IDOR)*.<sup>2</sup>

### 15.5.1 Manually authorizing requests with *IAuthorizationService*

All of the approaches to authorization so far have been *declarative*. You apply the [Authorize] attribute, with or without a policy name, and you let the framework take care of performing the authorization itself.

For this recipe-editing example, you need to use *imperative* authorization, so you can authorize the user after you've loaded the Recipe from the database. Instead of applying a marker saying, "Authorize this method," you need to write some of the authorization code yourself.

**DEFINITION** *Declarative* and *imperative* are two different styles of programming. Declarative programming describes *what* you're trying to achieve and lets the framework figure out how to achieve it. Imperative programming describes *how* to achieve something by providing each of the steps needed.

<sup>2</sup> You can read about insecure direct object reference (IDOR) and ways to counteract it on the Open Web Application Security Project (OWASP): <https://owasp.org/www-chapter-ghana/assets/slides/IDOR.pdf>.

ASP.NET Core exposes `IAuthorizationService`, which you can inject into your Razor Pages and controllers for imperative authorization. The following listing shows how you can update the `Edit.cshtml` Razor Page (shown partially in listing 15.13) to use the `IAuthorizationService` and verify whether the action is allowed to continue execution.

#### Listing 15.14 Using `IAuthorizationService` for resource-based authorization

```
[Authorize]
public class EditModel : PageModel
{
    [BindProperty]
    public Recipe Recipe { get; set; }

    private readonly RecipeService _service;
    private readonly IAuthorizationService _authService; <-- Only authenticated users should be allowed to edit recipes.

    public EditModel(
        RecipeService service,
        IAuthorizationService authService)
    {
        _service = service;
        _authService = authService;
    }

    Load the Recipe from the database. <--> public async Task<IActionResult> OnGet(int id)
    {
        Recipe = _service.GetRecipe(id);
        var authResult = await _authService
            .AuthorizeAsync(User, Recipe, "CanManageRecipe");
        if (!authResult.Succeeded)
        {
            return new ForbidResult(); <--> If authorization failed, returns a Forbidden result
        }

        return Page();
    }
} <--> If authorization was successful, continues displaying the Razor Page
```

**IAuthorizationService is injected into the class constructor using DI.**

**Calls `IAuthorizationService`, providing `ClaimsPrincipal`, `resource`, and the `policy name`**

`IAuthorizationService` exposes an `AuthorizeAsync` method, which requires three things to authorize the request:

- The `ClaimsPrincipal` user object, exposed on the `PageModel` as `User`
- The resource being authorized: `Recipe`
- The policy to evaluate: `"CanManageRecipe"`

The authorization attempt returns an `AuthorizationResult` object, which indicates whether the attempt was successful via the `Succeeded` property. If the attempt wasn't successful, you should return a new `ForbidResult`, which will be converted either into an HTTP 403 Forbidden response or will redirect the user to the “access denied”

page, depending on whether you're building a traditional web app with Razor Pages or a Web API.

**NOTE** As mentioned in section 15.2.2, which type of response is generated depends on which authentication services are configured. The default Identity configuration, used by Razor Pages, generates redirects. The JWT bearer token authentication typically used with Web APIs generates HTTP 401 and 403 responses instead.

You've configured the imperative authorization in the Edit.cshtml Razor Page itself, but you still need to define the "CanManageRecipe" policy that you use to authorize the user. This is the same process as for declarative authorization, so you have to do the following:

- Create a *policy* in `ConfigureServices` by calling `AddAuthorization()`
- Define one or more *requirements* for the policy
- Define one or more *handlers* for each requirement
- Register the handlers in the DI container

With the exception of the handler, these steps are all identical to the declarative authorization approach with the `[Authorize]` attribute, so I'll only run through them briefly here.

First, you can create a simple `IAuthorizationRequirement`. As with many requirements, this contains no data and simply implements the marker interface.

```
public class IsRecipeOwnerRequirement : IAuthorizationRequirement { }
```

Defining the policy in `ConfigureServices` is similarly simple, as you have only this single requirement. Note that there's nothing resource-specific in any of this code so far:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddAuthorization(options => {
        options.AddPolicy("CanManageRecipe", policyBuilder =>
            policyBuilder.AddRequirements(new IsRecipeOwnerRequirement());
    });
}
```

You're halfway there; all you need to do now is create an authorization handler for `IsRecipeOwnerRequirement` and register it with the DI container.

### 15.5.2 Creating a resource-based AuthorizationHandler

Resource-based authorization handlers are essentially the same as the authorization handler implementations you saw in section 15.4.2. The only difference is that the handler also has access to the resource being authorized.

To create a resource-based handler, you should derive from the `AuthorizationHandler<TRequirement, TResource>` base class, where `TRequirement` is the type of

requirement to handle, and `TResource` is the type of resource that you provide when calling `IAuthorizationService`. Compare this to the `AuthorizationHandler<T>` class you implemented previously, where you only specified the requirement.

This listing shows the handler implementation for your recipe application. You can see that you've specified the requirement as `IsRecipeOwnerRequirement` and the resource as `Recipe`, and you have implemented the `HandleRequirementAsync` method.

### Listing 15.15 `IsRecipeOwnerHandler` for resource-based authorization

```

public class IsRecipeOwnerHandler : AuthorizationHandler<IsRecipeOwnerRequirement, Recipe>
{
    private readonly UserManager<ApplicationUser> _userManager;
    public IsRecipeOwnerHandler(
        UserManager<ApplicationUser> userManager)
    {
        _userManager = userManager;
    }
    protected override async Task HandleRequirementAsync(
        AuthorizationHandlerContext context,
        IsRecipeOwnerRequirement requirement,
        Recipe resource)
    {
        var appUser = await _userManager.GetUserAsync(context.User);
        if(appUser == null)           ← If you aren't authenticated,
        {                            appUser will be null.
            return;
        }
        if(resource.CreatedById == appUser.Id) ← Checks whether the current
        {                                user created the Recipe by
            context.Succeed(requirement);   ←
        }
    }
}

```

The code is annotated with several callouts:

- A callout from the requirement parameter in the constructor to the requirement parameter in the `HandleRequirementAsync` method, labeled "Implements the necessary base class, specifying the requirement and resource type".
- A callout from the `_userManager` field to the `userManager` parameter in the constructor, labeled "Injects an instance of the `UserManager<T>` class using DI".
- A callout from the `appUser` variable to the condition where it is null, labeled "As well as the context and requirement, you're also provided the resource instance." and "If you aren't authenticated, appUser will be null."
- A callout from the `CreatedById` check to the `Succeed` call, labeled "If the user created the document, Succeed the requirement; otherwise, do nothing."

This handler is slightly more complicated than the examples you've seen previously, primarily because you're using an additional service, `UserManager<>`, to load the  `ApplicationUser` entity based on `ClaimsPrincipal` from the request.

**NOTE** In practice, the `ClaimsPrincipal` will likely already have the `Id` added as a claim, making the extra step unnecessary in this case. This example shows the general pattern if you need to use dependency-injected services.

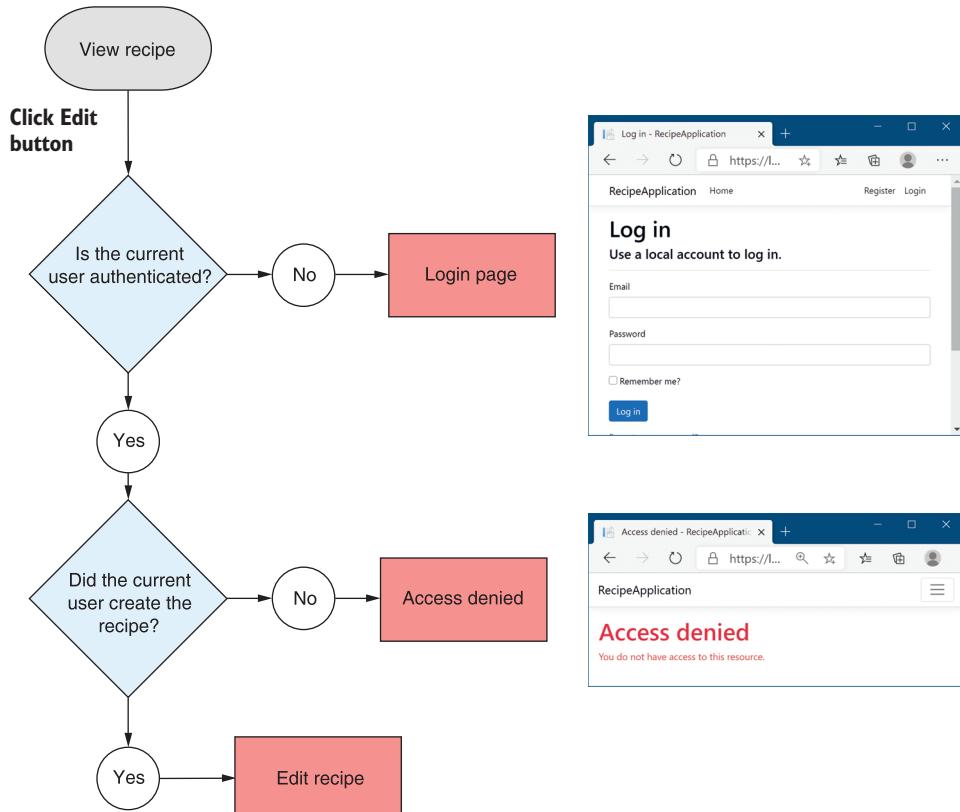
The other significant difference is that the `HandleRequirementAsync` method has provided the `Recipe` resource as a method argument. This is the same object that you provided when calling `AuthorizeAsync` on `IAuthorizationService`. You can use this resource to verify whether the current user created it. If so, you `Succeed()` the requirement; otherwise you do nothing.

The final task is to add `IRecipeOwnerHandler` to the DI container. Your handler uses an additional dependency, `UserManager<>`, which uses EF Core, so you should register the handler as a scoped service:

```
services.AddScoped<IAuthorizationHandler, IRecipeOwnerHandler>();
```

**TIP** If you’re wondering how to know whether you register a handler as scoped or a singleton, think back to chapter 10. Essentially, if you have scoped dependencies, you must register the handler as scoped; otherwise singleton is fine.

With everything hooked up, you can take the application for a spin. If you try to edit a recipe you didn’t create by clicking the Edit button on the recipe, you’ll either be redirected to the login page (if you hadn’t yet authenticated) or you’ll be presented with an “access denied” page, as shown in figure 15.7.



**Figure 15.7** If you’re logged in but not authorized to edit a recipe, you’ll be redirected to an “access denied” page. If you’re not logged in, you’ll be redirected to the login page.

By using resource-based authorization, you're able to enact more fine-grained authorization requirements that you can apply at the level of an individual document or resource. Instead of only being able to authorize that a user can edit *any* recipe, you can authorize whether a user can edit *this* recipe.

### Resource-based authorization versus business-logic checks

The value proposition of using the ASP.NET Core framework's resource-based authorization approach isn't always clear when compared to using simple, manual, business-logic based checks (as in listing 15.13). Using `IAuthorizationService` and the authorization infrastructure adds an explicit dependency on the ASP.NET Core framework that you may not want to use if you're performing authorization checks in your domain model services.

This is a valid concern without an easy answer. I tend to favor simple business-logic checks inside the domain, without relying on the framework's authorization infrastructure, to make my domain easier to test and framework-independent. But doing so loses some of the benefits of such a framework:

- The `IAuthorizationService` uses declarative policies, even though you are calling the authorization framework imperatively.
- You can decouple the need to authorize an action from the actual requirements.
- You can easily rely on peripheral services and properties of the request, which may be harder (or undesirable) with business logic checks.

You can achieve these benefits in business-logic checks, but that typically requires creating a lot of infrastructure too, so you lose a lot of the benefits of keeping things simple. Which approach is best will depend on the specifics of your application design, and there may well be cases for using both.

All the authorization techniques you've seen so far have focused on server-side checks. Both the `[Authorize]` attribute and resource-based authorization approaches focus on stopping users from executing a protected action on the server. This is important from a security point of view, but there's another aspect you should consider too: the user experience when they don't have permission.

You've protected the code executing on the server, but arguably the Edit button should never have been visible to the user if they weren't going to be allowed to edit the recipe! In the next section we'll look at how you can conditionally hide the Edit button by using resource-based authorization in your view models.

## 15.6 Hiding elements in Razor templates from unauthorized users

All the authorization code you've seen so far has revolved around protecting action methods or Razor Pages on the server side, rather than modifying the UI for users. This is important and should be the starting point whenever you add authorization to an app.

**WARNING** Malicious users can easily circumvent your UI, so it's important to always authorize your actions and Razor Pages on the server, never on the client alone.

From a user-experience point of view, however, it's not friendly to have buttons or links that look like they're available, but which present you with an "access denied" page when they're clicked. A better experience would be for the links to be disabled, or not visible at all.

You can achieve this in several ways in your own Razor templates. In this section, I'm going to show you how to add an additional property to the `PageModel`, called `CanEditRecipe`, which the Razor view template will use to change the rendered HTML.

**TIP** An alternative approach would be to inject `IAuthorizationService` directly into the view template using the `@inject` directive, as you saw in chapter 10, but you should prefer to keep logic like this in the page handler.

When you're finished, the rendered HTML will look unchanged for recipes you created, but the Edit button will be hidden when viewing a recipe someone else created, as shown in figure 15.8.

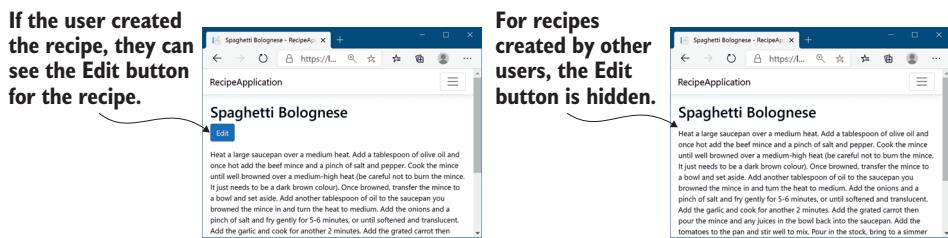


Figure 15.8 Although the HTML will appear unchanged for recipes you created, the Edit button is hidden when you view recipes created by a different user.

The following listing shows the `PageModel` for the `View.cshtml` Razor Page, which is used to render the recipe page shown in figure 15.8. As you've already seen for resource-based authorization, you can use the `IAuthorizationService` to determine whether the current user has permission to edit the Recipe by calling `AuthorizeAsync`. You can then set this value as an additional property on the `PageModel`, called `CanEditRecipe`.

#### Listing 15.16 Setting the `CanEditRecipe` property in the `View.cshtml` Razor Page

```
public class ViewModel : PageModel
{
    public Recipe Recipe { get; set; }
    public bool CanEditRecipe { get; set; }
```

The `CanEditRecipe` property will be used to control whether the Edit button is rendered.

```

private readonly RecipeService _service;
private readonly IAuthorizationService _authService;
public ViewModel(
    RecipeService service,
    IAuthorizationService authService)
{
    _service = service;
    _authService = authService;
}

public async Task<IActionResult> OnGetAsync(int id)
{
    Recipe = _service.GetRecipe(id);
    var isAuthorised = await _authService
        .AuthorizeAsync(User, recipe, "CanManageRecipe");
    CanEditRecipe = isAuthorised.Succeeded;
    return Page();
}

```

The diagram shows three annotations on the right side of the code:

- A callout points to the line `var isAuthorised = await _authService` with the text "Loads the Recipe resource for use with IAuthorizationService".
- A callout points to the line `.AuthorizeAsync(User, recipe, "CanManageRecipe");` with the text "Verifies whether the user is authorized to edit the Recipe".
- A callout points to the line `CanEditRecipe = isAuthorised.Succeeded;` with the text "Sets the CanEditRecipe property on the PageModel as appropriate".

Instead of blocking execution of the Razor Page (as you did previously in the `Edit.cshtml` page handler), use the result of the call to `AuthorizeAsync` to set the `CanEditRecipe` value on the `PageModel`. You can then make a simple change to the `View.cshtml` Razor template: add an `if` clause around the rendering of the `Edit` link.

```

@if(Model.CanEditRecipe)
{
    <a asp-page="Edit" asp-route-id="@Model.Id"
        class="btn btn-primary">Edit</a>
}

```

This ensures that only users who will be able to execute the `Edit.cshtml` Razor Page can see the link to that page.

**WARNING** The `if` clause means the `Edit` link will not be displayed unless the user created the recipe, but *a malicious user can still circumvent your UI*. It's important to keep the server-side authorization check in your `Edit.cshtml` page handler to protect against these circumvention attempts.

With that final change, you've finished adding authorization to the recipe application. Anonymous users can browse the recipes created by others, but they must log in to create new recipes. Additionally, authenticated users can only edit the recipes that they created, and they won't see an `Edit` link for other people's recipes.

Authorization is a key aspect of most apps, so it's important to bear it in mind from an early point. Although it's possible to add authorization later, as you did with the recipe app, it's normally preferable to consider authorization sooner rather than later in the app's development.

In the next chapter we're going to be looking at your ASP.NET Core application from a different point of view. Instead of focusing on the code and logic behind your app, we're going to look at how you prepare an app for production. You'll see how to

specify the URLs your application uses and how to publish an app so that it can be hosted in IIS. Finally, you'll learn about the bundling and minification of client-side assets, why you should care, and how to use `BundlerMinifier` in ASP.NET Core.

## Summary

- Authentication is the process of determining who a user is. It's distinct from authorization, the process of determining what a user can do. Authentication typically occurs before authorization.
- You can use the authorization services in any part of your application, but it's typically applied using the `AuthorizationMiddleware` by calling `UseAuthorization()`. This should be placed after the calls to `UseRouting()` and `UseAuthentication()`, and before the call to `UseEndpoints()` for correct operation.
- You can protect Razor Pages and MVC actions by applying the `[Authorize]` attribute. The routing middleware records the presence of the attribute as metadata with the selected endpoint. The authorization middleware uses this metadata to determine how to authorize the request.
- The simplest form of authorization requires that a user be authenticated before executing an action. You can achieve this by applying the `[Authorize]` attribute to a Razor Page, action, controller, or globally. You can also apply attributes conventionally to a subset of Razor Pages.
- Claims-based authorization uses the current user's claims to determine whether they're authorized to execute an action. You define the claims needed to execute an action in a *policy*.
- Policies have a name and are configured in `Startup.cs` as part of the call to `AddAuthorization()` in `ConfigureServices`. You define the policy using `AddPolicy()`, passing in a name and a lambda that defines the claims needed.
- You can apply a policy to an action or Razor Page by specifying the policy in the `authorize` attribute; for example, `[Authorize("CanAccessLounge")]`. This policy will be used by the `AuthorizationMiddleware` to determine if the user is allowed to execute the selected endpoint.
- In a Razor Pages app, if an unauthenticated user attempts to execute a protected action, they'll be redirected to the login page for your app. If they're already authenticated but don't have the required claims, they'll be shown an "access denied" page instead.
- For complex authorization policies, you can build a custom policy. A custom policy consists of one or more requirements, and a requirement can have one or more handlers. You can combine requirements and handlers to create policies of arbitrary complexity.
- For a policy to be authorized, every requirement must be satisfied. For a requirement to be satisfied, one or more of the associated handlers must indicate success, and none must indicate explicit failure.

- `AuthorizationHandler<T>` contains the logic that determines whether a requirement is satisfied. For example, if a requirement requires that users be over 18, the handler could look for a `DateOfBirth` claim and calculate the user's age.
- Handlers can mark a requirement as satisfied by calling `context.Succeed(requirement)`. If a handler can't satisfy the requirement, then it shouldn't call anything on the context, as a different handler could call `Succeed()` and satisfy the requirement.
- If a handler calls `context.Fail()`, the requirement will fail, even if a different handler marked it as a success using `Succeed()`. Only use this method if you want to override any calls to `Succeed()` from other handlers, to ensure the authorization policy will fail authorization.
- Resource-based authorization uses details of the resource being protected to determine whether the current user is authorized. For example, if a user is only allowed to edit their own documents, you need to know the author of the document before you can determine whether they're authorized.
- Resource-based authorization uses the same policy, requirements, and handler system as before. Instead of applying authorization with the `[Authorize]` attribute, you must manually call `IAuthorizationService` and provide the resource you're protecting.
- You can modify the user interface to account for user authorization by adding additional properties to your `PageModel`. If a user isn't authorized to execute an action, you can remove or disable the link to that action method in the UI. You should always authorize on the server, even if you've removed links from the UI.

# *Publishing and deploying your application*

## **This chapter covers**

- Publishing an ASP.NET Core application
- Hosting an ASP.NET Core application in IIS
- Customizing the URLs for an ASP.NET Core app
- Optimizing client-side assets with bundling and minification

We've covered a vast amount of ground so far in this book. We've gone over the basic mechanics of building an ASP.NET Core application, such as configuring dependency injection, loading app settings, and building a middleware pipeline. We've looked at the UI side, using Razor templates and layouts to build an HTML response. And we've looked at higher-level abstractions, such as EF Core and ASP.NET Core Identity, that let you interact with a database and add users to your application. In this chapter we're taking a slightly different route. Instead of looking at ways to build bigger and better applications, we'll focus on what it means to deploy your application so that users can access it.

We'll start by looking again at the ASP.NET Core hosting model in section 16.1 and examining why you might want to host your application behind a reverse proxy instead of exposing your app directly to the internet. I'll show you the difference

between running an ASP.NET Core app in development using `dotnet run` and publishing the app for use on a remote server. Finally, I'll describe some of the options available to you when deciding how and where to deploy your app.

In section 16.2, I'll show you how to deploy your app to one such option, a Windows server running IIS (Internet Information Services). This is a typical deployment scenario for many developers already familiar with ASP.NET, so it will act as a useful case study, but it's certainly not the only possibility. I won't go into all the technical details of configuring the venerable IIS system, but I'll show you the bare minimum required to get it up and running. If your focus is cross-platform development, then don't worry, I don't dwell on IIS for too long.

In section 16.3, I'll provide an introduction to hosting on Linux. You'll see how it differs from hosting applications on Windows, learn the changes you need to make to your apps, and find out about some gotchas to look out for. I'll describe how reverse proxies on Linux differ from IIS and point you to some resources you can use to configure your environments, rather than giving exhaustive instructions in this book.

If you're *not* hosting your application using IIS, you'll likely need to set the URL that your ASP.NET Core app is using when you deploy your application. In section 16.4 I'll show two approaches to this: using the special `ASPNETCORE_URLS` environment variable and using command-line arguments. Although generally not an issue during development, setting the correct URLs for your app is critical when you need to deploy it.

In the final section of this chapter, we'll look at a common optimization step used when deploying your application. Bundling and minification are used to reduce the number and size of requests that browsers must make to your app to fully load a page. I'll show you how to use a simple tool to create bundles when you build your application, and how to conditionally load these when in production to optimize your app's page size.

This chapter covers a relatively wide array of topics, all related to deploying your app. But before we get into the nitty-gritty, I'll go over the hosting model for ASP.NET Core so that we're on the same page. This is significantly different from the hosting model of the previous version of ASP.NET, so if you're coming from that background, it's best to try to forget what you know!

## 16.1 Understanding the ASP.NET Core hosting model

If you think back to chapter 1, you may remember that we discussed the hosting model of ASP.NET Core. ASP.NET Core applications are, essentially, console applications. They have a `static void Main` function that serves as the entry point for the application, like a standard .NET console app would.

What makes an app an ASP.NET Core app is that it runs a web server, typically Kestrel, inside the console app process. Kestrel provides the HTTP functionality to receive requests and return responses to clients. Kestrel passes any requests it receives to the body of your application to generate a response, as shown in figure 16.1. This

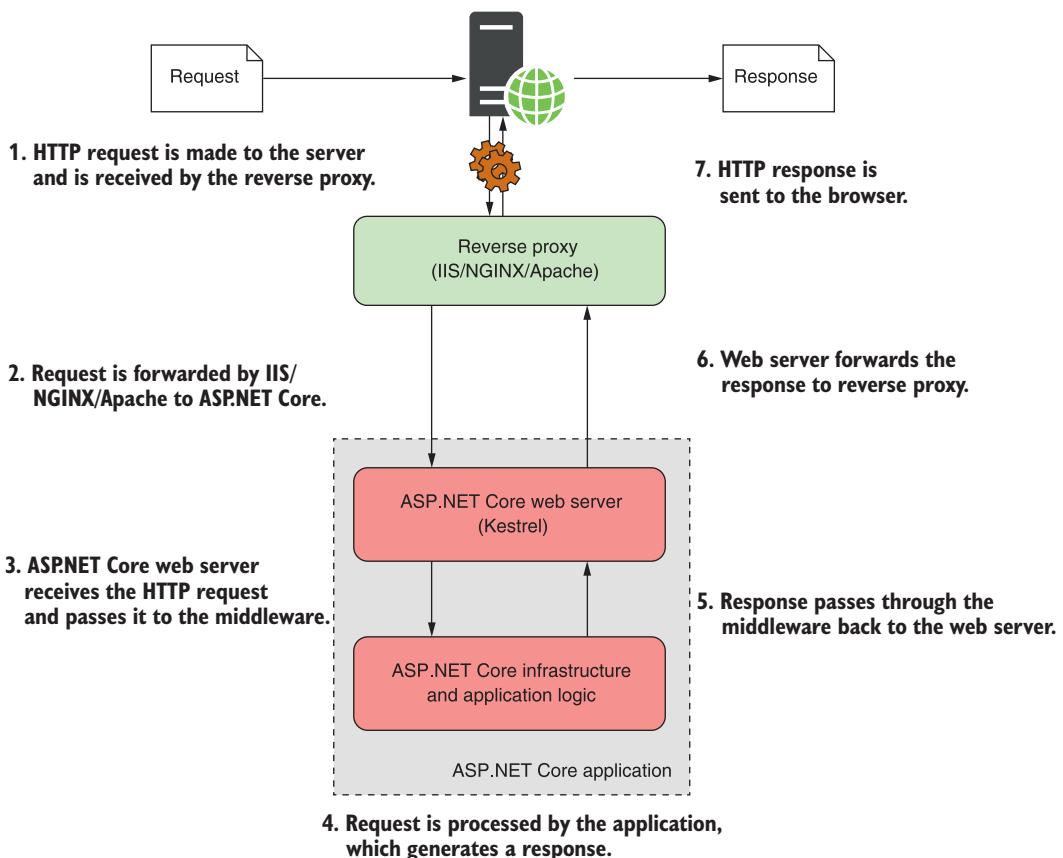


Figure 16.1 The hosting model for ASP.NET Core. Requests are received by the reverse proxy and are forwarded to the Kestrel web server. The same application can run behind various reverse proxies without modification.

hosting model decouples the server and reverse proxy from the application itself, so that the same application can run unchanged in multiple environments.

In this book we've focused on the lower half of figure 16.1—the ASP.NET Core application itself—but the reality is that you'll often want to place your ASP.NET Core apps behind a reverse proxy, such as IIS on Windows, or NGINX or Apache on Linux. The *reverse proxy* is the program that listens for HTTP requests from the internet and then makes requests to your app as though the request had come from the internet directly.

**DEFINITION** A *reverse proxy* is software that's 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.

If you’re running your application using a Platform as a Service (PaaS) offering, such as Azure App Service, you’re using a reverse proxy there too—one that is managed by Azure. Using a reverse proxy has many benefits:

- *Security*—Reverse proxies are specifically designed to be exposed to malicious internet traffic, so they’re typically well-tested and battle-hardened.
- *Performance*—You can configure reverse proxies to provide performance improvements by aggressively caching responses to requests.
- *Process management*—An unfortunate reality is that apps sometimes crash. Some reverse proxies can act as monitors/schedulers to ensure that if an app crashes, the proxy can automatically restart it.
- *Support for multiple apps*—It’s common to have multiple apps running on a single server. Using a reverse proxy makes it easier to support this scenario by using the host name of a request to decide which app should receive the request.

I don’t want to make it seem like using a reverse proxy is all sunshine and roses. There are some downsides:

- *Complexity*—One of the biggest complaints is how complex reverse proxies can be. If you’re managing the proxy yourself (as opposed to relying on a PaaS implementation), there can be lots of proxy-specific pitfalls to look out for.
- *Inter-process communication*—Most reverse proxies require two processes: a reverse proxy and your web app. Communicating between the two is often slower than if you directly exposed your web app to requests from the internet.
- *Restricted features*—Not all reverse proxies support all the same features as an ASP.NET Core app. For example, Kestrel supports HTTP/2, but if your reverse proxy doesn’t, you won’t see the benefits.

Whether you choose to use a reverse proxy or not, when the time comes to host your app, you can’t copy your code files directly to the server. First you need to *publish* your ASP.NET Core app, to optimize it for production. In section 16.1.1, we’ll look at building an ASP.NET Core app so it can be run on your development machine, compared to publishing it so that it can be run on a server.

### 16.1.1 Running vs. publishing an ASP.NET Core app

One of the key changes in ASP.NET Core from previous versions of ASP.NET is making it easy to build apps using your favorite code editors and IDEs. Previously, Visual Studio was required for ASP.NET development, but with the .NET CLI and the OmniSharp plugin you can now build apps with the tools you’re comfortable with, on any platform.

As a result, whether you build using Visual Studio or the .NET CLI, the same tools are being used under the hood. Visual Studio provides an additional GUI, functionality, and wrappers for building your app, but it executes the same commands as the .NET CLI behind the scenes.

As a refresher, you've used four main .NET CLI commands so far to build your apps:

- `dotnet new`—Creates an ASP.NET Core application from a template
- `dotnet restore`—Downloads and installs any referenced NuGet packages for your project
- `dotnet build`—Compiles and builds your project
- `dotnet run`—Executes your app, so you can send requests to it

If you've ever built a .NET application, whether it's an ASP.NET app or a .NET Framework console app, you'll know that the output of the build process is written to the bin folder. The same is true for ASP.NET Core applications.

If your project compiles successfully when you call `dotnet build`, the .NET CLI will write its output to a bin folder in your project's directory. Inside this bin folder are several files required to run your app, including a .dll file that contains the code for your application. Figure 16.2 shows part of the output of the bin folder for an ASP.NET Core application.



Figure 16.2 The bin folder for an ASP.NET Core app after running `dotnet build`. The application is compiled into a single .dll file, `ExampleApp.dll`.

**NOTE** On Windows you will also have an executable .exe file, `ExampleApp.exe`. This is a simple wrapper file for convenience that makes it easier to run the application contained in `ExampleApp.dll`.

When you call `dotnet run` in your project folder (or run your application using Visual Studio), the .NET CLI uses the .dll to run your application. But this doesn't contain everything you need to deploy your app.

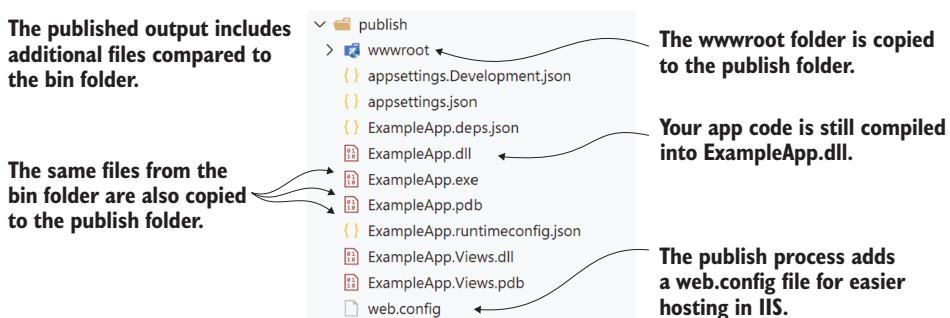
To host and deploy your app on a server, you first need to *publish* it. You can publish your ASP.NET Core app from the command line using the `dotnet publish` command. This builds and packages everything your app needs to run. The following command packages the app from the current directory and builds it to a subfolder

called publish. I've used the Release configuration, instead of the default Debug configuration, so that the output will be fully optimized for running in production:

```
dotnet publish --output publish --configuration Release
```

**TIP** Always use the release configuration when publishing your app for deployment. This ensures the compiler generates optimized code for your app.

Once the command completes, you'll find your published application in the publish folder, as shown in figure 16.3.



**Figure 16.3** The publish folder for the app after running `dotnet publish`. The app is still compiled into a single .dll file, but all the additional files, such as `wwwroot` and `appsettings.json`, are also copied to the output.

As you can see, the `ExampleApp.dll` file is still there, along with some additional files. Most notably, the publish process has copied across the `wwwroot` folder of static files. When running your application locally with `dotnet run`, the .NET CLI uses these files from your application's project folder directly. Running `dotnet publish` copies the files to the output directory, so they're included when you deploy your app to a server.

If your first instinct is to try running the application in the publish folder using the `dotnet run` command you already know and love, then you'll be disappointed. Instead of the application starting up, you'll be presented with a somewhat confusing message: "Couldn't find a project to run."

To run a published application, you need to use a slightly different command. Instead of calling `dotnet run`, you must pass the path to your application's .dll file to the `dotnet` command. If you're running the command from the publish folder, then for the example app in figure 16.3, that would look something like

```
dotnet ExampleApp.dll
```

This is the command that your server will run when running your application in production.

When you're developing, the dotnet run command does a whole load of work to make things easier on you: it makes sure your application is built, looks for a project file in the current folder, works out where the corresponding .dlls will be (in the bin folder), and finally, runs your app.

In production, you don't need any of this extra work. Your app is already built; it only needs to be run. The dotnet <dll> syntax does this alone, so your app starts much faster.

**NOTE** The dotnet command used to run your published application is part of the .NET Runtime, whereas the dotnet command used to build and run your application during development is part of the .NET SDK.

### Framework-dependent deployments vs. self-contained deployments

.NET Core applications can be deployed in two different ways: runtime-dependent deployments (RDD) and self-contained deployments (SCD).

Most of the time, you'll use an RDD. This relies on the .NET 5.0 runtime being installed on the target machine that runs your published app, but you can run your app on any platform—Windows, Linux, or macOS—withouth having to recompile.

In contrast, an SCD contains *all* the code required to run your app, so the target machine doesn't need to have .NET 5.0 installed. Instead, publishing your app will package up the .NET 5.0 runtime with your app's code and libraries.

Each approach has its pros and cons, but in most cases I tend to create RDDs. The final size of RDDs is much smaller, as they only contain your app code, instead of the whole .NET 5.0 framework, which SCDs contain. Also, you can deploy your RDD apps to any platform, whereas SCDs must be compiled specifically for the target machine's operating system, such as Windows 10 64-bit or Red Hat Enterprise Linux 64bit.

That said, SCD deployments are excellent for isolating your application from dependencies on the hosting machine. SCD deployments don't rely on the version of .NET installed on a hosting provider, so you can, for example, use preview versions of .NET in Azure App Service without needing the preview version to be supported.

In this book, I only discuss RDDs for simplicity, but if you want to create an SCD, just provide a runtime identifier, in this case Windows 10 64-bit, when you publish your app:

```
dotnet publish -c Release -r win10-x64 -o publish_folder
```

The output will contain an .exe file, which is your application, and a ton of .dlls (about 65 MB of .dlls for a sample app), which are the .NET 5.0 framework. You need to deploy this whole folder to the target machine to run your app. In .NET 5.0 it's possible to trim some of these assemblies during the publish process, but this comes with risks in some scenarios. For more details, see Microsoft's ".NET Core application publishing overview" documentation at <https://docs.microsoft.com/dotnet/core/deploying/>.

We've established that publishing your app is important for preparing it to run in production, but how do you go about deploying it? How do you get the files from your computer onto a server so that people can access your app? You have many, many options for this, so in the next section I'll give you a brief list of approaches to consider.

### 16.1.2 Choosing a deployment method for your application

To deploy any application to production, you generally have two fundamental requirements:

- A server that can run your app
- A means of loading your app onto the server

Historically, putting an app into production was a laborious and error-prone process. For many people, this is still true. If you're working at a company that hasn't changed practices in recent years, you may need to request a server or virtual machine for your app and provide your application to an operations team who will install it for you. If that's the case, you may have your hands tied regarding how you deploy.

For those who have embraced continuous integration (CI) or continuous delivery/deployment (CD), there are many more possibilities. CI/CD is the process of detecting changes in your version control system (for example, Git, SVN, Mercurial, Team Foundation Version Control) and automatically building, and potentially deploying, your application to a server with little to no human intervention.<sup>1</sup>

You can find many different CI/CD systems out there: Azure DevOps, GitHub Actions, Jenkins, TeamCity, AppVeyor, Travis, and Octopus Deploy, to name a few. Each can manage some or all of the CI/CD process and can integrate with many different systems.

Rather than pushing any particular system, I suggest trying out some of the services available and seeing which works best for you. Some are better suited to open source projects, some are better when you're deploying to cloud services—it all depends on your particular situation.

If you're getting started with ASP.NET Core and don't want to have to go through the setup process of getting CI working, you still have lots of options. The easiest way to get started with Visual Studio is to use the built-in deployment options. These are available from Visual Studio via the Build > Publish *AppName* menu option, which presents you with the screen shown in figure 16.4.

From here, you can publish your application directly from Visual Studio to many different locations.<sup>2</sup> This is great when you're getting started, though I recommend looking at a more automated and controlled approach for larger applications, or when you have a whole team working on a single app.

---

<sup>1</sup> There are important but subtle differences between these terms. Atlassian has a good comparison article, "Continuous integration vs. continuous delivery vs. continuous deployment," here: <http://mng.bz/vzp4>.

<sup>2</sup> For guidance on choosing your Visual Studio publishing options, see Microsoft's "Deploy your app to a folder, IIS, Azure, or another destination" documentation: <http://mng.bz/4Z8j>.

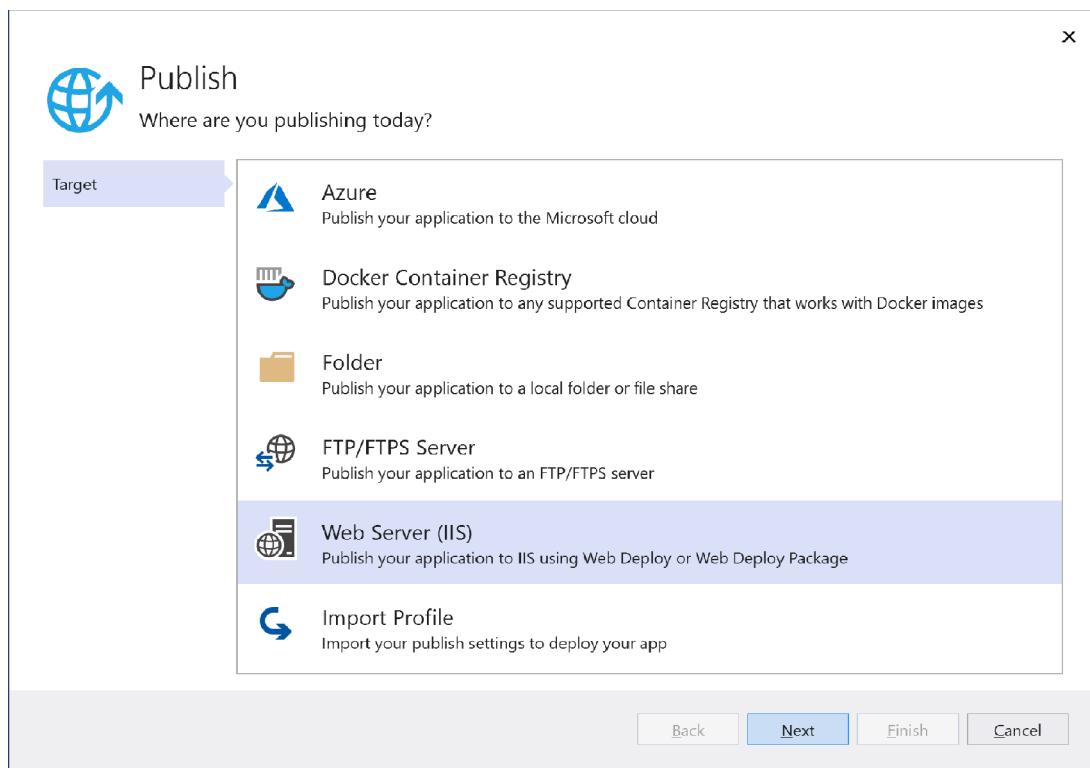


Figure 16.4 The Publish application screen in Visual Studio 2019. This provides easy options for publishing your application directly to Azure App Service, to IIS, to an FTP site, or to a folder on the local machine.

Given the number of possibilities available in this space, and the speed with which these options change, I'm going to focus on one specific scenario in this chapter: You've built an ASP.NET Core application and you need to deploy it. You have access to a Windows server that's already serving (previous version) ASP.NET applications using IIS, and you want to run your ASP.NET Core app alongside them.

In the next section, you'll see an overview of the steps required to run an ASP.NET Core application in production, using IIS as a reverse proxy. It won't be a master class in configuring IIS (there's so much depth to the 20-year-old product that I wouldn't know where to start!), but I'll cover the basics needed to get your application serving requests.

## 16.2 Publishing your app to IIS

In this section I'll briefly show you how to publish your first app to IIS. You'll add an application pool and website to IIS and ensure your app has the necessary configuration to work with IIS as a reverse proxy. The deployment itself will be as simple as copying your published app to IIS's hosting folder.

In section 16.1 you learned about the need to publish an app before you deploy it, and the benefits of using a reverse proxy when you run an ASP.NET Core app in production. If you’re deploying your application to Windows, IIS will be your reverse proxy and will be responsible for managing your application.

IIS is an old and complex beast, and I can’t possibly cover everything related to configuring it in this book. Nor would you want me to—it would be very boring! Instead, in this section I’ll provide an overview of the basic requirements for running ASP.NET Core behind IIS, along with the changes you may need to make to your application to support IIS.

If you’re on Windows and want to try out these steps locally, you’ll need to manually enable IIS on your development machine. If you’ve done this with older versions of Windows, nothing much has changed. You can find a step-by-step guide to configuring IIS and troubleshooting tips in the ASP.NET Core documentation at <http://mng.bz/6g2R>.

### 16.2.1 Configuring IIS for ASP.NET Core

The first step in preparing IIS to host ASP.NET Core applications is to install the ASP.NET Core Windows Hosting Bundle.<sup>3</sup> This includes several components needed to run .NET apps:

- *The .NET Runtime*—Runs your .NET 5.0 application
- *The ASP.NET Core Runtime*—Required to run ASP.NET Core apps
- *The IIS AspNetCore Module*—Provides the link between IIS and your app, so that IIS can act as a reverse proxy

If you’re going to be running IIS on your development machine, make sure you install the bundle as well, or you’ll get strange errors from IIS.

**TIP** The Windows Hosting Bundle provides everything you need for running ASP.NET Core behind IIS on Windows. If you’re hosting your application on Linux or Mac, or aren’t using IIS on Windows, you’ll need to install the .NET Runtime and ASP.NET Core Runtime to run runtime-dependent ASP.NET Core apps.

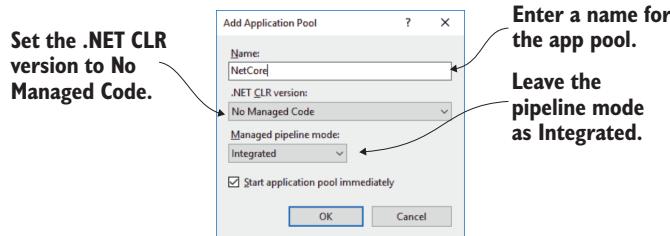
Once you’ve installed the bundle, you need to configure an *application pool* in IIS for your ASP.NET Core apps. Previous versions of ASP.NET would run in a *managed* app pool that used .NET Framework, but for ASP.NET Core you should create a *No Managed Code* pool. The native ASP.NET Core Module runs inside the pool, which boots the .NET 5.0 Runtime itself.

**DEFINITION** An *application pool* in IIS represents an application process. You can run each app in IIS in a separate application pool to keep them isolated from one another.

---

<sup>3</sup> You can download the ASP.NET Core Windows Hosting Bundle from <http://mng.bz/opED>.

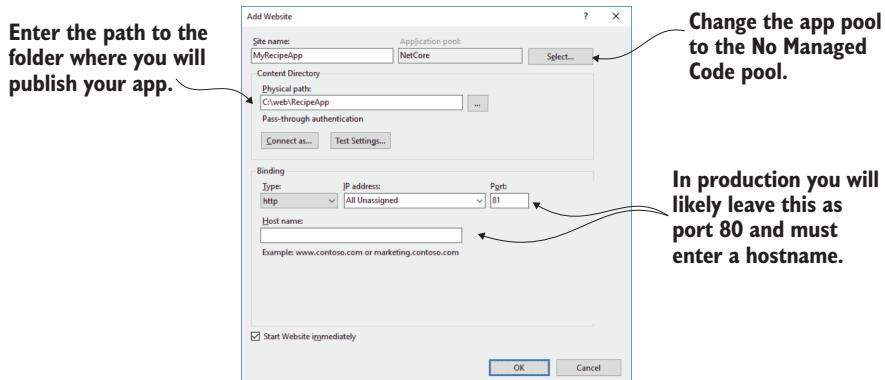
To create an unmanaged application pool, right-click Application Pools in IIS and choose Add Application Pool. Provide a name for the app pool in the resulting dialog box, such as NetCore, and set the .NET CLR version to No Managed Code, as shown in figure 16.5.



**Figure 16.5** Creating an app pool in IIS for your ASP.NET Core app. The .NET CLR version should be set to No Managed Code.

Now that you have an app pool, you can add a new website to IIS. Right-click the Sites node and choose Add Website. In the Add Website dialog box, shown in figure 16.6, you provide a name for the website and the path to the folder where you'll publish your website. I've created a folder that I'll use to deploy the Recipe app from previous chapters. It's important to change the Application Pool for the app to the new NetCore app pool you created. In production, you'd also provide a hostname for the application, but I've left it blank for now and changed the port to 81, so the application will bind to the URL `http://localhost:81`.

**NOTE** When you deploy an application to production, you need to register a host name with a domain registrar, so your site is accessible by people on the internet. Use that hostname when configuring your application in IIS, as shown in figure 16.6.



**Figure 16.6** Adding a new website to IIS for your app. Be sure to change the Application Pool to the No Managed Code pool created in the previous step. You also provide a name, the path where you'll publish your app files, and the URL that IIS will use for your app.

Once you click OK, IIS creates the application and attempts to start it. But you haven't published your app to the folder, so you won't be able to view it in a browser yet.

You need to carry out one more critical setup step before you can publish and run your app: you must grant permissions for the NetCore app pool to access the path where you'll publish your app. To do this, right-click the folder that will host your app in Windows File Explorer and choose Properties. In the Properties dialog box, choose Security > Edit > Add. Enter `IIS AppPool\NetCore` in the text box, as shown in figure 16.7, where NetCore is the name of your app pool, and click OK. Close all the dialog boxes by clicking OK, and you're all set.

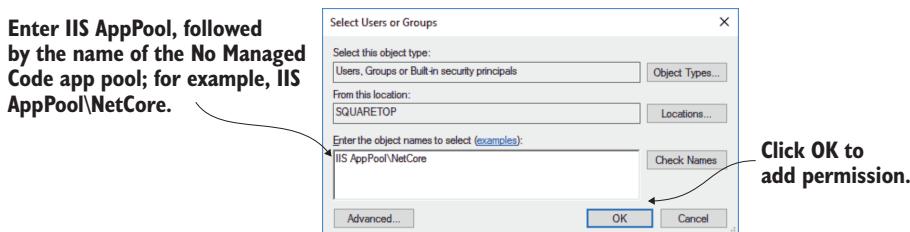


Figure 16.7 Adding permission for the NetCore app pool to the website's publish folder.

Out of the box, the ASP.NET Core templates are configured to work seamlessly with IIS, but if you've created an app from scratch, you may need to make a couple of changes. In the next section I'll briefly show the changes you need to make and explain why they're necessary.

### 16.2.2 Preparing and publishing your application to IIS

As I discussed in section 16.1, IIS acts as a reverse proxy for your ASP.NET Core app. That means IIS needs to be able to communicate directly with your app to forward incoming requests to and outgoing responses from your app.

IIS handles this with the ASP.NET Core Module, but there's a certain degree of negotiation required between IIS and your app. For this to work correctly, you need to configure your app to use IIS integration.

IIS integration is added by default when you use the `IHostBuilder.ConfigureWebHostDefaults()` helper method used in the default templates. If you're customizing your own HostBuilder, you need to ensure you add IIS integration with the `UseIIS()` or `UseIISIntegration()` extension method.

#### **Listing 16.1 Adding IIS Integration to a host builder**

```
public class Program
{
    public static void Main(string[] args)
    {
```

```

        CreateHostBuilder(args).Build().Run();
    }
    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureWebHost(webBuilder => {
                webBuilder.UseKestrel();
                webBuilder.UseStartup<Startup>();
                webBuilder.UseIIS();
                webBuilder.UseIISIntegration();
            });
}

```

**NOTE** If you’re not using your application with IIS, the `UseIIS()` and `UseIISIntegration()` methods will have no effect on your app, so it’s safe to include them anyway.

### In-process vs. out-of-process hosting in IIS

The reverse-proxy description I gave in section 16.1 assumes that your application is running in a separate process from the reverse proxy itself. That is the case if you’re running on Linux and was the default for IIS up until ASP.NET Core 3.0.

In ASP.NET Core 3.0, ASP.NET Core switched to using an *in-process* hosting model by default for applications deployed to IIS. In this model, IIS hosts your application directly in the IIS process, reducing inter-process communication and boosting performance.

You can switch to the out-of-process hosting model with IIS if you wish, which can sometimes be useful for troubleshooting problems. Rick Strahl has an excellent post on the differences between the hosting models, how to switch between them, and the advantages of each: “ASP.NET Core In Process Hosting on IIS with ASP.NET Core” at <http://mng.bz/QmEv>.

In general, you shouldn’t need to worry about the differences between the hosting models, but it’s something to be aware of if you’re deploying to IIS. If you choose to use the out-of-process hosting model, you should use the `UseIISIntegration()` extension method. If you use the in-process model, use `UseIIS()`. Alternatively, play it safe and use both—the correct extension method will be activated based on the hosting model used in production. Neither extension does anything if you don’t use IIS.

When running behind IIS, these extension methods configure your app to pair with IIS so that it can seamlessly accept requests. Among other things, the extensions do the following:

- Define the URL that IIS will use to forward requests to your app and configure your app to listen on this URL
- Configure your app to interpret requests coming from IIS as coming from the client by setting up header forwarding
- Enable Windows authentication if required

Adding the IIS extension methods is the only change you need to make to your application to be able to host in IIS, but there's one additional aspect to be aware of when you publish your app.

As with previous versions of ASP.NET, IIS relies on a web.config file to configure the applications it runs. It's important that your application include a web.config file when it's published to IIS; otherwise you could get broken behavior or even expose files that shouldn't be exposed.<sup>4</sup>

If your ASP.NET Core project already includes a web.config file, the .NET CLI or Visual Studio will copy it to the publish directory when you publish your app. If your app doesn't include a web.config file, the publish command will create the correct one for you. If you don't need to customize the web.config file, it's generally best not to include one in your project and let the CLI create the correct file for you.

With these changes, you're finally in a position to publish your application to IIS. Publish your ASP.NET Core app to a folder, either from Visual Studio or with the .NET CLI, by running

```
dotnet publish --output publish_folder --configuration Release
```

This will publish your application to the publish\_folder folder. You can then copy your application to the path specified in IIS, as shown in figure 16.6. At this point, if all has gone smoothly, you should be able to navigate to the URL you specified for your app (<http://localhost:81>, in my case) and see it running, as shown in figure 16.8.

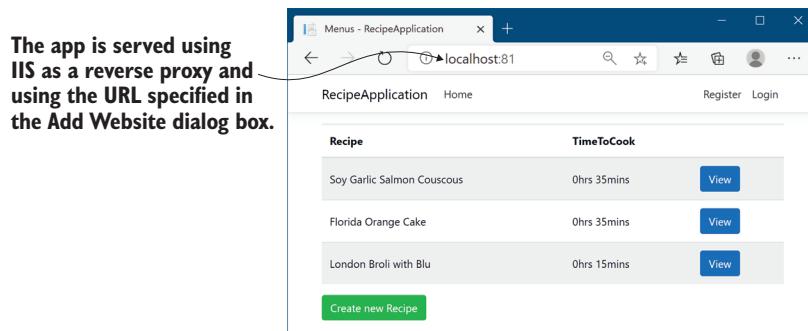


Figure 16.8 The published application, using IIS as a reverse proxy listening at the URL <http://localhost:81>.

And there you have it, your first application running behind a reverse proxy. Even though ASP.NET Core uses a different hosting model than previous versions of ASP.NET, the process of configuring IIS is similar.

<sup>4</sup> For details on using web.config to customize the IIS AspNetCore Module, see Microsoft's "ASP.NET Core Module" documentation: <http://mng.bz/Xdna>.

As is often the case when it comes to deployment, the success you have is highly dependent on your precise environment and your app itself. If, after following these steps, you find you can't get your application to start, I highly recommend checking out the documentation at <https://docs.microsoft.com/aspnet/core/publishing/iis>. This contains many troubleshooting steps to get you back on track if IIS decides to throw a hissy fit.

This section was deliberately tailored to deploying to IIS, as it provides a great segue for developers who are already used to deploying ASP.NET apps and want to deploy their first ASP.NET Core app. But that's not to say that IIS is the only, or best, place to host your application.

In the next section I'll provide a brief introduction to hosting your app on Linux, behind a reverse proxy like NGINX or Apache. I won't go into configuration of the reverse proxy itself, but I will provide an overview of things to consider and resources you can use to run your applications on Linux.

## 16.3 **Hosting an application on Linux**

One of the great new features in ASP.NET Core is the ability to develop and deploy applications cross-platform, whether that be on Windows, Linux, or macOS. The ability to run on Linux, in particular, opens up the possibility of cheaper deployments to cloud hosting, deploying to small devices like a Raspberry Pi or to Docker containers.

One of the characteristics of Linux is that it's almost infinitely configurable. Although that's definitely a feature, it can also be extremely daunting, especially if you're coming from the Windows world of wizards and GUIs. This section provides an overview of what it takes to run an application on Linux. It focuses on the broad steps you need to take, rather than the somewhat tedious details of the configuration itself. Instead, I'll point to resources you can refer to as necessary.

### 16.3.1 **Running an ASP.NET Core app behind a reverse proxy on Linux**

You'll be glad to hear that running your application on Linux is broadly the same as running your application on Windows with IIS:

- 1 *Publish your app using dotnet publish.* If you're creating an RDD, the output is the same as you'd use with IIS. For an SCD, you must provide the target platform moniker, as described in section 16.1.1.
- 2 *Install the necessary prerequisites on the server.* For an RDD deployment, you must install the .NET 5.0 Runtime and the necessary prerequisites. You can find details on this in Microsoft's "Install .NET on Linux" documentation at <https://docs.microsoft.com/en-gb/dotnet/core/install/linux>.
- 3 *Copy your app to the server.* You can use any mechanism you like: FTP, USB stick, whatever you need to get your files onto the server!
- 4 *Configure a reverse proxy and point it to your app.* As you know by now, you may want to run your app behind a reverse proxy, for the reasons described in section 16.1.

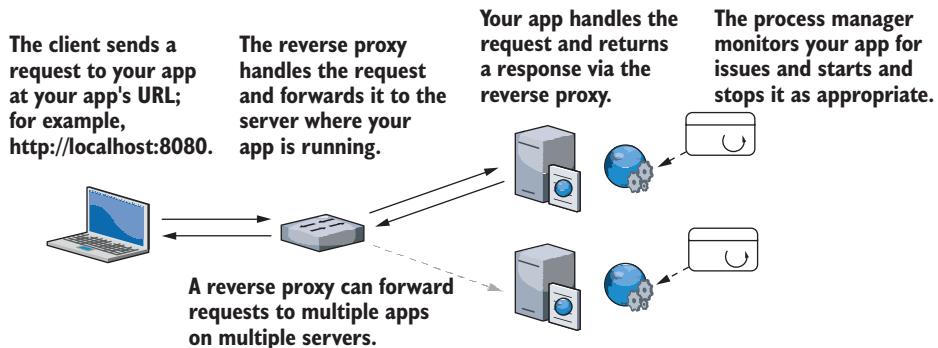
On Windows, you'd use IIS, but on Linux you have more options. NGINX, Apache, and HAProxy are all commonly used options.

- 5 *Configure a process-management tool for your app.* On Windows, IIS acts both as a reverse proxy and a process manager, restarting your app if it crashes or stops responding. On Linux you typically need to configure a separate process manager to handle these duties; the reverse proxies won't do it for you.

The first three points are generally the same, whether you're running on Windows with IIS or on Linux, but the last two points are more interesting. In contrast to the monolithic IIS, Linux has a philosophy of small applications, each with a single responsibility.

IIS runs on the same server as your app and takes on multiple duties—proxying traffic from the internet to your app, but also monitoring the app process itself. If your app crashes or stops responding, IIS will restart the process to ensure you can keep handling requests.

In Linux, the reverse proxy might be running on the same server as your app, but it's also common for it to be running on a different server entirely, as shown in figure 16.9. This is similarly true if you choose to deploy your app to Docker; your app would typically be deployed in a container without a reverse proxy, and a reverse proxy on a server would point to your Docker container.



**Figure 16.9** On Linux it's common for a reverse proxy to be on a different server from your app. The reverse proxy forwards incoming requests to your app, while a process manager, such as systemd, monitors your apps for crashes and restarts it as appropriate.

As the reverse proxies aren't necessarily on the same server as your app, they can't be used to restart your app if it crashes. Instead, you need to use a process manager such as systemd to monitor your app. If you're using Docker, you'd typically use a container orchestrator such as Kubernetes (<https://kubernetes.io>) to monitor the health of your containers.

## Running ASP.NET Core applications in Docker

Docker is the most commonly used engine for *containerizing* your applications. A container is like a small, lightweight virtual machine, specific to your app. It contains an operating system, your app, and any dependencies for your app. This container can then be run on any machine that runs Docker, and your app will run exactly the same, regardless of the host operating system and what's installed on it. This makes deployments highly repeatable: you can be confident that if the container runs on your machine, it will run on the server too.

ASP.NET Core is well-suited to container deployments, but moving to Docker involves a big shift in your deployment methodology and may or may not be right for you and your apps. If you're interested in the possibilities afforded by Docker and want to learn more, I suggest checking out the following resources:

- *Docker in Practice*, 2nd ed., by Ian Miell and Aidan Hobson Sayers (Manning, 2019) provides a vast array of practical techniques to help you get the most out of Docker (<http://mng.bz/nM8d>).
- Even if you're not deploying to Linux, you can use Docker with Docker for Windows. Check out the free e-book, *Introduction to Windows Containers* by John McCabe and Michael Friis (Microsoft Press, 2017) from <https://aka.ms/containersebook>.
- You can find a lot of details on building and running your ASP.NET Core applications on Docker in the .NET documentation at <http://mng.bz/vz5a>.
- Steve Gordon has an excellent blog post series on Docker for ASP.NET Core developers at <https://www.stevejgordon.co.uk/docker-dotnet-developers>.

Configuring these systems is a laborious task that makes for dry reading, so I won't detail them here. Instead, I recommend checking out the ASP.NET Core docs. They have a guide for NGINX and systemd, "Host ASP.NET Core on Linux with Nginx" (<http://mng.bz/yYGd>), and a guide for configuring Apache with systemd, "Host ASP.NET Core on Linux with Apache" (<http://mng.bz/MXVB>).

Both guides cover the basic configuration of the respective reverse proxies and systemd supervisors, but more importantly, they also show how to configure them *securely*. The reverse proxy sits between your app and the unfettered internet, so it's important to get it right!

Configuring the reverse proxy and the process manager is typically the most complex part of deploying to Linux, and that isn't specific to .NET development: the same would be true if you were deploying a Node.js web app. But you need to consider a few things *inside* your application when you're going to be deploying to Linux, as you'll see in the next section.

### 16.3.2 Preparing your app for deployment to Linux

Generally speaking, your app doesn't care which reverse proxy it sits behind, whether it's NGINX, Apache, or IIS—your app receives requests and responds to them without the reverse proxy affecting things. When you're hosting behind IIS, you need to add `UseIISIntegration()`; similarly, when you're hosting on Linux, you need to add an extension method to your app setup.

When a request arrives at the reverse proxy, it contains some information that is lost after the request is forwarded to your app. For example, the original request comes with the IP address of the client/browser connecting to your app: once the request is forwarded from the reverse proxy, the IP address is that of the *reverse proxy*, not the *browser*. Also, if the reverse proxy is used for SSL offloading (see chapter 18), then a request that was originally made using HTTPS may arrive at your app as an HTTP request.

The standard solution to these issues is for the reverse proxy to add additional headers before forwarding requests to your app. For example, the `X-Forwarded-For` header identifies the original client's IP address, while the `X-Forwarded-Proto` header indicates the original scheme of the request (`http` or `https`).

For your app to behave correctly, it needs to look for these headers in incoming requests and modify the request as appropriate. A request to `http://localhost` with the `X-Forwarded-Proto` header set to `https` should be treated the same as if the request was to `https://localhost`.

You can use `ForwardedHeadersMiddleware` in your middleware pipeline to achieve this. This middleware overrides `Request.Scheme` and other properties on `HttpContext` to correspond to the forwarded headers. If you're using the default `Host.CreateDefaultBuilder()` method in `Program.cs`, this is partially handled for you—the middleware is automatically added to the pipeline in a disabled state. To enable it, set the environment variable `ASPNETCORE_FORWARDEDHEADERS_ENABLED=true`.

If you're using your own `HostBuilder` instance, instead of the default builder, you can add the middleware to the start of your middleware pipeline manually, as shown in listing 16.2, and configure it with the headers to look for.

**WARNING** It's important that `ForwardedHeadersMiddleware` be placed early in the middleware pipeline to correct `Request.Scheme` before any middleware that depends on the scheme runs.

#### Listing 16.2 Configuring an app to use forwarded headers in `Startup.cs`

```
public class Startup
{
    public class Configure(IApplicationBuilder app)
    {
        app.UseForwardedHeaders(new ForwardedHeadersOptions
        {
            ForwardedHeaders = ForwardedHeaders.XForwardedFor |
                               ForwardedHeaders.XForwardedProto
        });
    }
}
```

Configures the headers the middleware should look for and use

Adds `ForwardedHeadersMiddleware` early in your pipeline

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

    app.UseAuthentication();
    app.UseMvc();
}

}
```

The forwarded  
headers middleware  
must be placed before  
all other middleware.

**NOTE** This behavior isn't specific to reverse proxies on Linux; the `UseIis()` extension adds `ForwardedHeadersMiddleware` under the hood as part of its configuration when your app is running behind IIS.

Aside from considering the forwarded headers, you need to consider a few minor things when deploying your app to Linux that may trip you up if you're used to deploying to Windows alone:

- *Line endings (LF on Linux versus CRLF on Windows)*—Windows and Linux use different character codes in text to indicate the end of a line. This isn't often an issue for ASP.NET Core apps, but if you're writing text files on one platform and reading them on a different platform, it's something to bear in mind.
- *Path directory separator ("\" on Windows, "/" on Linux)*—This is one of the most common bugs I see when Windows developers move to Linux. Each platform uses a different separator in file paths, so while loading a file using the "subdir\myfile.json" path will work fine on Windows, it won't on Linux. Instead, you should use `Path.Combine` to create the appropriate separator for the current platform; for example, `Path.Combine("subdir", "myfile.json")`.
- *Environment variables can't contain ":"*—On some Linux distributions, the colon character (:) isn't allowed in environment variables. As you saw in chapter 11, this character is typically used to denote different sections in ASP.NET Core configuration, so you often need to use it in environment variables. Instead, you can use a double underscore in your environment variables (`__`) and ASP.NET Core will treat it the same as if you'd used a colon.
- *Time zone and culture data may be missing*—Linux distributions don't always come with time zone or culture data, which can cause localization issues and exceptions at runtime. You can install the time zone data using your distribution's package manager.<sup>5</sup>
- *Directory structures are different*—Linux distributions use a completely different folder structure than Windows, so you need to bear that in mind if your app hard-codes paths. In particular, consider differences in temporary/cache folders.

The preceding list is not exhaustive by any means, but as long as you set up `ForwardedHeadersMiddleware` and take care to use cross-platform constructs like `Path.Combine`,

<sup>5</sup> I ran into this problem myself. You can read about the issue in detail and how I solved it on my blog: <http://mng.bz/aoem>.

you shouldn't have too many problems running your applications on Linux. But configuring a reverse proxy isn't the simplest of activities, so wherever you're planning on hosting your app, I suggest checking the documentation for guidance at <https://docs.microsoft.com/aspnet/core/publishing>.

## 16.4 Configuring the URLs for your application

At this point, you've deployed an application, but there's one aspect you haven't configured: the URLs for your application. When you're using IIS as a reverse proxy, you don't have to worry about this inside your app. IIS integration with the ASP.NET Core Module works by dynamically creating a URL that's used to forward requests between IIS and your app. The hostname you configure in IIS (in figure 16.6) is the URL that external users see for your app; the internal URL that IIS uses when forwarding requests is never exposed.

If you're not using IIS as a reverse proxy—maybe you're using NGINX or exposing your app directly to the internet—you may find you need to configure the URLs your application listens to directly.

By default, ASP.NET Core will listen for requests on the URL `http://localhost:5000`. There are lots of ways to set this URL, but in this section I'll describe two: using environment variables or using command-line arguments. These are the two most common approaches I see (outside of IIS) for controlling which URLs your app uses.

**TIP** For further ways to set your application's URL, see my “5 ways to set the URLs for an ASP.NET Core app” blog entry: <http://mng.bz/go0v>.

In chapter 10 you learned about configuration in ASP.NET Core, and in particular about the concept of hosting environments so that you can use different settings when running in development compared to production. You choose the hosting environment by setting an environment variable on your machine called `ASPNETCORE_ENVIRONMENT`. The ASP.NET Core framework magically picks up this variable when your app starts and uses it to set the hosting environment.

You can use a similar special environment variable to specify the URL that your app uses; this variable is called `ASPNETCORE_URLS`. When your app starts up, it looks for this value and uses it as the application's URL. By changing this value, you can change the default URL used by all ASP.NET Core apps on the machine.

For example, you could set a temporary environment variable in Windows from the command window using

```
set ASPNETCORE_URLS=http://localhost:8000
```

Running a published application using `dotnet <app.dll>` within the same command window, as shown in figure 16.10, shows that the app is now listening on the URL provided in the `ASPNETCORE_URLS` variable.

The screenshot shows a Windows Command Prompt window titled "Command Prompt (light) - dotnet R". The command entered is "dotnet RecipeApplication.dll". The output shows the application starting and listening on http://localhost:8000. A callout bubble points to the first line of output with the text: "Set the ASPNETCORE\_URLS environment variable. It applies for the lifetime of the console window." Another callout bubble points to the second line of output with the text: "Your app listens using the URL from the ASPNETCORE\_URLS variable."

```
C:\web\RecipeApplication>set ASPNETCORE_URLS=http://localhost:8000
C:\web\RecipeApplication>dotnet RecipeApplication.dll
info: Microsoft.Hosting.Lifetime[0]
      Now listening on: http://localhost:8000
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:\web\RecipeApplication
```

Figure 16.10 Change the `ASPNETCORE_URLS` environment variable to change the URL used by ASP.NET Core apps.

You can instruct an app to listen on multiple URLs by separating them with a semicolon, or you can listen to a specific port without specifying the localhost hostname. If you set the `ASPNETCORE_URLS` environment variable to

```
http://localhost:5001;http://*:5002
```

your ASP.NET Core apps will listen for requests sent to the following:

- `http://localhost:5001`—This address is only accessible on your local computer, so it will not accept requests from the wider internet.
- `http://*:5002`—Any URL on port 5002. External requests from the internet can access the app on port 5002, using any URL that maps to your computer.

Note that you *can't* specify a different hostname, like `tastyrecipes.com`, for example. ASP.NET Core will listen to all requests on a given port. The exception is the localhost hostname, which only allows requests that came from your own computer.

**NOTE** If you find the `ASPNETCORE_URLS` variable isn't working properly, ensure you don't have a `launchSettings.json` file in the directory. When present, the values in this file take precedence. By default, `launchSettings.json` isn't included in the publish output, so this generally won't be an issue in production.

Setting the URL of an app using a single environment variable works great for some scenarios, most notably when you're running a single application in a virtual machine, or within a Docker container.<sup>6</sup>

If you're not using Docker containers, the chances are you're hosting multiple apps side-by-side on the same machine. A single environment variable is no good for setting URLs in this case, as it would change the URL of every app.

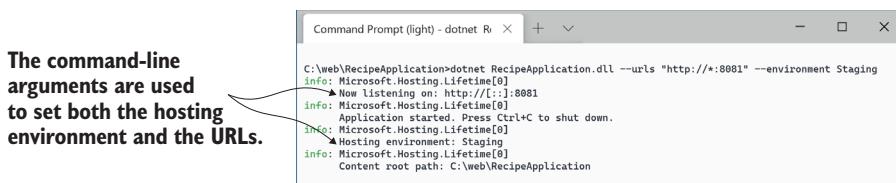
<sup>6</sup> ASP.NET Core is well-suited to running in containers, but working with containers is a separate book in its own right. For details on hosting and publishing apps using Docker, see Microsoft's "Host ASP.NET Core in Docker containers" documentation: <http://mng.bz/e5GV>.

In chapter 11 you saw that you could set the hosting environment using the `ASPNETCORE_ENVIRONMENT` variable, but you could also set the environment using the `--environment` flag when calling `dotnet run`:

```
dotnet run --no-launch-profile --environment Staging
```

You can set the URLs for your application in a similar way, using the `--urls` parameter. Using command-line arguments enables you to have multiple ASP.NET Core applications running on the same machine, listening to different ports. For example, the following command would run the recipe application, set it to listen on port 8081, and set the environment to staging, as shown in figure 16.11:

```
dotnet RecipeApplication.dll --urls "http://*:8081" --environment Staging
```



**Figure 16.11** Setting the hosting environment and URLs for an application using command-line arguments. The values passed at the command line override values provided from `appSettings.json` or environment variables.

Remember, you don't need to set your URLs in this way if you're using IIS as a reverse proxy; IIS integration handles this for you. Setting the URLs is only necessary when you're manually configuring the URL your app is listening on; for example, if you're using NGINX or are exposing Kestrel directly to clients.

**WARNING** If you are running your ASP.NET Core application without a reverse proxy, you should use *host filtering* for security reasons, to ensure your app only responds to requests for hostnames you expect. For more details, see my “Adding host filtering to Kestrel in ASP.NET Core” blog entry: <http://mng.bz/pVXK>.

Continuing the theme of deployment-related tasks, in the next section we'll take a look at optimizing some of your client-side assets for production. If you're building a Web API, this isn't something you'll have to worry about in your ASP.NET Core app, but for traditional web apps it's worth considering.

## 16.5 Optimizing your client-side assets using *BundlerMinifier*

In this section we'll explore the performance of your ASP.NET Core application in terms of the number and size of requests. You'll see how to improve the performance of your app using bundling and minification but ensure your app is still easy to debug while you're building it. Finally, we'll look at a common technique for improving app performance in production: using a content delivery network (CDN).

Have you ever used a web app or opened a web page that seemed to take forever to load? Once you stray off the beaten track of Amazon, Google, or Microsoft, it's only a matter of time before you're stuck twiddling your thumbs while the web page slowly pops into place.

Next time this happens to you, open the browser developer tools (for example, press F12 in Edge or Chrome) and take a look at the number and size of the requests the web app is making. In many cases, a high number of requests generating large responses will be responsible for the slow loading of a web page.

We'll start by exploring the problem of performance by looking at a single page from your recipe application: the login page. This is a simple page, and it isn't inherently slow, but even this is sufficient to investigate the impact of request size.

As a user, when you click the Login button, the browser sends a request to /Identity/Account/Login. Your ASP.NET Core app executes the Login.cshtml Razor Page in the default UI, which executes a Razor template and returns the generated HTML in the response, as shown in figure 16.12. That's a single request-response—a single round-trip.

But that's not it for the web page. The HTML returned by the page includes links to CSS files (for styling the page), JavaScript files (for client-side functionality—client-side form validation, in this case), and, potentially, images and other resources (though you don't have any others in this recipe app).

The browser must request each of these additional files and wait for the server to return them before the whole page is loaded. When you're developing locally, this all happens quickly, as the request doesn't have far to go, but once you deploy your app to production, it's a different matter.

Users will be requesting pages from your app from a wide variety of distances from the server, and over a wide variety of network speeds. Suddenly, the number and size of the requests and responses for your app will have a massive impact on the overall perceived speed of your app. This, in turn, can have a significant impact on how users perceive your site, and, for e-commerce sites, even how much money they spend.<sup>7</sup>

A great way to explore how your app will behave for non-optimal networks is to use the network-throttling feature in Chrome's and Edge's developer tools. This simulates the delays and network speeds associated with different types of networks, so you can

---

<sup>7</sup> There has been a lot of research done on this, including stats such as “a 0.1-second delay in page load time equals 7% loss in conversions” from <http://mng.bz/4Z2Q>.

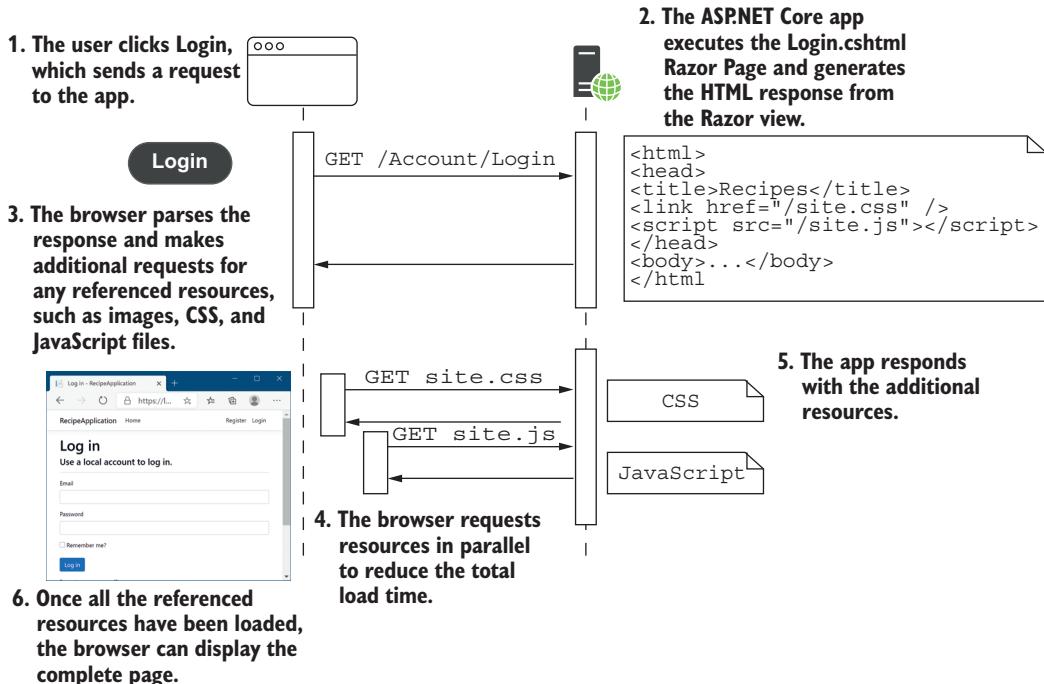


Figure 16.12 Loading a complete page for your app. The initial request returns the HTML for the page, but this may include links to other resources, such as CSS and JavaScript files. The browser must make additional requests to your app for all the outstanding resources before the page can be fully loaded.

get an idea of how your app behaves in the wild. In figure 16.13 I've loaded the login page for the recipe app, but this time with the network set to a modest Fast 3G speed.

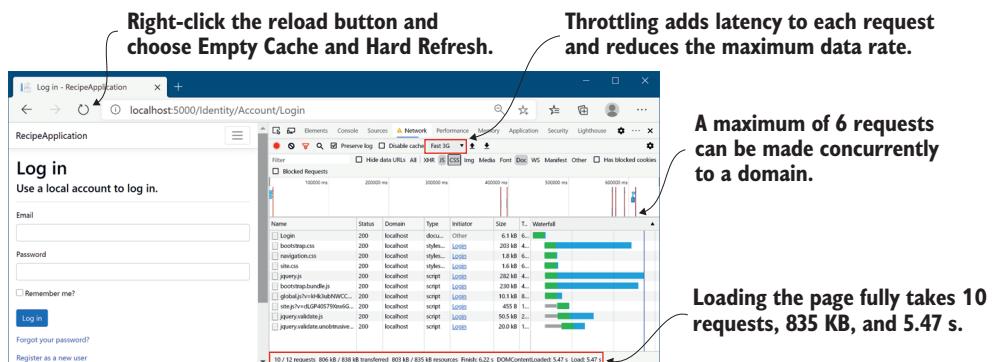


Figure 16.13 Exploring the effect of network speed on application load times. Chrome and Edge let you simulate a slower network, so you can get an impression of the experience users will have when loading your application once it's in production.

**NOTE** I've added additional files to the template—navigation.css and global.js—to make the page more representative of a real app.

Throttling the network doesn't change anything about the page or the data requested—there are 10 separate requests and almost 1 MB loaded for this single page—but it dramatically impacts the time for the page to load. Without throttling, the login page loads locally in 200 ms; with Fast 3G throttling, the login page takes 5.47 seconds to load!

**NOTE** Don't be too alarmed by these numbers. I'm making a point of reloading all the files with every request to emphasize the point, whereas in practice browsers go to great lengths to cache files to avoid having to send this amount of data.

The time it takes to fully load a page of your app is based primarily on two things:

- *The total size of the responses*—This is straight-up math; you can only return data at a certain speed, so the more data you need to return, the longer it takes.
- *The number of requests*—In general, the more requests the browser must make, the longer it takes to fully load the page. In HTTP/1.0 and HTTP/1.1, you can only make six concurrent requests to a server, so any requests after the sixth must wait for an earlier request to *finish* before they can even *start*. HTTP/2.0, which is supported by Kestrel, doesn't have this limit, but you can't always rely on clients using it.

How can you improve your app speed, assuming all the files you're serving are needed? In general, this is a big topic with lots of possibilities, such as using a CDN to serve your static files. Two of the simplest ways to improve your site's performance are to use *bundling* and *minification* to reduce the number and size of requests the browser must make to load your app.

### 16.5.1 Speeding up an app using bundling and minification

In figure 16.13 for the recipe app, you made a total of 10 requests to the server:

- One initial request for the HTML
- Three requests for CSS files
- Six requests for JavaScript files

Some of the CSS and JavaScript files are standard vendor libraries, like Bootstrap and jQuery, that are included as part of the default Razor templates, and some (navigation.css, site.css, global.js, and site.js) are files specific to your app. In this section, we're going to look at optimizing your custom CSS and JavaScript files.

If you're trying to reduce the number of requests for your app, an obvious first thought is to avoid creating multiple files in the first place. For example, instead of creating a navigation.css file and a site.css file, you could use a single file that contains all the CSS, instead of separating it out.

That's a valid solution, but putting all your code into one file may make it harder to manage and debug. As developers, we generally try to avoid this sort of monster file. A better solution is to split your code into as many files as you want, and then *bundle* the files when you come to deploy your code.

**DEFINITION** *Bundling* is the process of concatenating multiple files into a single file, to reduce the number of requests.

Similarly, when you write JavaScript, you should use descriptive variable names, comments where necessary, and whitespace to create easily readable and debuggable code. When you come to deploy your scripts, you can process and optimize them for size, instead of readability. This process is called *minification*.

**DEFINITION** *Minification* involves processing code to reduce its size without changing the behavior of the code. Processing has many different levels, which typically include removing comments and whitespace, and can extend to renaming variables to give them shorter names or removing whole sections of code entirely if they're unused.

As an example, look at the JavaScript in the following listing. This (very contrived) function adds up some numbers and returns them. It includes (excessively) descriptive variable names, comments, and plenty of use of whitespace, but it's representative of the JavaScript you might find in your own app.

### Listing 16.3 Example JavaScript function before minification

```
function myFunc() {
    // this function doesn't really do anything,
    // it's just here so that we can show off minifying!
    function innerFunctionToAddTwoNumbers(
        theFirstNumber, theSecondNumber) {
        // i'm nested inside myFunc
        return theFirstNumber + theSecondNumber;
    }
    var shouldAddNumbers = true;
    var totalOfAllTheNumbers = 0;

    if (shouldAddNumbers == true) {
        for (var index = 0; i < 10; i++) {
            totalOfAllTheNumbers =
                innerFunctionToAddTwoNumbers(totalOfAllTheNumbers, index);
        }
    }
    return totalOfAllTheNumbers;
}
```

This function takes a total of 588 bytes as it's currently written, but after minification it's reduced to 95 bytes—16% of its original size. The behavior of the code is identical, but the output, shown in the following listing, is optimized to reduce its size. It's clearly

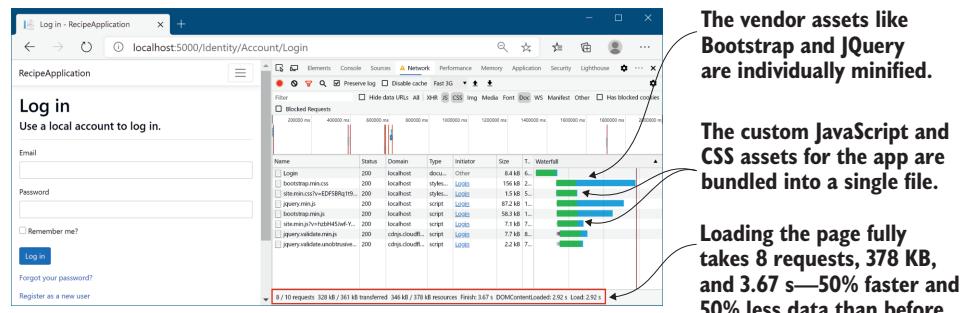
not something you'd want to debug: you'd only use minified versions of your file in production; you'd use the original source files when developing.

#### Listing 16.4 Example JavaScript function after minification

```
function myFunc() {function r(n,t){return n+t}var n=0,t;if(1)for(t=0;i<10;i++)n=r(n,t);return n}
```

Optimizing your static files using bundling and minification can provide a free boost to your app's performance when you deploy your app to production, while still letting you develop using easy-to-read and separated files.

Figure 16.14 shows the impact of bundling and minifying the files for the login page of the recipe app. Each of the vendor files has been minified to reduce its size, and the custom assets have been bundled and minified to reduce both their size and the number of requests. This reduced the number of requests from 10 to 8, the total amount of data from 580 KB to 270 KB, and the load time from 6.45 s to 3.15 s.



**Figure 16.14** By bundling and minifying your client-side resources, you can reduce both the number of requests required and the total data to transfer, which can significantly improve performance. In this example, bundling and minification cut the time to fully load in half.

**NOTE** The vendor assets, such as jQuery and Bootstrap, aren't bundled with your custom scripts in figure 16.14. This lets you load those files from a CDN, as I'll touch on in section 16.5.4.

This performance improvement can be achieved with little effort on your part, and with no impact on your development process. In the next section I'll show how you can include bundling and minification as part of your ASP.NET Core build process, and how to customize the bundling and minification processes for your app.

#### 16.5.2 Adding BundlerMinifier to your application

Bundling and minification isn't a new idea, so you have many ways to achieve the same result. The previous version of ASP.NET performed bundling and minification in managed code, whereas JavaScript task runners such as gulp, Grunt, and webpack are

commonly used for these sorts of tasks. In fact, if you’re writing a SPA, you’re almost certainly already performing bundling and minification as a matter of course.

ASP.NET Core includes support for bundling and minification via a NuGet package called BuildBundlerMinifier or a Visual Studio extension version called Bundler & Minifier. You don’t have to use either of these, and if you’re already using other tools such as gulp or webpack, I suggest you continue to use them instead. But if you’re getting started with a new project, I suggest considering BundlerMinifier; you can always switch to a different tool later.

You have two options for adding BundlerMinifier to your project:

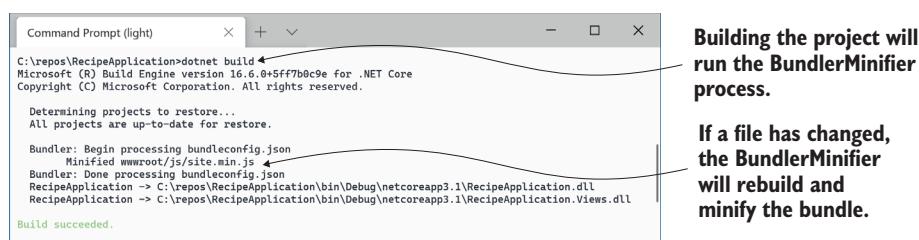
- You can install the Bundler & Minifier Visual Studio extension from Tools > Extensions and Updates.
- You can add the BuildBundlerMinifier NuGet package to your project.

Whichever approach you use, they both use the same underlying BundlerMinifier library.<sup>8</sup> I prefer to use the NuGet package approach, as it’s cross-platform and will automatically bundle your resources for you, but the extension is useful for performing ad hoc bundling. If you do use the Visual Studio extension, make sure you enable Bundle on build, as you’ll see shortly.

You can install the BuildBundlerMinifier NuGet package in your project with this command:

```
dotnet add package BuildBundlerMinifier
```

With the BuildBundlerMinifier package installed, whenever you build your project, the BundlerMinifier will check your CSS and JavaScript files for changes. If something has changed, new bundled and minified files are created, as shown in figure 16.15, where I modified a JavaScript file.



**Figure 16.15** Whenever the project is built, the BuildBundlerMinifier tool looks for changes in the input files and builds new bundles as necessary.

<sup>8</sup> The BundlerMinifier library and extension are open source on GitHub: <https://github.com/madskristensen/BundlerMinifier/>.

As you can see in figure 16.15, the bundler minified your JavaScript code and created a new file at wwwroot/js/site.min.js. But why did it pick this name? Well, because you told it to in bundleconfig.json. You add this JSON file to the root folder of your project, and it controls the BundlerMinifier process.

Listing 16.5 shows a typical bundleconfig.json configuration for a small app. This defines which files to include in each bundle, where to write each bundle, and what minification options to use. Two bundles are configured here: one for CSS and one for JavaScript. You can add bundles to the JSON array, or you can customize the existing provided bundles.

#### Listing 16.5 A typical bundleconfig.json file

```
[  
  {  
    "outputFileName": "wwwroot/css/site.min.css", ← The bundler will  
    "inputFiles": [  
      "wwwroot/css/site.css" ← create a file with  
      "wwwroot/css/navigation.css" ← this name.  
    ],  
    {  
      "outputFileName": "wwwroot/js/site.min.js", ← The files listed in inputFiles  
      "inputFiles": [  
        "wwwroot/js/site.js" ← are minified and concatenated  
        "wwwroot/js/*.js", ← into outputFileName.  
        "!wwwroot/js/site.min.js" ←  
      ],  
      "minify": { ← You can specify multiple  
        "enabled": true, ← bundles, each with an  
        "renameLocals": true ← output filename.  
      },  
      "sourceMap": false ←  
    }  
  }  
]
```

**You should create separate bundles for CSS and JavaScript.**

**The JavaScript bundler has some additional options.**

The bundler will create a file with this name.

The files listed in inputFiles are minified and concatenated into outputFileName.

You can specify multiple bundles, each with an output filename.

You can use globbing patterns to specify files to include.

The ! symbol excludes the matching file from the bundle.

You can optionally create a source map for the bundled JavaScript file.

For each bundle you can list a specific set of files to include, as specified in the `inputFiles` property. If you add a new CSS or JavaScript file to your project, you have to remember to come back to `bundleconfig.json` and add it to the list.

Alternatively, you can use *globbing* patterns to automatically include new files by default. This isn't always possible, such as when you need to ensure that files are concatenated in a given order, but I find it preferable in many cases. The example in the previous listing uses globbing for the JavaScript bundle: the pattern includes all `.js` files in the `wwwroot/js` folder but excludes the minified output file itself.

**DEFINITION** *Globbing* patterns use wildcards to represent many different characters. For example, `*.css` would match all files with a `.css` extension, whatever the filename.

When you build your project, the BundlerMinifier optimizes your CSS files into a single wwwroot/css/site.min.css file and your JavaScript into a single wwwroot/js/site.min.js file, based on the settings in bundleconfig.json. In the next section we'll look at how to include these files when you run in production, continuing to use the original files when developing locally.

**NOTE** The BundlerMinifier package is great for optimizing your CSS and JavaScript resources. But images are another important resource to consider for optimization, and they can easily constitute the bulk of your page's size. Unfortunately, there aren't any built-in tools for this, so you'll need to look at other options; TinyPNG (<https://tinypng.com>) is one.

### 16.5.3 Using minified files in production with the Environment

#### Tag Helper

Optimizing files using bundling and minification is great for performance, but you want to use the original files during development. The easiest way to achieve this split in ASP.NET Core is to use the Environment Tag Helper to conditionally include the original files when running in the development hosting environment, and to use the optimized files in other environments.

You learned about the Environment Tag Helper in chapter 8, where we used it to show a banner on the homepage when the app was running in the testing environment. Listing 16.6 shows how you can take a similar approach in the \_Layout.cshtml page for the CSS files of your recipe app by using two Environment Tag Helpers: one for when you're in development, and one for when you aren't (you're in production or staging, for example). You can use similar Tag Helpers for the JavaScript files in the app.

**Listing 16.6 Using Environment Tag Helpers to conditionally render optimized files**



When the app detects it isn't running in the development hosting environment (as you learned in chapter 11), it will switch to rendering links for the optimized files.

This gives you the best of both worlds: performance in production and ease of development.

The example in listing 16.6 also included a minified version of Bootstrap, even though you didn't configure this as part of the BundlerMinifier. It's common for CSS and JavaScript libraries like Bootstrap to include a pre-minified version of the file for you to use in production. For those that do, it's often better to exclude them from your bundling process, as this allows you to potentially serve the file from a public CDN.

#### 16.5.4 Serving common files from a CDN

A public CDN is a website that hosts commonly used files, such as Bootstrap or jQuery, which you can reference from your own apps. They have several advantages:

- They're normally fast.
- They save your server from having to serve the file, saving bandwidth.
- Because the file is served from a different server, it doesn't count toward the six concurrent requests allowed to your server in HTTP/1.0 and HTTP/1.1.<sup>9</sup>
- Many different apps can reference the same file, so a user visiting your app may have already cached the file by visiting a different website, and they may not need to download it at all.

It's easy to use a CDN in principle: reference the CDN file instead of the file on your own server. Unfortunately, you need to cater for the fact that, like any server, CDNs can sometimes fail. If you don't have a fallback mechanism to load the file from a different location, such as your server, this can result in your app looking broken.

Luckily, ASP.NET Core includes several Tag Helpers to make working with CDNs and fallbacks easier. Listing 16.7 shows how you could update your CSS Environment Tag Helper to serve Bootstrap from a CDN when running in production, and to include a fallback. The fallback test creates a temporary HTML element and applies a Bootstrap style to it. If the element has the expected CSS properties, the fallback test passes, because Bootstrap must be loaded. If it doesn't have the required properties, Bootstrap will be loaded from the alternative, local link instead.

##### Listing 16.7 Serving Bootstrap CSS styles from a CDN with a local fallback

```
<environment include="Development">
    <link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.css" />
    <link rel="stylesheet" href="~/css/navigation.css" />
    <link rel="stylesheet" href="~/css/site.css" />
</environment>
<environment exclude="Development">
    <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/
        ↗ bootstrap/4.3.1/css/bootstrap.min.css"
```

By default, Bootstrap is loaded from a CDN.

<sup>9</sup> This limit is not fixed in stone, but modern browsers all use the same limit. See Push Technology's "Browser connection limitations" article: <http://mng.bz/OEWw>.

```
asp-fallback-test-class="sr-only"
asp-fallback-test-property="position"
asp-fallback-test-value="absolute"
asp-fallback-href="~/lib/bootstrap/dist/css/bootstrap.min.css" />
<link rel="stylesheet" href("~/css/site.min.css" />
</environment>
```

... and checks that the element has a CSS position of absolute. This indicates Bootstrap was loaded.

The fallback test applies the sr-only class to an element ...

If the fallback check fails, the CDN must have failed, so Bootstrap is loaded from the local link.

Optimizing your static files is an important step to consider before you put your app into production, as it can have a significant impact on performance. Luckily, the BuildBundlerMinifier package makes it easy to optimize your CSS and JavaScript files. If you couple that with serving common files from a CDN, your app will be as speedy as possible for users in production.

That brings us to the end of this chapter on publishing your app. This last mile of app development, deploying an application to a server where users can access it, is a notoriously thorny issue. Publishing an ASP.NET Core application is easy enough, but the multitude of hosting options available makes providing concise steps for every situation difficult.

Whichever hosting option you choose, there's one critical topic that is often overlooked but is crucial for resolving issues quickly: logging. In the next chapter you'll learn about the logging abstractions in ASP.NET Core, and how you can use them to keep tabs on your app once it's in production.

## Summary

- ASP.NET Core apps are console applications that self-host a web server. In production, you typically use a reverse proxy, which handles the initial request and passes it to your app. Reverse proxies can provide additional security, operations, and performance benefits, but they can also add complexity to your deployments.
- .NET has two parts: the .NET SDK (also known as the .NET CLI) and the .NET Runtime. When you're developing an application, you use the .NET CLI to restore, build, and run your application. Visual Studio uses the same .NET CLI commands from the IDE.
- When you want to deploy your app to production, you need to publish your application, using dotnet publish. This creates a folder containing your application as a DLL, along with all its dependencies.
- To run a published application, you don't need the .NET CLI because you won't be building the app. You only need the .NET Runtime to run a published app. You can run a published application using the dotnet app.dll command, where app.dll is the application .dll created by the dotnet publish command.

- To host ASP.NET Core applications in IIS, you must install the ASP.NET Core Module. This allows IIS to act as a reverse proxy for your ASP.NET Core app. You must also install the .NET Runtime and the ASP.NET Core Runtime, which are installed as part of the ASP.NET Core Windows Hosting Bundle.
- IIS can host ASP.NET Core applications using one of two modes: in-process and out-of-process. The out-of-process mode runs your application as a separate process, as is typical for most reverse proxies. The in-process mode runs your application as part of the IIS process. This has performance benefits, as no inter-process communication is required.
- To prepare your application for publishing to IIS with ASP.NET Core, ensure you call `UseIISIntegration()` and `UseIIS()` on `WebHostBuilder`. The `ConfigureWebHostDefaults` static helper method does this automatically.
- When you publish your application using the .NET CLI, a `web.config` file will be added to the output folder. It's important that this file be deployed with your application when publishing to IIS, as it defines how your application should be run.
- The URL that your app will listen on is specified by default using the environment variable `ASPNETCORE_URLS`. Setting this value will change the URL for all the apps on your machine. Alternatively, pass the `--urls` command-line argument when running your app; for example, `dotnet app.dll --urls http://localhost:80`.
- It's important to optimize your client-side assets to reduce the size and number of files that must be downloaded by a client's web browser when a page loads. You can achieve this by bundling and minifying your assets.
- You can use the `BuildBundlerMinifier` package to combine multiple JavaScript or CSS files together in a process called bundling. You can reduce the size of the files in a process called minification, in which unnecessary whitespace is removed and variables are renamed while preserving the function of the file.
- You can install a Visual Studio extension to control bundling and minification, or you can install the `BuildBundlerMinifier` package to automatically perform bundling and minification on each build of the project. Using the extension allows you to minify on an ad hoc basis, but using the NuGet package allows you to automate the process.
- The settings for bundling and minification are stored in the `bundleconfig.json` file, where you can define the different output bundle files and choose which files to include in the bundle. You can explicitly specify files, or you can use globbing patterns to include multiple files using wildcard characters. Globbing is typically easier and less error prone, but you will need to specify files explicitly if they must be bundled in a specific order.
- Use the `Environment Tag Helper` to conditionally render your bundles in production only. This lets you optimize for performance in production and readability in development.

- For common files shared by multiple apps, such as jQuery or Bootstrap, you can serve files from a CDN. These are websites that host the common files, so your app doesn't need to serve them itself.
- Use Link and Script Tag Helpers to check that the file has loaded correctly from the CDN. These can test that a file has been downloaded by a client and ensures that your server is used as a fallback should the CDN fail.

## *Part 3*

# *Extending your applications*

We covered a huge amount of content in parts 1 and 2: we looked at all the main functional components you'll use to build both traditional server-rendered Razor Pages apps as well as Web APIs. In part 3 we'll look at six different topics that build on what you've learned so far: logging, security, custom components, interacting with third-party HTTP APIs, background services, and testing.

Adding logging to your application is one of those activities that's often left until after you discover a problem in production. Adding sensible logging from the get-go will help you quickly diagnose and fix errors as they arise. Chapter 17 introduces the logging framework built in to ASP.NET Core. You'll see how you can use it to write log messages to a wide variety of locations, whether it's the console, a file, or a third-party remote-logging service.

Correctly securing your app is an important part of web development these days. Even if you don't feel you have any sensitive data in your application, you have to make sure you protect your users from attacks by adhering to security best practices. In chapter 18 I'll describe some common vulnerabilities, how attackers can exploit them, and what you can do to protect your applications.

In part 1 you learned about the middleware pipeline, and you saw how it is fundamental to all ASP.NET Core applications. In chapter 19 you'll learn how to create your own custom middleware, as well as simple endpoints, for when you don't need the full power of Razor Pages or a Web API controller. You'll also learn how to handle some complex chicken-and-egg configuration issues that

often arise in real-life applications. Finally, you'll learn how to replace the built-in dependency injection container with a third-party alternative.

In chapter 20 you'll learn how to create custom components for working with Razor Pages and API controllers. You'll learn how to create custom Tag Helpers and validation attributes, and I'll introduce a new component—view components—for encapsulating logic with Razor view rendering. You'll also learn how to replace the attribute-based validation framework used by default in ASP.NET Core with an alternative.

Most apps you build aren't designed to stand on their own. It's very common for your app to need to interact with third-party APIs, whether those are APIs for sending emails, fetching exchange rates, or taking payments. In chapter 21 you'll learn how to interact with third-party APIs using the `IHttpClientFactory` abstraction to simplify configuration, to add transient fault handling, and to avoid common pitfalls.

This book deals primarily with serving HTTP traffic, both server-rendered web pages using Razor Pages and Web APIs commonly used by mobile and single-page applications. However, many apps require long-running background tasks that execute jobs on a schedule or that process items from a queue. In chapter 22 I'll show how you can create these long-running background tasks in your ASP.NET Core applications. I'll also show how to create standalone services that only have background tasks, without any HTTP handling, and how to install them as a Windows Service or as a Linux `systemd` daemon.

Chapter 23, the final chapter, covers testing your application. The exact role of testing in application development can sometimes lead to philosophical arguments, but in chapter 23 I'll stick to the practicalities of testing your app with the xUnit test framework. You'll see how to create unit tests for your apps, how to test code that's dependent on EF Core using an in-memory database provider, and how to write integration tests that can test multiple aspects of your application at once.



# *Monitoring and troubleshooting errors with logging*

---

## **This chapter covers**

- Understanding the components of a log message
- Writing logs to multiple output locations
- Controlling log verbosity in different environments using filtering
- Using structured logging to make logs searchable

Logging is one of those topics that seems unnecessary, right up until you desperately need it! There's nothing more frustrating than finding an issue that you can only reproduce in production, and then discovering there are no logs to help you debug it.

*Logging* is the process of recording events or activities in an app, and it often involves writing a record to a console, a file, the Windows Event Log, or some other system. You can record anything in a log message, though there are generally two different types of messages:

- *Informational messages*—A standard event occurred: a user logged in, a product was placed in a shopping cart, or a new post was created on a blogging app.
- *Warnings and errors*—An error or unexpected condition occurred: a user had a negative total in the shopping cart, or an exception occurred.

Historically, a common problem with logging in larger applications was that each library and framework would generate logs in a slightly different format, if at all. When an error occurred in your app and you were trying to diagnose it, this inconsistency made it harder to connect the dots in your app to get the full picture and understand the problem.

Luckily, ASP.NET Core includes a new, generic logging interface that you can plug into. It's used throughout the ASP.NET Core framework code itself, as well as by third-party libraries, and you can easily use it to create logs in your own code. With the ASP.NET Core logging framework, you can control the verbosity of logs coming from each part of your code, including the framework and libraries, and you can write the log output to any destination that plugs into the framework.

In this chapter, I cover the ASP.NET Core logging framework in detail and explain how you can use it to record events and diagnose errors in your own apps. In section 17.1 I'll describe the architecture of the logging framework. You'll learn how DI makes it easy for both libraries and apps to create log messages, as well as to write those logs to multiple destinations.

In section 17.2 you'll learn how to write your own log messages in your apps with the `ILogger` interface. We'll break down the anatomy of a typical log record and look at its properties, such as the log level, category, and message.

Writing logs is only useful if you can read them, so in section 17.3 you'll learn how to add *logging providers* to your application. Logging providers control where your app writes your log messages. This could be to the console, to a file, or even to an external service. I'll show you how to add a logging provider that writes logs to a file, and how to configure a popular third-party logging provider called Serilog in your app.

Logging is an important part of any application, but determining *how much* logging is enough can be a tricky question. On the one hand, you want to provide sufficient information to be able to diagnose any problems. On the other, you don't want to fill your logs with data that makes it hard to find the important information when you need it. Even worse, what is sufficient in development might be far too much once you're running in production.

In section 17.4 I'll explain how you can filter log messages from various sections of your app, such as the ASP.NET Core infrastructure libraries, so that your logging providers only write the important messages. This lets you keep that balance between extensive logging in development and only writing important logs in production.

In the final section of this chapter, I'll touch on some of the benefits of *structured logging*, an approach to logging that you can use with some providers for the ASP.NET Core logging framework. Structured logging involves attaching data to log messages as key-value pairs to make it easier to search and query logs. You might attach a unique customer ID to every log message generated by your app, for example. Finding all the log messages associated with a user is much simpler with this approach, compared to recording the customer ID in an inconsistent manner as part of the log message.

We'll start this chapter by digging into what logging involves, and why your future self will thank you for using logging effectively in your application. Then we'll look at the pieces of the ASP.NET Core logging framework you'll use directly in your apps, and how they fit together.

## 17.1 Using logging effectively in a production app

Imagine you've just deployed a new app to production, when a customer calls saying that they're getting an error message using your app. How would you identify what caused the problem? You could ask the customer what steps they were taking, and potentially try to recreate the error yourself, but if that doesn't work, you're left trawling through the code, trying to spot errors with nothing else to go on.

Logging can provide the extra context you need to quickly diagnose a problem. Arguably, the most important logs capture the details about the error itself, but the events that led to the error can be just as useful in diagnosing the cause of an error.

There are many reasons for adding logging to an application, but, typically, the reasons fall into one of three categories:

- Logging for auditing or analytics reasons, to trace when events have occurred
- Logging errors
- Logging non-error events to provide a breadcrumb trail of events when an error does occur

The first of these reasons is simple. You may be required to keep a record of every time a user logs in, for example, or you may want to keep track of how many times a particular API method is called. Logging is an easy way to record the behavior of your app, by writing a message to the log every time an interesting event occurs.

I find the second reason for logging to be the most common. When an app is working perfectly, logs often go completely untouched. It's when there's an issue and a customer comes calling that logs become invaluable. A good set of logs can help you understand the conditions in your app that caused an error, including the context of the error itself, but also the context in previous requests.

**TIP** Even with extensive logging in place, you may not realize you have an issue in your app unless you look through your logs regularly. For any medium to large app, this becomes impractical, so monitoring services such as Raygun (<https://raygun.io>) or Sentry (<https://sentry.io>) can be invaluable for notifying you of issues quickly.

If this sounds like a lot of work, then you're in luck. ASP.NET Core does a ton of the "breadcrumb" logging for you so that you can focus on creating high-quality log messages that provide the most value when diagnosing problems.

### 17.1.1 Highlighting problems using custom log messages

ASP.NET Core uses logging throughout its libraries. Depending on how you configure your app, you'll have access to the details of each request and EF Core query, even without adding additional logging messages to your own code. In figure 17.1 you can see the log messages created when you view a single recipe in the recipe application.

```

Command Prompt (light) - dotnet n. x + □ ×
info: Microsoft.AspNetCore.Hosting.Diagnostics[1]
  Request starting HTTP/2 GET https://localhost:5001/Recipes/View/5
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[8]
  Executing endpoint /Recipes/View/5
info: Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[1]
  Route matched with {page = "Recipes/View", area = ""}. Executing page /Recipes/View<
info: Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[1]
  Executing handler method RecipeApplication.Pages.Recipes.ViewModel.OnGetSync - ModelState is Valid
info: Microsoft.EntityFrameworkCore.Infrastructure[100]
  Executed DbCommand<Text> (1ms) [Parameters=@_id={5}, CommandType='Text', CommandTimeout='30']
    SELECT * FROM [Ingredient] AS r
    WHERE (@r.[RecipeId] = @_id OR @_id IS NULL)
    AND NOT (@r.[IsDeleted])
    LIMIT 2
  ) AS r*
  LEFT JOIN [Ingredient] AS i ON r.[RecipeId] = i.[RecipeId]
  ORDER BY r.[RecipeId], i.[IngredientId]
info: Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[102]
  Executed handler method RecipeApplication.Pages.Recipes.ViewModel.OnGetSync - ModelState is Valid
info: Microsoft.AspNetCore.Mvc.StatusCodeResult[1]
  Executing HttpStatusCodeResult, setting HTTP status code 404<
info: Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[4]
  Executed page /Recipes/View in 76.128ms
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[1]
  Executed endpoint /Recipes/View

```

**A single request is made to the URL /Recipe/View/2. Internal ASP.NET Core framework libraries generate logs for the Razor Pages endpoint.**

**The OnGetAsync handler on the Recipes/View.cshtml Razor Page is executed.**

**EF Core logs the SQL requests made to the database.**

**The ActionResult type and the HTTP response code are logged.**

**Figure 17.1** The ASP.NET Core Framework libraries use logging throughout. A single request generates multiple log messages that describe the flow of the request through your application.

This gives you a lot of useful information. You can see which URL was requested, the Razor Page and page handler that were invoked, the EF Core database command, the action result invoked, and the response. This information can be invaluable when trying to isolate a problem, whether a bug in a production app or a feature in development when you're working locally.

This infrastructure logging can be useful, but log messages that you create yourself can have even greater value. For example, you may be able to spot the cause of the error from the log messages in figure 17.1—we're attempting to view a recipe with an unknown RecipeId of 5, but it's far from obvious. If you explicitly add a log message to your app when this happens, as in figure 17.2, the problem is much more apparent.

```

Command Prompt (light) - dotnet n. x + □ ×
info: Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[1]
  Route matched with {page = "Recipes/View", area = ""}. Executing page /Recipes/View
info: Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[101]
  Executing handler method RecipeApplication.Pages.Recipes.ViewModel.OnGetSync - ModelState is Valid
info: Microsoft.EntityFrameworkCore.Database.Command[2010]
  Executed DbCommand<Text> (1ms) [Parameters=@_id={5}, CommandType='Text', CommandTimeout='30']
    SELECT * FROM [Ingredient] AS r
    WHERE (@r.[RecipeId] = @_id OR @_id IS NULL)
    AND NOT (@r.[IsDeleted])
    LIMIT 2
  ) AS r*
  LEFT JOIN [Ingredient] AS i ON r.[RecipeId] = i.[RecipeId]
  ORDER BY r.[RecipeId], i.[IngredientId]
WRN: RecipeApplication.Pages.Recipes.ViewModel[12]
  Could not find recipe id 5
info: Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[102]
  Executed handler method OnGetSync, returned result Microsoft.AspNetCore.Mvc.NotFoundResult.
info: Microsoft.AspNetCore.Mvc.StatusCodeResult[1]
  Executing HttpStatusCodeResult, setting HTTP status code 404<
info: Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[4]
  Executed page /Recipes/View in 76.128ms
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[1]
  Executed endpoint /Recipes/View

```

**You can write your own logs from your app's classes, like this log from the Razor Page ViewModel. These are often more useful for diagnosing issues and tracing the behavior of your app.**

**Figure 17.2** You can write your own logs. These are often more useful for identifying issues and interesting events in your apps.

This custom log message easily stands out and clearly states both the problem (the recipe with the requested ID doesn't exist) and the parameters/variables that led to the issue (the ID value of 5). Adding similar log messages to your own applications will make it easier for you to diagnose problems, track important events, and generally know what your app is doing.

Hopefully you're now motivated to add logging to your apps, so we'll dig into the details of what that involves. In section 17.1.2 you'll see how to create a log message and how to define where the log messages are written. We'll look in detail at these two aspects in sections 17.2 and 17.3; first, though, we'll look at where they fit in terms of the ASP.NET Core logging framework as a whole.

### 17.1.2 The ASP.NET Core logging abstractions

The ASP.NET Core logging framework consists of a number of logging abstractions (interfaces, implementations, and helper classes), the most important of which are shown in figure 17.3:

- **ILogger**—This is the interface you'll interact with in your code. It has a `Log()` method, which is used to write a log message.
- **ILoggerProvider**—This is used to create a custom instance of an `ILogger`, depending on the provider. A “console” `ILoggerProvider` would create an `ILogger` that writes to the console, whereas a “file” `ILoggerProvider` would create an `ILogger` that writes to a file.
- **ILoggerFactory**—This is the glue between the `ILoggerProvider` instances and the `ILogger` you use in your code. You register `ILoggerProvider` instances with an `ILoggerFactory` and call `CreateLogger()` on the `ILoggerFactory` when you need an `ILogger`. The factory creates an `ILogger` that wraps each of the providers, so when you call the `Log()` method, the log is written to every provider.

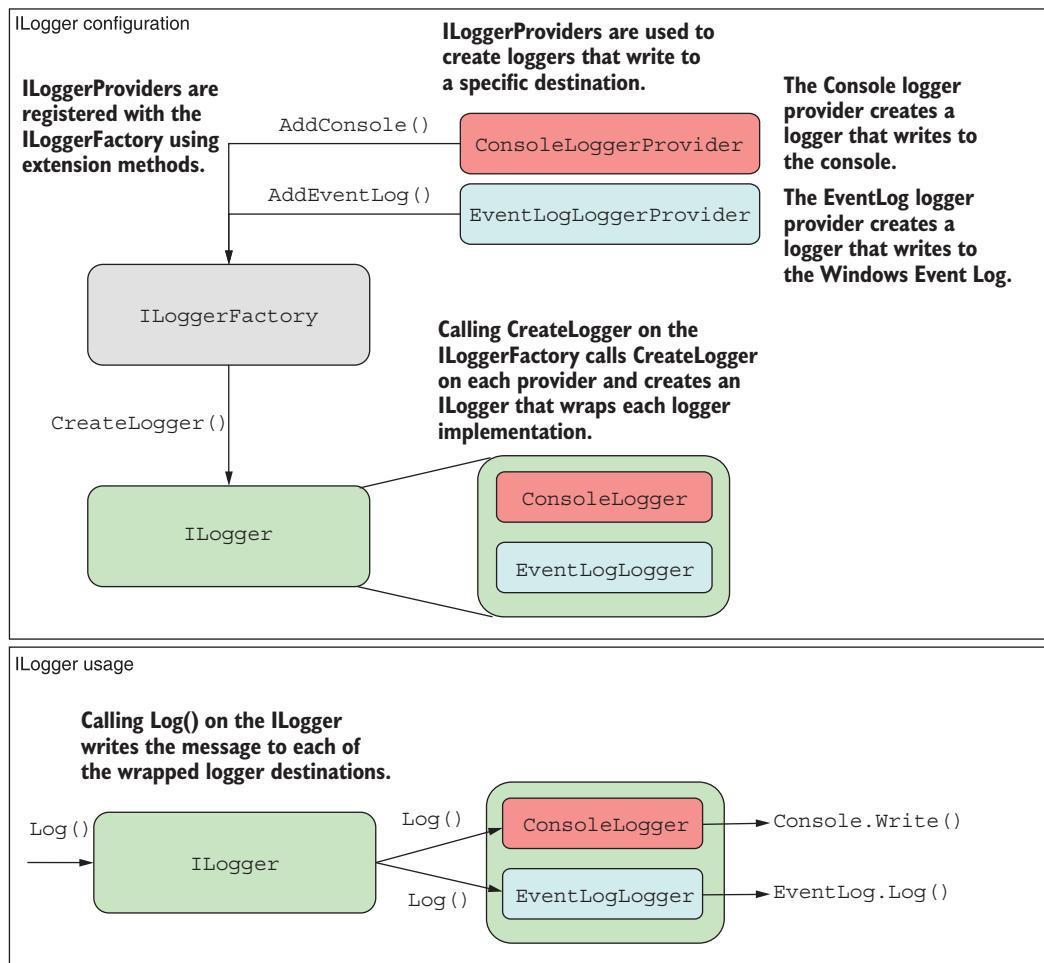
The design in figure 17.3 makes it easy to add or change where your application writes the log messages, without having to change your application code. The following listing shows all the code required to add an `ILoggerProvider` that writes logs to the console.

**Listing 17.1 Adding a console log provider to `IHost` in `Program.cs`**

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        new HostBuilder()
            .ConfigureLogging(builder => builder.AddConsole()) <----->
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

Add new providers with the `ConfigureLogging` extension method on `HostBuilder`.



**Figure 17.3** The components of the ASP.NET Core logging framework. You register logging providers with an `ILoggerFactory`, which is used to create implementations of `ILogger`. You write logs to the `ILogger`, which uses the `ILogger` implementations to output logs to the console or a file. This design allows you to send logs to multiple locations without having to configure those locations when you create a log message.

**NOTE** The console logger is added by default in the `CreateDefaultBuilder` method, as you'll see in section 17.3.

Other than this configuration on `IHostBuilder`, you don't interact with `ILoggerProvider` instances directly. Instead, you write logs using an instance of `ILogger`, as you'll see in the next section.

## 17.2 Adding log messages to your application

In this section we'll look in detail at how to create log messages in your own application. You'll learn how to create an instance of `ILogger`, and how to use it to add logging to an existing application. Finally, we'll look at the properties that make up a logging record, what they mean, and what you can use them for.

Logging, like almost everything in ASP.NET Core, is available through DI. To add logging to your own services, you only need to inject an instance of `ILogger<T>`, where `T` is the type of your service.

**NOTE** When you inject `ILogger<T>`, the DI container indirectly calls `ILoggerFactory.CreateLogger<T>()` to create the wrapped `ILogger` of figure 17.3. In section 17.2.2 you'll see how to work directly with `ILoggerFactory` if you prefer. The `ILogger<T>` interface also implements the non-generic `ILogger` interface but adds additional convenience methods.

You can use the injected `ILogger` instance to create log messages, which it writes to each configured `ILoggerProvider`. The following listing shows how to inject an `ILogger<>` instance into the `PageModel` of the `Index.cshtml` Razor Page for the recipe application from previous chapters and how to write a log message indicating how many recipes were found.

### Listing 17.2 Injecting `ILogger` into a class and writing a log message

```
public class IndexModel : PageModel
{
    private readonly RecipeService _service;
    private readonly ILogger<IndexModel> _log;

    public ICollection<RecipeSummaryViewModel> Recipes { get; set; }

    public IndexModel(
        RecipeService service,
        ILogger<IndexModel> log)
    {
        _service = service;
        _log = log;
    }

    public void OnGet()
    {
        Recipes = _service.GetRecipes();
        _log.LogInformation(
            "Loaded {RecipeCount} recipes", Recipes.Count);
    }
}
```

In this example, you're using one of the many extension methods on `ILogger` to create the log message, `LogInformation()`. There are many extension methods on `ILogger` that let you easily specify a `LogLevel` for the message.

**DEFINITION** The *log level* of a log is how important it is and is defined by the `LogLevel` enum. Every log message has a log level.

You can also see that the message you pass to the `LogInformation` method has a placeholder indicated by braces, `{RecipeCount}`, and you pass an additional parameter, `Recipes.Count`, to the logger. The logger will replace the placeholder with the parameter at runtime. Placeholders are matched with parameters by position, so if you include two placeholders, for example, the second placeholder is matched with the second parameter.

**TIP** You could have used normal string interpolation to create the log message; for example,  `$"Loaded {Recipes.Count} recipes"`. But I recommend always using placeholders, as they provide additional information for the logger that can be used for structured logging, as you'll see in section 17.5.

When the `OnGet` page handler in the `IndexModel` executes, `ILogger` writes a message to any configured logging providers. The exact format of the log message will vary from provider to provider, but figure 17.4 shows how the console provider would display the log message from listing 17.2.



**Figure 17.4** An example log message, as it's written to the default console provider. The log level category provides information about how important the message is and where it was generated. The EventId provides a way to identify similar log messages.

The exact presentation of the message will vary depending on where the log is written, but each log record includes up to six common elements:

- *Log level*—The log level of the log is how important it is and is defined by the `LogLevel` enum.
- *Event category*—The category may be any string value, but it's typically set to the name of the class creating the log. For `ILogger<T>`, the full name of the type `T` is the category.
- *Message*—This is the content of the log message. It can be a static string, or it can contain placeholders for variables, as shown in listing 17.2. Placeholders are indicated by braces, `{ }`, and are substituted with the provided parameter values.
- *Parameters*—If the message contains placeholders, they're associated with the provided parameters. For the example in listing 17.2, the value of `Recipes.Count` is

assigned to the placeholder called `RecipeCount`. Some loggers can extract these values and expose them in your logs, as you'll see in section 17.5.

- *Exception*—If an exception occurs, you can pass the exception object to the logging function along with the message and other parameters. The logger will log the exception, in addition to the message itself.
- *EventId*—This is an optional integer identifier for the error, which can be used to quickly find all similar logs in a series of log messages. You might use an `EventId` of 1000 when a user attempts to load a non-existent recipe, and an `EventId` of 1001 when a user attempts to access a recipe they don't have permission to access. If you don't provide an `EventId`, the value 0 will be used.

Not every log message will have all these elements—you won't always have an `Exception` or parameters, for example. There are various overloads to the logging methods that take these elements as additional method parameters. Besides these optional elements, each message will have, at the very least, a level, category, and message. These are the key features of the log, so we'll look at each in turn.

### 17.2.1 Log level: How important is the log message?

Whenever you create a log using `ILogger`, you must specify the *log level*. This indicates how serious or important the log message is, and it's an important factor when it comes to filtering which logs get written by a provider, as well as finding the important log messages after the fact.

You might create an `Information` level log when a user starts to edit a recipe. This is useful for tracing the application's flow and behavior, but it's not important, because everything is normal. But if an exception is thrown when the user attempts to save the recipe, you might create a `Warning` or `Error` level log.

The log level is typically set by using one of several extension methods on the `ILogger` interface, as shown in listing 17.3. This example creates an `Information` level log when the `View` method executes, and a `Warning` level error if the requested recipe isn't found.

**Listing 17.3 Specifying the log level using extension methods on `ILogger`**

```
private readonly ILogger _log;           ← An ILogger instance is injected into the
public async IActionResult OnGet(int id)  controller using constructor injection.
{
    _log.LogInformation(
        "Loading recipe with id {RecipeId}", id);   | Writes an Information
                                                        | level log message

    Recipe = _service.GetRecipeDetail(id);
    if (Recipe is null)
    {
        _log.LogWarning(
            "Could not find recipe with id {RecipeId}", id); | Writes a Warning
                                                        | level log message
        return NotFound();
    }
}
```

```
        return Page();
    }
```

The `LogInformation` and `.LogWarning` extension methods create log messages with a log level of `Information` and `Warning`, respectively. There are six log levels to choose from, ordered here from most to least serious:

- **Critical**—For disastrous failures that may leave the app unable to function correctly, such as out-of-memory exceptions or if the hard drive is out of disk space or the server is on fire.
- **Error**—For errors and exceptions that you can't handle gracefully; for example, exceptions thrown when saving an edited entity in EF Core. The operation failed, but the app can continue to function for other requests and users.
- **Warning**—For when an unexpected or error condition arises that you can work around. You might log a `Warning` for handled exceptions, or when an entity isn't found, as in listing 17.3.
- **Information**—For tracking normal application flow; for example, logging when a user logs in, or when they view a specific page in your app. Typically these log messages provide context when you need to understand the steps leading up to an error message.
- **Debug**—For tracking detailed information that's particularly useful during development. Generally this only has short-term usefulness.
- **Trace**—For tracking very detailed information, which may contain sensitive information like passwords or keys. It's rarely used, and not used at all by the framework libraries.

Think of these log levels in terms of a pyramid, as shown in figure 17.5. As you progress down the log levels, the importance of the messages goes down, but the frequency goes up. Generally you'll find many `Debug` level log messages in your application, but (hopefully) few `Critical` or `Error` level messages.

This pyramid shape will become more meaningful when we look at filtering in section 17.4. When an app is in production, you typically don't want to record all the `Debug` level messages generated by your application. The sheer volume of messages would be overwhelming to sort through and could end up filling your disk with messages that say, "Everything's OK!" Additionally, `Trace` messages shouldn't be enabled in production, as they may leak sensitive data. By filtering out the lower log levels, you can ensure that you generate a sane number of logs in production but have access to all the log levels in development.

In general, logs of a higher level are more important than lower-level logs, so a `Warning` log is more important than an `Information` log, but there's another aspect to consider. Where the log came from, or who created the log, is a key piece of information that's recorded with each log message and is called the *category*.

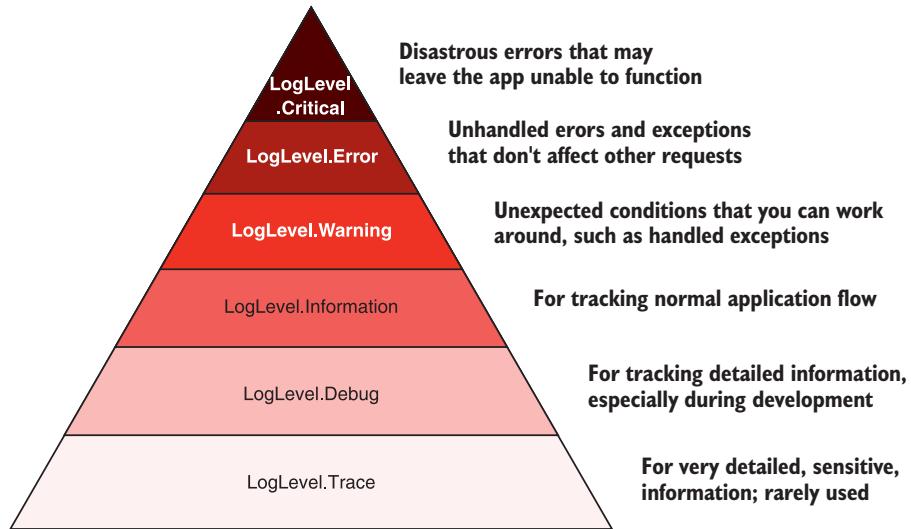


Figure 17.5 The pyramid of log levels. Logs with a level near the base of the pyramid are used more frequently but are less important. Logs with a level near the top should be rare but are important.

### 17.2.2 Log category: Which component created the log

As well as a log level, every log message also has a *category*. You set the log level independently for every log message, but the category is set when you create the `ILogger` instance. Like log levels, the category is particularly useful for filtering, as you'll see in section 17.4. It's written to every log message, as shown in figure 17.6.

**Every log message has an associated category.**

**The category is typically set to the name of the class creating the log message.**

```
Command Prompt (light) - dotnet r... + - ×
info: Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[3]
  Route matched with {page = "/Recipes/View", area = ""}. Executing page /Recipes/View
info: Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[101]
  Executing handler method RecipeApplication.Pages.Recipes.ViewModel.OnGetAsync - ModelState is Valid
info: Microsoft.EntityFrameworkCore.Database.Command[201]
  Executed DbCommand (2ms) [Parameters=@__id_0='1', CommandType='Text', CommandTimeout='30']
  SELECT * FROM [Recipes] WHERE [Id] = @__id_0 AND NOT ([IsDeleted])
  SELECT * FROM [Recipes] AS [t]
  LEFT JOIN [Ingredient] AS [i] ON [t].[RecipeId] = [i].[RecipeId]
  ORDER BY [t].[RecipeId], [i].[IngredientId]
warn: Microsoft.AspNetCore.Mvc.RazorPages.Infrastructure.PageActionInvoker[12]
  Executed handler method OnGetAsync, returned result Microsoft.AspNetCore.Mvc.NotFoundResult.
```

Figure 17.6 Every log message has an associated category, which is typically the class name of the component creating the log. The default console logging provider outputs the log category for every log.

The category is a string, so you can set it to anything, but the convention is to set it to the fully qualified name of the type that's using `ILogger`. In section 17.2 I achieved

this by injecting `ILogger<T>` into `RecipeController`; the generic parameter `T` is used to set the category of the `ILogger`.

Alternatively, you can inject `ILoggerFactory` into your methods and pass an explicit category when creating an `ILogger` instance. This lets you change the category to an arbitrary string.

#### Listing 17.4 Injecting `ILoggerFactory` to use a custom category

```
public class RecipeService
{
    private readonly ILogger _log;
    public RecipeService(ILoggerFactory factory)
    {
        _log = factory.CreateLogger("RecipeApp.RecipeService");
    }
}
```

The code is annotated with two callouts:

- A callout pointing to the constructor parameter `factory` with the text "Injects an `ILoggerFactory` instead of an `ILogger` directly".
- A callout pointing to the line `_log = factory.CreateLogger("RecipeApp.RecipeService");` with the text "Passes a category as a string when calling `CreateLogger`".

There is also an overload of `CreateLogger()` with a generic parameter that uses the provided class to set the category. If the `RecipeService` in listing 17.4 was in the `RecipeApp` namespace, the `CreateLogger` call could be written equivalently as

```
_log = factory.CreateLogger<RecipeService>();
```

Similarly, the final `ILogger` instance created by this call would be the same as if you'd directly injected `ILogger<RecipeService>` instead of `ILoggerFactory`.

**TIP** Unless you're using heavily customized categories for some reason, favor injecting `ILogger<T>` into your methods over `ILoggerFactory`.

The final compulsory part of every log entry is fairly obvious: the *log message*. At the simplest level, this can be any string, but it's worth thinking carefully about what information would be useful to record—anything that will help you diagnose issues later on.

### 17.2.3 Formatting messages and capturing parameter values

Whenever you create a log entry, you must provide a *message*. This can be any string you like, but as you saw in listing 17.2, you can also include placeholders indicated by braces, {}, in the message string:

```
_log.LogInformation("Loaded {RecipeCount} recipes", Recipes.Count);
```

Including a placeholder and a parameter value in your log message effectively creates a key-value pair, which some logging providers can store as additional information associated with the log. The previous log message would assign the value of `Recipes.Count` to a key, `RecipeCount`, and the log message itself is generated by replacing the placeholder with the parameter value, to give the following (where `Recipes.Count=3`):

```
"Loaded 3 recipes"
```

You can include multiple placeholders in a log message, and they'll be associated with the additional parameters passed to the log method. The order of the placeholders in the format string must match the order of the parameters you provide.

**WARNING** You must pass at least as many parameters to the log method as there are placeholders in the message. If you don't pass enough parameters, you'll get an exception at runtime.

For example, the log message

```
_log.LogInformation("User {UserId} loaded recipe {RecipeId}", 123, 456)
```

would create the parameters `UserId=123` and `RecipeId=456`. *Structured logging* providers could store these values, in addition to the formatted log message "User 123 loaded recipe 456". This makes it easier to search the logs for a particular `UserId` or `RecipeId`.

**DEFINITION** *Structured or semantic logging* attaches additional structure to log messages to make them more easily searchable and filterable. Rather than storing only text, it stores additional contextual information, typically as key-value pairs. JSON is a common format used for structured log messages, as it has all of these properties.

Not all logging providers use semantic logging. The default console logging provider doesn't, for example—the message is formatted to replace the placeholders, but there's no way of searching the console by key-value.

But even if you're not using structured logging initially, I recommend writing your log messages as though you are, with explicit placeholders and parameters. That way, if you decide to add a structured logging provider later, you'll immediately see the benefits. Additionally, I find that thinking about the parameters that you can log in this way prompts you to record more parameter values, instead of only a log message. There's nothing more frustrating than seeing a message like "Cannot insert record due to duplicate key" but not having the key value logged!

**TIP** Generally speaking, I'm a fan of C# 6's interpolated strings, but don't use them for your log messages when a placeholder and parameter would also make sense. Using placeholders instead of interpolated strings will give you the same output message but will also create key-value pairs that can be searched later.

We've looked a lot at how you can create log messages in your app, but we haven't focused on where those logs are written. In the next section we'll look at the built-in ASP.NET Core logging providers, how they're configured, and how you can replace the defaults with a third-party provider.

## 17.3 Controlling where logs are written using logging providers

In this section you'll learn how to control where your log messages are written by adding additional `I ILoggerProviders` to your application. As an example, you'll see how to add a simple file logger provider that writes your log messages to a file, in addition to the existing console logger provider. In section 17.3.2 you'll learn how to swap out the default logging infrastructure entirely for an alternative implementation using the open source Serilog library.

Up to this point, we've been writing all our log messages to the console. If you've run any ASP.NET Core sample apps locally, you'll have probably already seen the log messages written to the console window.

**NOTE** If you're using Visual Studio and debugging using the IIS Express option (the default), you won't see the console window (though the log messages are written to the Debug Output window instead). For that reason, I normally ensure I select the app name from the drop-down list in the debug toolbar, instead of IIS Express.

Writing log messages to the console is great when you're debugging, but it's not much use for production. No one's going to be monitoring a console window on a server, and the logs wouldn't be saved anywhere or be searchable. Clearly, you'll need to write your production logs somewhere else.

As you saw in section 17.1, *logging providers* control the destination of your log messages in ASP.NET Core. They take the messages you create using the `I ILogger` interface and write them to an output location, which varies depending on the provider.

**NOTE** This name always gets to me—the log *provider* effectively *consumes* the log messages you create and outputs them to a destination. You can probably see the origin of the name from figure 17.3, but I still find it somewhat counterintuitive.

Microsoft has written several first-party log providers for ASP.NET Core that are available out-of-the-box in ASP.NET Core. These include

- *Console provider*—Writes messages to the console, as you've already seen.
- *Debug provider*—Writes messages to the debug window when you're debugging an app in Visual Studio or Visual Studio Code, for example.
- *EventLog provider*—Writes messages to the Windows Event Log. Only outputs log messages when running on Windows, as it requires Windows-specific APIs.
- *EventSource provider*—Writes messages using Event Tracing for Windows (ETW) or LTTng tracing on Linux.

There are also many third-party logging provider implementations, such as an Azure App Service provider, an elmah.io provider, and an Elasticsearch provider. On top of that, there are integrations with other pre-existing logging frameworks like NLog and

Serilog. It's always worth looking to see whether your favorite .NET logging library or service has a provider for ASP.NET Core, as most do.

You configure the logging providers for your app in Program.cs using HostBuilder. The CreateDefaultBuilder helper method configures the console and debug providers for your application automatically, but it's likely you'll want to change or add to these.

You have two options when you need to customize logging for your app:

- Use your own HostBuilder instance, instead of Host.CreateDefaultBuilder, and configure it explicitly.
- Add an additional ConfigureLogging call after CreateDefaultBuilder.

If you only need to customize logging, the latter approach is simpler. But if you find you also want to customize the other aspects of the HostBuilder created by CreateDefaultBuilder (such as your app configuration settings), it may be worth ditching the CreateDefaultBuilder method and creating your own instance instead.

In section 17.3.1 I'll show how to add a simple third-party logging provider to your application that writes log messages to a file, so that your logs are persisted. In section 17.3.2 I'll show how to go one step further and replace the default ILoggerFactory in ASP.NET Core with an alternative implementation using the popular open source Serilog library.

### 17.3.1 Adding a new logging provider to your application

In this section we're going to add a logging provider that writes to a rolling file, so our application writes logs to a new file each day. We'll continue to log using the console and debug providers as well, because they will be more useful than the file provider when developing locally.

To add a third-party logging provider in ASP.NET Core, follow these steps:

- 1 Add the logging provider NuGet package to the solution. I'm going to be using a provider called NetEscapades.Extensions.Logging.RollingFile, which is available on NuGet and GitHub. You can add it to your solution using the NuGet Package Manager in Visual Studio, or using the .NET CLI by running

```
dotnet add package NetEscapades.Extensions.Logging.RollingFile
```

from your application's project folder.

**NOTE** This package is a simple file logging provider, available at <https://github.com/andrewlock/NetEscapades.Extensions.Logging>. It's based on the Azure App Service logging provider. If you would like more control over your logs, such as specifying the file format, consider using Serilog instead, as described in section 17.3.2.

- 2 Add the logging provider using the IHostBuilder.ConfigureLogging() extension method. You can add the file provider by calling AddFile(), as shown in

the next listing. `AddFile()` is an extension method provided by the logging provider package to simplify adding the provider to your app.

### Listing 17.5 Adding a third-party logging provider to `IHostBuilder`

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureLogging(builder => builder.AddFile())
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
}
```

The code is annotated with several callouts:

- A callout on the left side of the `CreateHostBuilder` method header contains the text: "The `CreateDefaultBuilder` method configures the console and debug providers as normal." It points to the first line of the `CreateHostBuilder` method definition.
- A callout on the right side of the `.ConfigureLogging` line contains the text: "Adds the new file logging provider to the logger factory". It points to the `builder.AddFile()` call.

**NOTE** Adding a new provider doesn't replace existing providers. Listing 17.5 uses the `CreateDefaultBuilder` helper method, so the console and debug logging providers have already been added. To remove them, call `builder.ClearProviders()` at the start of the `ConfigureLogging` method, or use a custom `HostBuilder`.

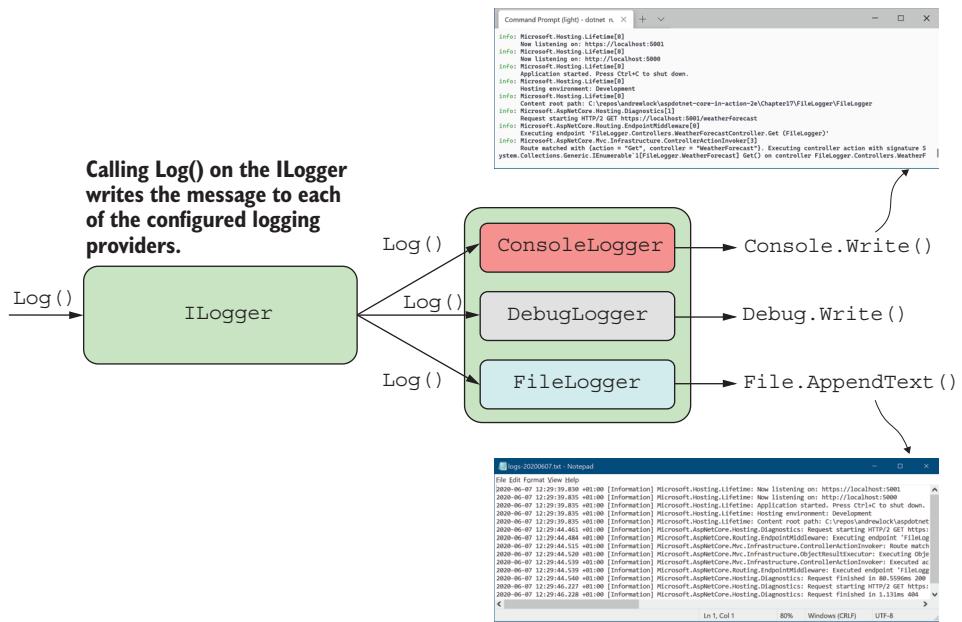
With the file logging provider configured, you can run the application and generate logs. Every time your application writes a log using an `ILogger` instance, `ILogger` writes the message to all configured providers, as shown in figure 17.7. The console messages are conveniently available, but you also have a persistent record of the logs stored in a file.

**TIP** By default, the rolling file provider will write logs to a subdirectory of your application. You can specify additional options such as filenames and file size limits using overloads of `AddFile()`. For production, I recommend using a more established logging provider, such as Serilog.

The key takeaway from listing 17.5 is that the provider system makes it easy to integrate existing logging frameworks and providers with the ASP.NET Core logging abstractions. Whichever logging provider you choose to use in your application, the principles are the same: call `ConfigureLogging` on `IHostBuilder` and add a new logging provider using extension methods like `AddConsole()`, or `AddFile()` in this case.

Logging your application messages to a file can be useful in some scenarios, and it's certainly better than logging to a non-existent console window in production, but it may still not be the best option.

If you discovered a bug in production and you needed to quickly look at the logs to see what happened, for example, you'd need to log on to the remote server, find the



**Figure 17.7 Logging a message with `ILogger` writes the log using all of the configured providers. This lets you, for example, log a convenient message to the console while also persisting the logs to a file.**

log files on disk, and trawl through them to find the problem. If you have multiple web servers, you'd have a mammoth job to fetch all the logs before you could even start to tackle the bug. Not fun. Add to that the possibility of file permission or drive space issues, and file logging seems less attractive.

Instead, it's often better to send your logs to a centralized location, separate from your application. Exactly where this location may be is up to you; the key is that each instance of your app sends its logs to the same location, separate from the app itself.

If you're running your app on Azure, you get centralized logging for free because you can collect logs using the Azure App Service provider. Alternatively, you could send your logs to a third-party log aggregator service such as Loggr (<http://loggr.net/>), elmah.io (<https://elmah.io/>), or Seq (<https://getseq.net/>). You can find ASP.NET Core logging providers for each of these services on NuGet, so adding them is the same process as adding the file provider you've seen already.

Another popular option is to use the open source Serilog library to simultaneously write to a variety of different locations. In the next section I'll show how you can replace the default `ILoggerFactory` implementation with Serilog in your application, opening up a wide range of possible options for where your logs are written.

### 17.3.2 Replacing the default `ILoggerFactory` with Serilog

In this section we'll replace the default `ILoggerFactory` in the recipe app with an implementation that uses Serilog. Serilog (<https://serilog.net>) is an open source project that can write logs to many different locations, such as files, the console, an Elasticsearch cluster,<sup>1</sup> or a database. This is similar to the functionality you get with the default `ILoggerFactory`, but due to the maturity of Serilog, you may find you can write to more places.

Serilog predates ASP.NET Core, but thanks to the logging abstractions around `ILoggerFactory` and `ILoggerProvider`, you can easily integrate with Serilog while still using the `ILogger` abstractions in your application code.

Serilog uses a similar design philosophy to the ASP.NET Core logging abstractions—you write logs to a central logging object, and the log messages are written to multiple locations, such as the console or a file. Serilog calls each of these locations a *sink*.<sup>2</sup>

When you use Serilog with ASP.NET Core, you'll typically replace the default `ILoggerFactory` with a custom factory that contains a single logging provider, `SerilogLoggerProvider`. This provider can write to multiple locations, as shown in figure 17.8. This configuration is a bit of a departure from the standard ASP.NET Core logging setup, but it prevents Serilog's features from conflicting with equivalent features of the default `LoggerFactory`, such as filtering (see section 17.4).

**TIP** If you're familiar with Serilog, you can use the examples in this section to easily integrate a working Serilog configuration with the ASP.NET Core logging infrastructure.

In this section we'll add a single sink to write the log messages to the console, but using the Serilog logging provider instead of the built-in console provider. Once you've set this up, adding additional sinks to write to other locations is trivial. Adding the Serilog logging provider to an application involves three steps:

- 1 Add the required Serilog NuGet packages to the solution.
- 2 Create a Serilog logger and configure it with the required sinks.
- 3 Call `UseSerilog()` on `IHostBuilder` to replace the default `ILoggerFactory` implementation with `SerilogLoggerFactory`. This configures the Serilog provider automatically and hooks up the already-configured Serilog logger.

To install Serilog into your ASP.NET Core app, you need to add the base NuGet package and the NuGet packages for any sinks you need. You can do this through the Visual Studio NuGet GUI, using the PMC, or using the .NET CLI. To add the

---

<sup>1</sup> Elasticsearch is a REST-based search engine that's often used for aggregating logs. You can find out more at [www.elastic.co/elasticsearch/](http://www.elastic.co/elasticsearch/).

<sup>2</sup> For a full list of available sinks, see <https://github.com/serilog/serilog/wiki/Provided-Sinks>. There are 93 different sinks at the time of writing!

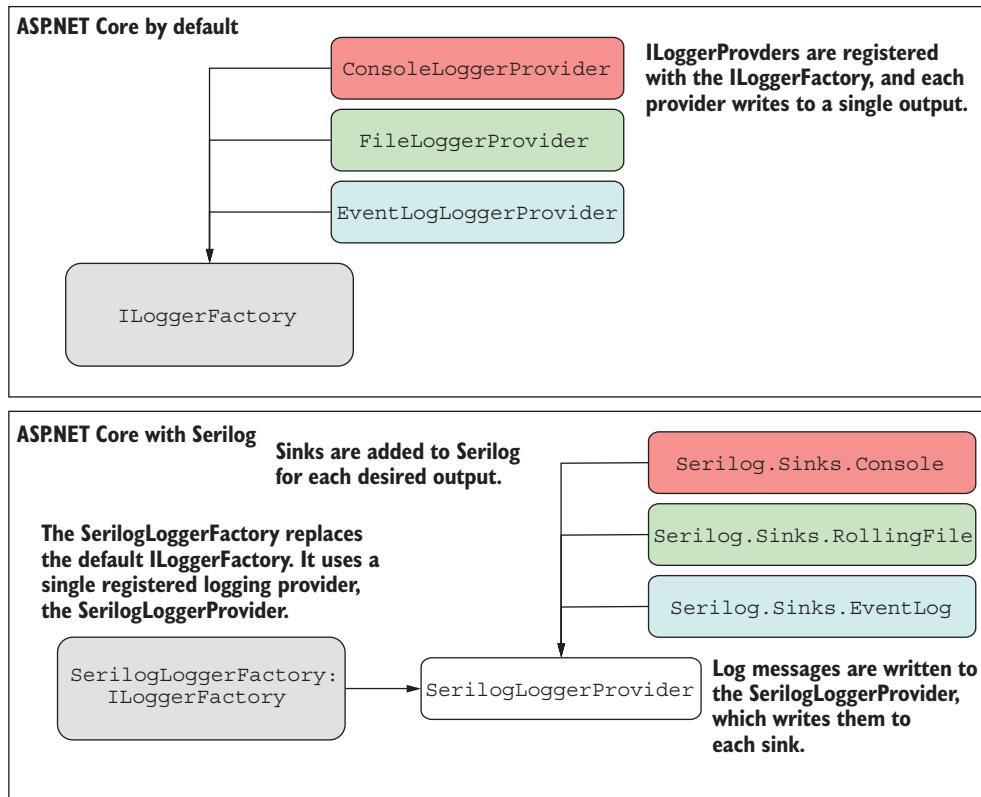


Figure 17.8 Configuration when using Serilog with ASP.NET Core compared to the default logging configuration. You can achieve the same functionality with both approaches, but you may find Serilog provides additional libraries for adding extra features.

Serilog ASP.NET Core package and a sink for writing to the console, run these commands:

```
dotnet add package Serilog.AspNetCore
dotnet add package Serilog.Sinks.Console
```

This adds the necessary NuGet packages to your project file and restores them. Next, create a Serilog logger and configure it to write to the console by adding the console sink, as shown in listing 17.6. This is the most basic of Serilog configurations, but you can add extra sinks and configuration here too.<sup>3</sup> I've also added a try-catch-finally block around our call to CreateHostBuilder, to ensure that logs are still written if

<sup>3</sup> You can customize Serilog until the cows come home, so it's worth consulting the documentation to see what's possible. The wiki is particularly useful: <https://github.com/serilog/serilog/wiki/Configuration-Basics>.

there's an error starting up the web host or there's a fatal exception. Finally, the Serilog logger factory is configured by calling `UseSerilog()` on the `IHostBuilder`.

#### Listing 17.6 Configuring a Serilog logging provider to use a console sink

```

public class Program
{
    public static void Main(string[] args)
    {
        Log.Logger = new LoggerConfiguration()           ← Creates a
                                                       LoggerConfiguration for
                                                       configuring the Serilog logger
        Serilog will                                ↗
        write logs to                                ↗
        the console.                                ↗
        .WriteTo.Console()
        .CreateLogger();                            ← This creates a Serilog logger instance
                                                    on the static Log.Logger property.

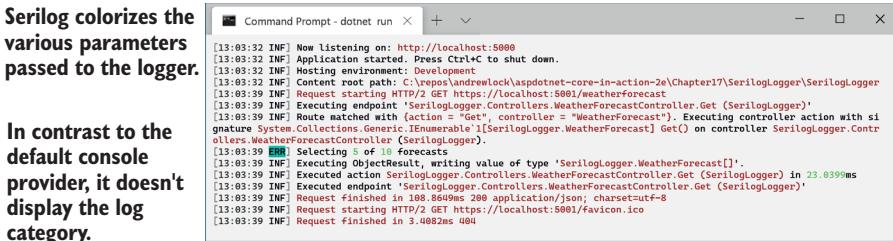
        try
        {
            CreateHostBuilder(args).Build().Run();
        }
        catch (Exception ex)
        {
            Log.Fatal(ex, "Host terminated unexpectedly");
        }
        finally
        {
            Log.CloseAndFlush();
        }
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .UseSerilog()                         ← Registers the
            .ConfigureWebHostDefaults(webBuilder =>   SerilogLoggerFactory
                                                       and connects the
                                                       Log.Logger as the sole
                                                       logging provider
            {
                webBuilder.UseStartup<Startup>();
            });
}

```

With the Serilog logging provider configured, you can run the application and generate some logs. Every time the app generates a log, `ILogger` writes the message to the Serilog provider, which writes it to every sink. In listing 17.6 you've only configured the console sink, so the output will look something like figure 17.9. The Serilog console sink colorizes the logs more than the built-in console provider, so I find it's somewhat easier to parse the logs visually.

**TIP** Serilog has many great features in addition to this. One of my favorites is the ability to add *enrichers*. These automatically add information to all your log messages, such as the process ID or environment variables, which can be useful when diagnosing problems. For an in-depth look at the recommended way to configure Serilog for ASP.NET Core apps, see the “Setting up Serilog in ASP.NET Core 3” post by Nicholas Blumhardt, the creator of Serilog: <http://mng.bz/Yqvz>.



**Figure 17.9 Example output using the Serilog provider and a console sink.** The output has more colorization than the built-in console provider, though by default it doesn't display the log category for each log.<sup>4</sup>

Serilog lets you easily plug in additional sinks to your application, in much the same way as you do with the default ASP.NET Core abstractions. Whether you choose to use Serilog or stick to other providers is up to you; the feature sets are quite similar, though Serilog is more mature. Whichever you choose, once you start running your apps in production, you'll quickly discover a different issue: the sheer number of log messages your app generates!

## 17.4 Changing log verbosity with filtering

In this section you'll see how to reduce the number of log messages written to the logger providers. You'll learn how to apply a base level filter, filter out messages from specific namespaces, and use logging provider-specific filters.

If you've been playing around with the logging samples, you'll probably have noticed that you get a lot of log messages, even for a single request like the one in figure 17.2: messages from the Kestrel server, messages from EF Core, not to mention your own custom messages. When you're debugging locally, having access to all that detailed information is extremely useful, but in production you'll be so swamped by noise that it'll make picking out the important messages difficult.

ASP.NET Core includes the ability to filter out log messages *before* they're written, based on a combination of three things:

- The log level of the message
- The category of the logger (who created the log)
- The logger provider (where the log will be written)

You can create multiple rules using these properties, and, for each log that's created, the most specific rule will be applied to determine whether the log should be written to the output. You could create the following three rules:

<sup>4</sup> If you wish to display the log category in the console sink, you can customize the `outputTemplate` and add `{SourceContext}`. For details, see the README on GitHub: <https://github.com/Serilog/Serilog-Sinks-Console#output-templates>.

- *The default minimum log level is Information*—If no other rules apply, only logs with a log level of Information or above will be written to providers.
- *For categories that start with Microsoft, the minimum log level is Warning*—Any logger created in a namespace that starts with Microsoft will only write logs that have a log level of Warning or above. This would filter out the noisy framework messages you saw in figure 17.6.
- *For the console provider, the minimum log level is Error*—Logs written to the console provider must have a minimum log level of Error. Logs with a lower level won't be written to the console, though they might be written using other providers.

Typically, the goal with log filtering is to reduce the number of logs written to certain providers, or from certain namespaces (based on the log category). Figure 17.10 shows a possible set of filtering rules that apply to the console and file logging providers.

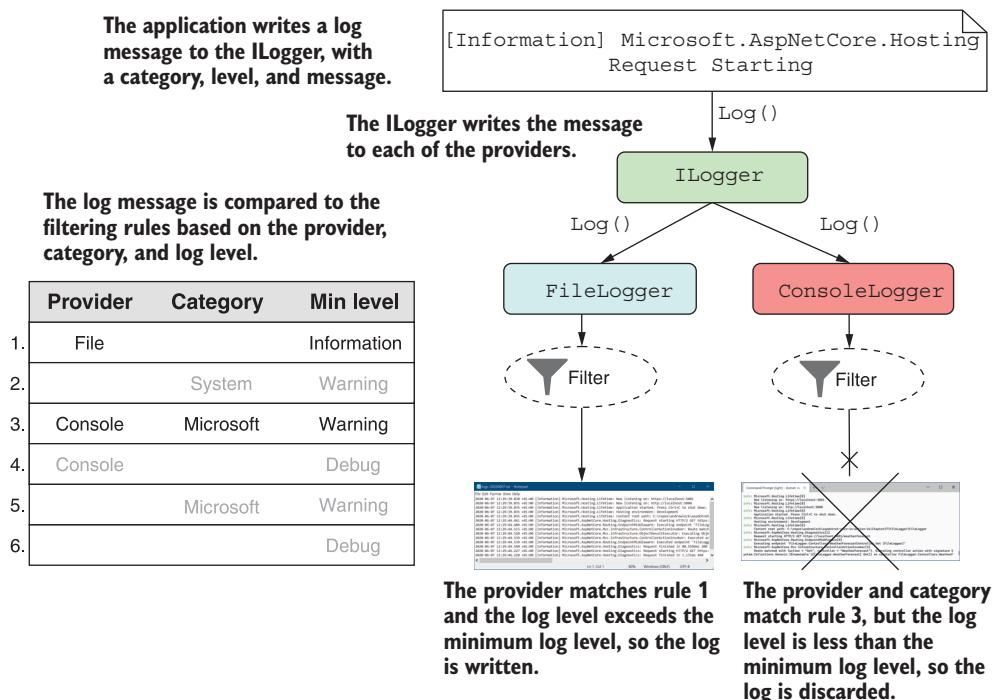


Figure 17.10 Applying filtering rules to a log message to determine whether a log should be written. For each provider, the most specific rule is selected. If the log exceeds the rule's required minimum level, the provider writes the log; otherwise it discards it.

In this example, the console logger explicitly restricts logs written in the Microsoft namespace to Warning or above, so the console logger ignores the log message shown. Conversely, the file logger doesn't have a rule that explicitly restricts the Microsoft

namespace, so it uses the configured minimum level of `Information` and writes the log to the output.

**TIP** Only a single rule is chosen when deciding whether a log message should be written; they aren't combined. In figure 17.10, rule 1 is considered more specific than rule 5, so the log is written to the file provider, even though both could technically apply.

You typically define your app's set of logging rules using the layered configuration approach discussed in chapter 11, because this lets you easily have different rules when running in development and production. You do this by calling `AddConfiguration` when configuring logging in `Program.cs`, but `CreateDefaultBuilder()` also does this for you automatically.

This listing shows how you'd add configuration rules to your application when configuring your own `HostBuilder`, instead of using the `CreateDefaultBuilder` helper method.

### Listing 17.7 Loading logging configuration in `ConfigureLogging`

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        new HostBuilder()
            .UseContentRoot(Directory.GetCurrentDirectory())
            .ConfigureAppConfiguration(config =>
                config.AddJsonFile("appsettings.json"))
            .ConfigureLogging((ctx, builder) =>
            {
                builder.AddConfiguration(
                    ctx.Configuration.GetSection("Logging"));
                builder.AddConsole(); ←
            })
            .ConfigureWebHostDefaults(webBuilder =>
            {
                webBuilder.UseStartup<Startup>();
            });
    }
}
```

In this example, I've loaded the configuration from a single file, `appsettings.json`, which contains all of our app configuration. The logging configuration is specifically contained in the "Logging" section of the `IConfiguration` object, which is available when you call `ConfigureLogging()`.

**TIP** As you saw in chapter 11, you can load configuration settings from multiple sources, like JSON files and environment variables, and can load them

conditionally based on the `IHostingEnvironment`. A common practice is to include logging settings for your production environment in `appsettings.json` and overrides for your local development environment in `appsettings.Development.json`.

The logging section of your configuration should look similar to the following listing, which shows how you could define the rules shown in figure 17.10.

#### Listing 17.8 The log filtering configuration section of `appsettings.json`

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Debug",
      "System": "Warning",
      "Microsoft": "Warning"
    },
    "File": {
      "LogLevel": {
        "Default": "Information"
      }
    },
    "Console": {
      "LogLevel": {
        "Default": "Debug",
        "Microsoft": "Warning"
      }
    }
  }
}
```

**Rules to apply if there are no applicable rules for a provider**

**Rules to apply to the File provider**

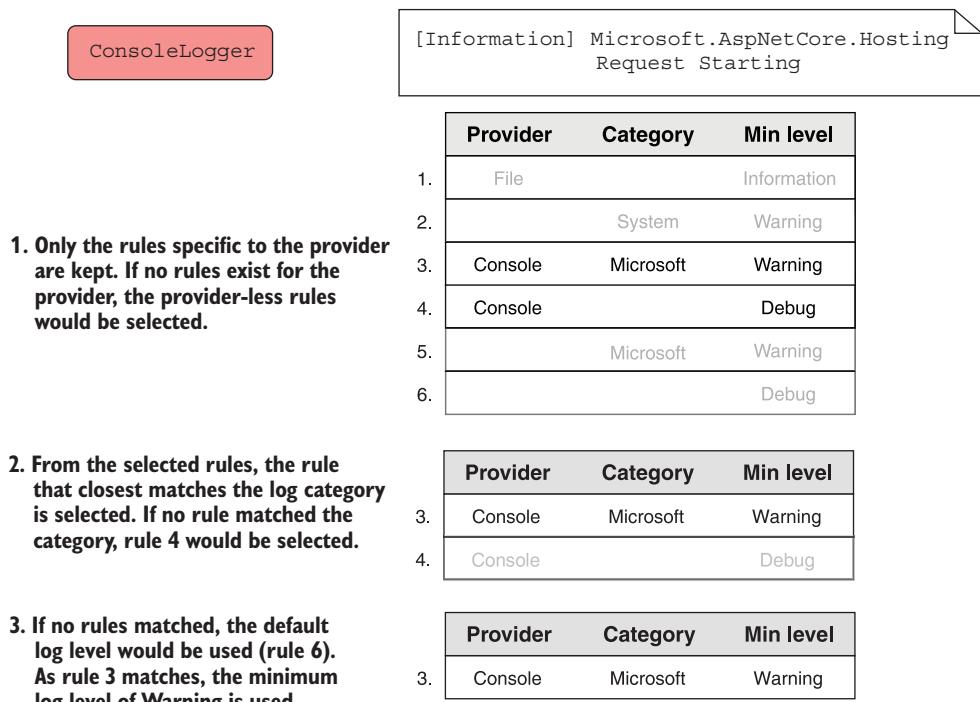
**Rules to apply to the Console provider**

When creating your logging rules, the important thing to bear in mind is that if you have *any* provider-specific rules, these will take precedence over the category-based rules defined in the `"LogLevel"` section. Therefore, for the configuration defined in listing 17.8, if your app *only* uses the file or console logging providers, the rules in the `"LogLevel"` section will effectively never apply.

If you find this confusing, don't worry, so do I. Whenever I'm setting up logging, I check the algorithm used to determine which rule will apply for a given provider and category, which is as follows:

- 1 Select all rules for the given provider. If no rules apply, select all rules that don't define a provider (the top `"LogLevel"` section from listing 17.8).
- 2 From the selected rules, select rules with the longest matching category prefix. If no selected rules match the category prefix, select the `"Default"` if present.
- 3 If multiple rules are selected, use the last one.
- 4 If no rules are selected, use the global minimum level, `"LogLevel:Default"` (which is `Debug` in listing 17.8).

Each of these steps (except the last) narrows down the applicable rules for a log message, until you're left with a single rule. You saw this in effect for a "Microsoft" category log in figure 17.10. Figure 17.11 shows the process in more detail.

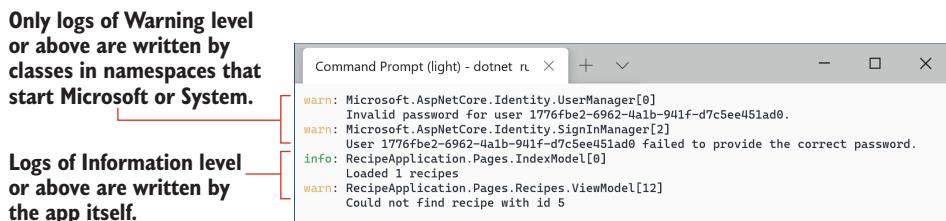


**Figure 17.11** Selecting a rule to apply from the available set for the console provider and an Information level log. Each step reduces the number of rules that apply until you're left with only one.

**WARNING** Log filtering rules aren't merged together; a single rule is selected. Including provider-specific rules will override global category-specific rules, so I tend to stick to category-specific rules in my apps to make the overall set of rules easier to understand.

With some effective filtering in place, your production logs should be much more manageable, as shown in figure 17.12. Generally I find it's best to limit the logs from the ASP.NET Core infrastructure and referenced libraries to `Warning` or above, while keeping logs that my app writes to `Debug` in development and `Information` in production.

This is close to the default configuration used in the ASP.NET Core templates. You may find you need to add additional category-specific filters, depending on which



**Figure 17.12** Using filtering to reduce the number of logs written. In this example, category filters have been added to the `Microsoft` and `System` namespaces, so only logs of Warning and above are recorded. That increases the proportion of logs that are directly relevant to your application.

NuGet libraries you use and the categories they write to. The best way to find out is generally to run your app and see if you get flooded with uninteresting log messages.

Even with your log verbosity under control, if you stick to the default logging providers like the file or console loggers, you'll probably regret it in the long run. These log providers work perfectly well, but when it comes to finding specific error messages, or analyzing your logs, you'll have your work cut out for you. In the next section you'll see how structured logging can help tackle this problem.

## 17.5 Structured logging: Creating searchable, useful logs

In this section you'll learn how structured logging makes working with log messages easier. You'll learn to attach key-value pairs to log messages and how to store and query for key values using the structured logging provider, Seq. Finally, you'll learn how to use scopes to attach key-value pairs to all log messages within a block.

Let's imagine you've rolled out the recipe application we've been working on to production. You've added logging to the app so that you can keep track of any errors in your application, and you're storing the logs in a file.

One day, a customer calls and says they can't view their recipe. Sure enough, when you look through the log messages, you see a warning:

```

warn: RecipeApplication.Controllers.RecipeController[12]
      Could not find recipe with id 3245
  
```

This piques your interest—why did this happen? Has it happened before for this *customer*? Has it happened before for this *recipe*? Has it happened for *other* recipes? Does it happen regularly?

How would you go about answering these questions? Given that the logs are stored in a text file, you might start doing basic text searches in your editor of choice, looking for the phrase "Could not find recipe with id". Depending on your notepad-fu skills, you could probably get a fair way in answering your questions, but it would likely be a laborious, error-prone, and painful process.

The limiting factor is that the logs are stored as *unstructured* text, so text processing is the only option available to you. A better approach is to store the logs in a *structured* format, so that you can easily query the logs, filter them, and create analytics. Structured logs could be stored in any format, but these days they're typically represented as JSON. For example, a structured version of the same recipe warning log might look something like this:

```
{  
    "eventLevel": "Warning",  
    "category": "RecipeApplication.Controllers.RecipeController",  
    "eventId": "12",  
    "messageTemplate": "Could not find recipe with {recipeId}",  
    "message": "Could not find recipe with id 3245",  
    "recipeId": "3245"  
}
```

This structured log message contains all the same details as the unstructured version, but in a format that would easily let you search for specific log entries. It makes it simple to filter logs by their EventLevel, or to only show those logs relating to a specific recipe ID.

**NOTE** This is only an example of what a structured log could look like. The format used for the logs will vary depending on the logging provider used and could be anything. The key point is that properties of the log are available as key-value pairs.

Adding structured logging to your app requires a logging provider that can create and store structured logs. Elasticsearch is a popular general search and analytics engine that can be used to store and query your logs. Serilog includes a sink for writing logs to Elasticsearch that you can add to your app in the same way as you added the console sink in section 17.3.2.

**TIP** If you want to learn more about Elasticsearch, *Relevant Search* by Doug Turnbull and John Berryman (Manning, 2016) is a great choice. It will show how you can make the most of your structured logs.

Elasticsearch is a powerful production-scale engine for storing your logs, but setting it up isn't for the faint of heart. Even after you've got it up and running, there's a somewhat steep learning curve associated with the query syntax. If you're interested in something more user-friendly for your structured logging needs, Seq (<https://getseq.net>) is a great option. In the next section I'll show you how adding Seq as a structured logging provider makes analyzing your logs that much easier.

### 17.5.1 Adding a structured logging provider to your app

To demonstrate the advantages of structured logging, in this section you'll configure an app to write logs to Seq. You'll see that the configuration is essentially identical to unstructured providers, but that the possibilities afforded by structured logging make considering it a no-brainer.

Seq is installed on a server or your local machine and collects structured log messages over HTTP, providing a web interface for you to view and analyze your logs. It is currently available as a Windows app or a Linux Docker container. You can install a free version for development, which will allow you to experiment with structured logging in general.<sup>5</sup>

From the point of view of your app, the process for adding the Seq provider should be familiar:

- 1 Install the Seq logging provider using Visual Studio or the .NET CLI with

```
dotnet add package Seq.Extensions.Logging
```

- 2 Add the Seq logging provider in Program.cs inside the `ConfigureLogging` method. To add the Seq provider in addition to the console and debug providers included as part of `CreateDefaultBuilder`, use

```
Host.CreateDefaultBuilder(args)
    .ConfigureLogging(builder => builder.AddSeq())
    .ConfigureWebHostDefaults(webBuilder =>
    {
        webBuilder.UseStartup<Startup>();
    });
}
```

That's all you need to add Seq to your app. This will send logs to the default local URL when you have Seq installed in your local environment. The `AddSeq()` extension method includes additional overloads to customize Seq when you move to production, but this is all you need to start experimenting locally.

If you haven't already, install Seq on your development machine and navigate to the Seq app at <http://localhost:5341>. In a different tab, open up your app and start browsing around and generating logs. Back in Seq, if you refresh the page, you'll see a list of logs, something like in figure 17.13. Clicking on a log expands it and shows you the structured data recorded for the log.

ASP.NET Core supports structured logging by treating each captured parameter from your message format string as a key-value pair. If you create a log message using the following format string,

```
_log.LogInformation("Loaded {RecipeCount} recipes", Recipes.Count);
```

the logging provider will create a `RecipeCount` parameter with a value of `Recipes.Count`. These parameters are added as properties to each structured log, as you can see in figure 17.13.

---

<sup>5</sup> You can download the Windows installer for Seq from <https://getseq.net/Download>.

Logs are listed in reverse chronological order.

Refresh the list of logs or turn on auto-refresh.

Clicking on a log reveals the structured data.

Each log includes various key-value pairs, as well as the standard level and message properties.

Warning level logs are highlighted in yellow.

Event	Level (Warning)	Type (0x0BED2097)	Retain	Raw JSON
✓ ✘ ActionId				462214d4-7174-4369-9108-665370d958fa
✓ ✘ ActionName				/Account/Login
✗ ✘ EventId			> {Id: 2}	
✓ ✘ ParentId				
✓ ✘ RequestId				0HW0836E8H88:000000025
✓ ✘ RequestPath				/Identity/Account/Login
✓ ✘ SourceContext				Microsoft.AspNetCore.Identity.SignInManager<
✓ ✘ SpanId				c1b77de-432a2e96415c349f7,
✓ ✘ TraceId				c1b77de-432a2e96415c349f7
✓ ✘ userId				1776fbe2-6962-4a1b-941f-d7c5ee451ad0

07 Jun 2020 20:36:26.982 Invalid password for user 1776fbe2-6962-4a1b-941f-d7c5ee451ad0.

**Figure 17.13** The Seq UI. Logs are presented as a list. You can view the structured logging details of individual logs, view analytics for logs in aggregate, and search by log properties.

Structured logs are generally easier to read than your standard-issue console output, but their real power comes when you need to answer a specific question. Consider the problem from before, where you see this error:

```
Could not find recipe with id 3245
```

You want to get a feel for how widespread the problem is. The first step would be to identify how many times this error has occurred and to see whether it's happened to any other recipes. Seq lets you filter your logs, but it also lets you craft SQL queries to analyze your data, so finding the answer to the question takes a matter of seconds, as shown in figure 17.14.

You can search and filter the logs using simple properties or SQL.

View the results as a table or as a graph.

RecipeId	(count)
3245	13

Computed in 0.1 ms

Results show that for logs with an EventId of 12, there were 13 occurrences, all with RecipId=3245.

**Figure 17.14** Querying logs in Seq. Structured logging makes log analysis like this example easy.

**NOTE** You don't need query languages like SQL for simple queries, but it makes digging into the data easier. Other structured logging providers may provide query languages other than SQL, but the principle is the same as in this Seq example.

A quick search shows that you've recorded the log message with EventId.Id=12 (the EventId of the warning we're interested in) 13 times, and every time the offending RecipeId was 3245. This suggests that there may be something wrong with that recipe in particular, which points you in the right direction to find the problem.

More often than not, figuring out errors in production involves logging detective work like this to isolate where the problem occurred. Structured logging makes this process significantly easier, so it's well worth considering, whether you choose Seq, Elasticsearch, or a different provider.

I've already described how you can add structured properties to your log messages using variables and parameters from the message, but as you can see in figure 17.13, there are far more properties visible than exist in the message alone.

*Scopes* provide a way to add arbitrary data to your log messages. They're available in some unstructured logging providers, but they shine when used with structured logging providers. In the final section of this chapter, I'll demonstrate how you can use them to add additional data to your log messages.

### 17.5.2 Using scopes to add additional properties to your logs

You'll often find in your apps that you have a group of operations that all use the same data, which would be useful to attach to logs. For example, you might have a series of database operations that all use the same transaction ID, or you might be performing multiple operations with the same user ID or recipe ID. *Logging scopes* provide a way of associating the same data to every log message in such a group.

**DEFINITION** *Logging scopes* are used to group multiple operations by adding the same data to each log message.

Logging scopes in ASP.NET Core are created by calling `ILogger.BeginScope<T>(T state)` and providing the state data to be logged. You create scopes inside a `using` block; any log messages written inside the scope block will have the associated data, whereas those outside won't.

#### Listing 17.9 Adding scope properties to log messages with `BeginScope`

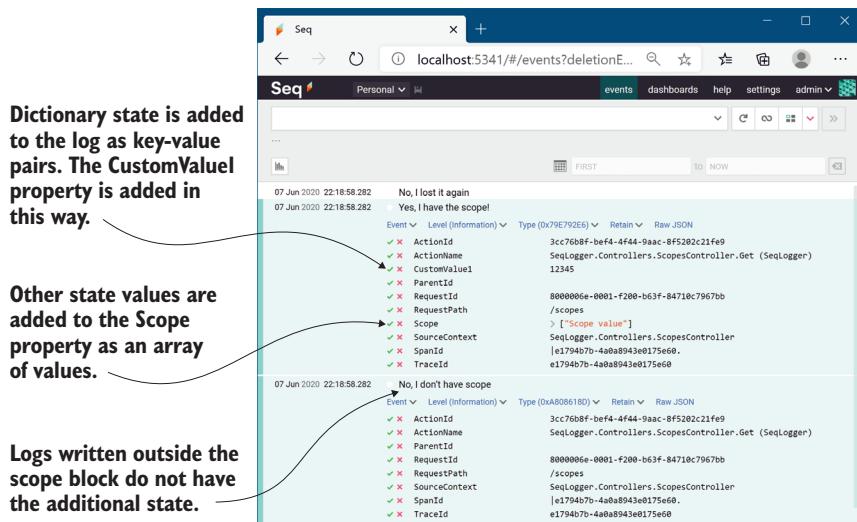
Log messages written outside the scope block don't include the scope state.

```
→ _logger.LogInformation("No, I don't have scope");           Calling BeginScope starts a scope block,  
using(_logger.BeginScope("Scope value"))                         with a scope state of "Scope value".  
using(_logger.BeginScope(new Dictionary<string, object>  
{
    {"CustomValue1", 12345 } }))                                You can pass  
{
    _logger.LogInformation("Yes, I have the scope!");            anything as the  
}                                                               state for a scope.  
→ _logger.LogInformation("No, I lost it again");           Log messages written  
inside the scope block include the scope state.
```

The scope state can be any object at all: an `int`, a `string`, or a `Dictionary`, for example. It's up to each logging provider implementation to decide how to handle the state you provide in the `BeginScope` call, but typically it will be serialized using `ToString()`.

**TIP** The most common use for scopes I've found is to attach additional key-value pairs to logs. To achieve this behavior in Seq and Serilog, you need to pass `Dictionary<string, object>` as the state object.<sup>6</sup>

When the log messages inside the scope block are written, the scope state is captured and written as part of the log, as shown in figure 17.15. The `Dictionary<>` of key-value pairs is added directly to the log message (`CustomValue1`), and the remaining state values are added to the `Scope` property. You will likely find the dictionary approach the more useful of the two, as the added properties are more easily filtered on, as you saw in figure 17.14.



**Figure 17.15 Adding properties to logs using scopes. Scope state added using the dictionary approach is added as structured logging properties, but other state is added to the Scope property. Adding properties makes it easier to associate related logs with one another.**

That brings us to the end of this chapter on logging. Whether you use the built-in logging providers or opt to use a third-party provider like Serilog or NLog, ASP.NET Core makes it easy to get detailed logs not only for your app code, but for the libraries

<sup>6</sup> Nicholas Blumhardt, the creator of Serilog and Seq, has examples and the reasoning for this on his blog in the “The semantics of `ILogger.BeginScope()`” article: <http://mng.bz/GxDD>.

that make up your app’s infrastructure, like Kestrel and EF Core. Whichever you choose, I encourage you to add more logs than you *think* you’ll need—future-you will thank me when it comes to tracking down a problem.

In the next chapter, we’ll look in detail at a variety of web security problems that you should consider when building your apps. ASP.NET Core takes care of some of these issues for you automatically, but it’s important to understand where your app’s vulnerabilities lie, so you can mitigate them as best you can.

## Summary

- Logging is critical to quickly diagnosing errors in production apps. You should always configure logging for your application so that logs are written to a durable location.
- You can add logging to your own services by injecting `ILogger<T>`, where `T` is the name of the service. Alternatively, inject `ILoggerFactory` and call `CreateLogger()`.
- The log level of a message indicates how important it is and ranges from `Trace` to `Critical`. Typically you’ll create many low-importance log messages and a few high-importance log messages.
- You specify the log level of a log by using the appropriate extension method of `ILogger` to create your log. To write an `Information` level log, use `ILogger.LogInformation(message)`.
- The log category indicates which component created the log. It is typically set to the fully qualified name of the class creating the log, but you can set it to any string if you wish. `ILogger<T>` will have a log category of `T`.
- You can format messages with placeholder values, similar to the `string.Format` method, but with meaningful names for the parameters. Calling `logger.LogInformation("Loading Recipe with id {RecipeId}", 1234)` would create a log reading "Loading Recipe with id 1234", but it would also capture the value `RecipeId =1234`. This *structured logging* makes analyzing log messages much easier.
- ASP.NET Core includes many logging providers out of the box. These include the console, debug, EventLog, and EventSource providers. Alternatively, you can add third-party logging providers.
- You can configure multiple `ILoggerProvider` instances in ASP.NET Core, which define where logs are output. The `CreateDefaultBuilder` method adds the console and debug providers, and you can add additional providers by calling `ConfigureLogging()`.
- Serilog is a mature logging framework that includes support for a large number of output locations. You can add Serilog to your app with the `Serilog.AspNetCore` package. This replaces the default `ILoggerFactory` with a Serilog-specific version.
- You can control logging output verbosity using configuration. The `CreateDefaultBuilder` helper uses the "Logging" configuration section to control

output verbosity. You typically will filter out more logs in production compared to when developing your application.

- Only a single log filtering rule is selected for each logging provider when determining whether to output a log message. The most specific rule is selected based on the logging provider and the category of the log message.
- Structured logging involves recording logs so that they can be easily queried and filtered, instead of the default unstructured format that's output to the console. This makes analyzing logs, searching for issues, and identifying patterns easier.
- You can add additional properties to a structured log by using scope blocks. A scope block is created by calling `ILogger.BeginScope<T>(state)` in a using block. The state can be any object and is added to all log messages inside the scope block.

# *Improving your application's security*

## **This chapter covers**

- Encrypting traffic using HTTPS and configuring local SSL certificates
- Defending against cross-site scripting attacks
- Protecting from cross-site request forgery attacks
- Allowing calls to your API from other apps using CORS

Web application security is a hot topic at the moment. Practically every week another breach is reported, or confidential details are leaked. It may seem like the situation is hopeless, but the reality is that the vast majority of breaches could have been avoided with the smallest amount of effort.

In this chapter we'll look at a few different ways to protect your application and your application's users from attackers. Because security is an extremely broad topic that covers lots of different avenues, this chapter is by no means an exhaustive guide. It's intended to make you aware of some of the most common threats to your app and how to counteract them, and also to highlight areas where you can inadvertently introduce vulnerabilities if you're not careful.

**TIP** I strongly advise exploring additional resources around security after you've read this chapter. The Open Web Application Security Project (OWASP) ([www.owasp.org](http://www.owasp.org)) is an excellent resource, though it can be a little dry. Alternatively, Troy Hunt has some excellent courses and workshops on security, geared toward .NET developers ([www.troyhunt.com/](http://www.troyhunt.com/)).

We'll start by looking at how to add HTTPS encryption to your website so that users can access your app without the risk of third parties spying on or modifying the content as it travels over the internet. This is effectively mandatory for production apps these days, and it is heavily encouraged by the makers of modern browsers such as Chrome and Firefox. You'll see how to use the ASP.NET Core development certificate to use HTTPS locally, how to configure an app for HTTPS in production, and how to enforce HTTPS across your whole app.

In sections 18.2 and 18.3 you'll learn about two potential attacks that should be on your radar: cross-site scripting (XSS) and cross-site request forgery (CSRF). We'll explore how the attacks work and how you can prevent them in your apps. ASP.NET Core has built-in protection against both types of attack, but you have to remember to use the protection correctly and resist the temptation to circumvent it, unless you're certain it's safe to do so.

Section 18.4 deals with a common scenario—you have an application that wants to use JavaScript AJAX (Asynchronous JavaScript and XML) requests to retrieve data from a second app. By default, web browsers block requests to other apps, so you need to enable cross-origin resource sharing (CORS) in your API to achieve this. We'll look at how CORS works, how to create a CORS policy for your app, and how to apply it to specific action methods.

The final section of this chapter, section 18.5, covers a collection of common threats to your application. Each one represents a potentially critical flaw that an attacker could use to compromise your application. The solutions to each threat are generally relatively simple; the important thing is to recognize where the flaws could exist in your own apps so that you can ensure that you don't leave yourself vulnerable.

We'll start by looking at HTTPS and why you should use it to encrypt the traffic between your users' browsers and your app. Without HTTPS, attackers could subvert many of the safeguards you add to your app, so it's an important first step to take.

## 18.1 Adding HTTPS to an application

In this section you'll learn about HTTPS: what it is, and why you need to be aware of it for all your production applications. You'll see two approaches to adding HTTPS to your application: supporting HTTPS directly in your application and using SSL/TLS-offloading with a reverse proxy. You'll then learn how to use the development certificate to work with HTTPS on your local machine, and how to add an HTTPS certificate to your app in production. Finally, you'll learn how to enforce HTTPS in your app using best practices such as security headers and HTTP redirection.

So far in this book, I've shown how the user's browser sends a request across the internet to your app using the HTTP protocol. We haven't looked too much into the details of that protocol, other than to establish that it uses *verbs* to describe the type of request (such as GET and POST), that it contains *headers* with metadata about the request, and optionally includes a *body* payload of data.

By default, HTTP requests are unencrypted; they're plain text files being sent over the internet. Anyone on the same network as a user (such as someone using the same public Wi-Fi in a coffee shop) can read the requests and responses sent back and forth. Attackers can even *modify* the requests or responses as they're in transit.

Using unencrypted web apps in this way presents both a privacy and a security risk to your users. Attackers could read the data sent in forms and returned by your app, inject malicious code into your responses to attack users, or steal authentication cookies and impersonate the user on your app.

To protect your users, your app should encrypt the traffic between the user's browser and your app as it travels over the network by using the HTTPS protocol. This is similar to HTTP traffic, but it uses an SSL/TLS<sup>1</sup> certificate to encrypt requests and responses, so attackers cannot read or modify the contents. In browsers, you can tell that a site is using HTTPS by the https:// prefix to URLs (notice the "s"), or sometimes, by a padlock, as shown in figure 18.1.

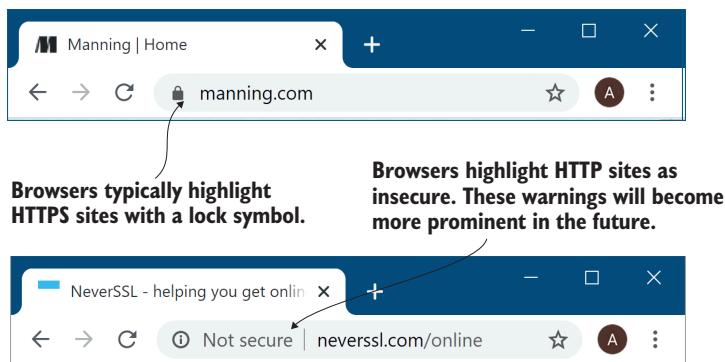


Figure 18.1 Encrypted apps using HTTPS and unencrypted apps using HTTP in Edge. Using HTTPS protects your application from being viewed or tampered with by attackers.

**TIP** For details on how the SSL/TLS protocols work, see chapter 9 of *Real-World Cryptography* by David Wong (Manning, 2021), <http://mng.bz/zxzl>.

<sup>1</sup> SSL is an older standard that facilitates HTTPS, but the SSL protocol has been superseded by Transport Layer Security (TLS), so I'll be using TLS preferentially throughout this chapter.

The reality is that, these days, you should always serve your production websites over HTTPS. The industry is pushing toward HTTPS by default, with most browsers moving to mark HTTP sites as explicitly “not secure.” Skipping HTTPS will hurt the perception of your app in the long run, so even if you’re not interested in the security benefits, it’s in your best interest to set up HTTPS.

To enable HTTPS you need to obtain and configure a TLS certificate for your server. Unfortunately, although that process is a lot easier than it used to be, and it’s now essentially free thanks to Let’s Encrypt (<https://letsencrypt.org/>), it’s still far from simple in many cases. If you’re setting up a production server, I recommend carefully following the tutorials on the Let’s Encrypt site. It’s easy to get it wrong, so take your time.

**TIP** If you’re hosting your app in the cloud, most providers will provide one-click TLS certificates so that you don’t have to manage certificates yourself. This is *extremely* useful, and I highly recommend it for everyone.<sup>2</sup>

As an ASP.NET Core application developer, you can often get away without *directly* supporting HTTPS in your app by taking advantage of the reverse-proxy architecture, as shown in figure 18.2, in a process called SSL/TLS offloading/termination. Instead of your application handling requests using HTTPS directly, it continues to use HTTP. The reverse proxy is responsible for encrypting and decrypting HTTPS traffic to the browser. This often gives you the best of both worlds—data is encrypted between the user’s browser and the server, but you don’t have to worry about configuring certificates in your application.<sup>3</sup>

Depending on the specific infrastructure where you’re hosting your app, SSL/TLS could be offloaded to a dedicated device on your network, a third-party service like Cloudflare, or a reverse proxy (such as IIS, NGINX, or HAProxy) running on the same or a different server.

Nevertheless, in some situations, you may need to handle SSL/TLS directly in your app:

- *If you’re exposing Kestrel to the internet directly, without a reverse proxy.* This became more common with ASP.NET Core 3.0 due to hardening of the Kestrel server. It is also often the case when you’re developing your app locally.
- *If having HTTP between the reverse proxy and your app is not acceptable.* While securing traffic *inside* your network is less critical compared to external traffic, it is undoubtedly more secure to use HTTPS for internal traffic too.
- *If you’re using technology that requires HTTPS.* Some newer network protocols, such as gRPC and HTTP/2 require an HTTPS connection.

---

<sup>2</sup> You don’t even have to be hosting your application in the cloud to take advantage of this. Cloudflare ([www.cloudflare.com](http://www.cloudflare.com)) provides a CDN service that you can add TLS to. You can even use it for free.

<sup>3</sup> If you’re concerned that the traffic is unencrypted between the reverse proxy and your app, then I recommend reading Troy Hunt’s “CloudFlare, SSL and unhealthy security absolutism” post: <http://mng.bz/eHCi>. It discusses the pros and cons of the issue as it relates to using Cloudflare to provide HTTPS encryption.

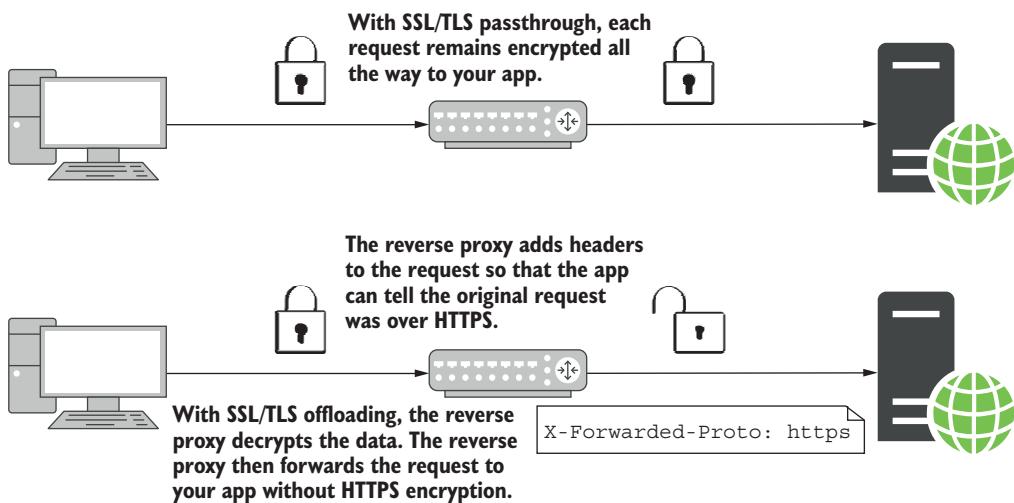


Figure 18.2 You have two options when using HTTPS with a reverse proxy: SSL/TLS passthrough and SSL/TLS offloading. In SSL/TLS passthrough, the data is encrypted all the way to your ASP.NET Core app. For SSL/TLS offloading, the reverse proxy handles decrypting the data, so your app doesn't have to.

In each of these scenarios, you'll need to configure a TLS certificate for your application so Kestrel can receive HTTPS traffic. In section 18.1.1 you'll see the easiest way to get started with HTTPS when developing locally, and in section 18.1.2 you'll see how to configure your application for production.

### 18.1.1 Using the ASP.NET Core HTTPS development certificates

Working with HTTPS certificates is easier than it used to be, but unfortunately it can still be a confusing topic, especially if you're a newcomer to the web. The .NET SDK, Visual Studio, and IIS Express try to improve this experience by handling a lot of the grunt-work for you.

The first time you run a `dotnet` command using the .NET SDK, the SDK installs an HTTPS development certificate onto your machine. Any ASP.NET Core application you create using the default templates (or for which you don't explicitly configure certificates) will use this development certificate to handle HTTPS traffic. However, the development certificate is not *trusted* by default. That means you'll get a browser warning, as shown in figure 18.3 when accessing a site after first installing the .NET SDK.

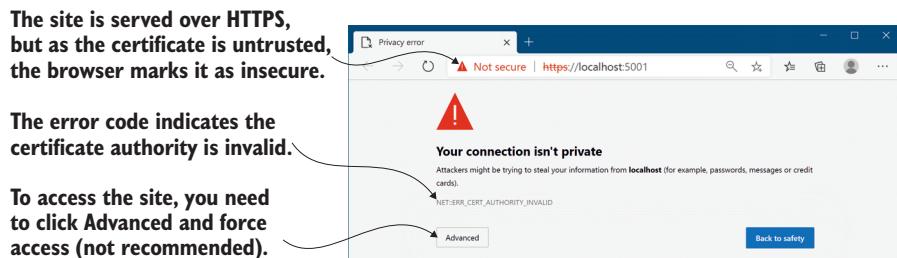


Figure 18.3 The developer certificate is not trusted by default, so apps serving HTTPS traffic using it will be marked as insecure by browsers. Although you can bypass the warnings if necessary, you should instead update the certificate to be trusted.

### A brief primer on certificates and signing

HTTPS uses *public key cryptography* as part of the data-encryption process. This uses two keys: a *public* key that anyone can see, and a *private* key that only *your* server can see. Anything encrypted with the public key can only be decrypted with the private key. That way, a browser can encrypt something with your server's public key, and only your server can decrypt it. A complete TLS certificate consists of both the public and private parts.

When a browser connects to your app, the server sends the public key part of the TLS certificate. But how does the browser know that it was definitely *your* server that sent the certificate? To achieve this, your TLS certificate contains additional certificates, including a certificate from a third party, a certificate authority (CA). This trusted certificate is called a *root certificate*.

CAs are special trusted entities, and browsers are hardcoded to trust certain root certificates. In order for the TLS certificate for your app to be trusted, it must contain (or be signed by) a trusted root certificate.

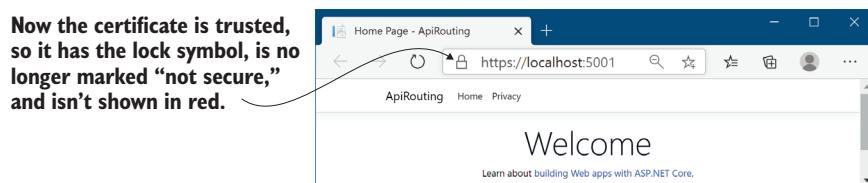
When you use the ASP.NET Core development certificate, or if you create your own self-signed certificate, your site's HTTPS is missing that trusted root certificate. That means browsers won't trust your certificate and won't connect to your server by default. To get around this, you need to tell your development machine to explicitly trust the certificate.

In production, you can't use a development or self-signed certificate, as a user's browser won't trust it. Instead, you need to obtain a signed HTTPS certificate from a service like Let's Encrypt, or from a cloud provider like AWS, Azure, or Cloudflare. These certificates will already be signed by a trusted CA, so they will be automatically trusted by browsers.

To solve these browser warnings, you need to *trust* the certificate. Trusting a certificate is a sensitive operation; it's saying, "I know this certificate doesn't look quite right, but just ignore that," so it's hard to do automatically. If you're running on Windows or macOS, you can trust the development certificate by running

```
dotnet dev-certs https --trust
```

This command trusts the certificate by registering it in the operating system's "certificate store." After you run this command, you should be able to access your websites without seeing any warnings or "not secure" labels, as shown in figure 18.4.



**Figure 18.4** Once the development certificate is trusted, you will no longer see browser warnings about the connection.

**TIP** You may need to close your browser after trusting the certificate to clear the browser's cache.

The developer certificate works smoothly on Windows and macOS. Unfortunately, trusting the certificate in Linux is a little trickier and depends on the particular flavor you're using. On top of that, software on Linux often uses its own certificate store, so you'll probably need to add the certificate directly to your favorite browser. I suggest looking at the documentation for your favorite browser to figure out the best approach. For advice on other platforms, such as Docker, see Microsoft's "How to set up a developer certificate for Docker" section in the "Enforce HTTPS in ASP.NET Core" documentation: <http://mng.bz/0mBJ>.

If you're using Windows, Visual Studio, and IIS Express for development, then you may not find the need to trust the development certificate. IIS Express acts as a reverse proxy when you're developing locally, so it handles the SSL/TLS setup itself. On top of that, Visual Studio should trust the IIS development certificate as part of installation, so you may never see the browser warnings at all.

The ASP.NET Core and IIS development certificates make it easy to use Kestrel with HTTPS locally, but those certificates won't help once you move to production. In the next section I'll show you how to configure Kestrel to use a production TLS certificate.

### 18.1.2 Configuring Kestrel with a production HTTPS certificate

Creating a TLS certificate for production is often a laborious process, as it requires proving to a third-party certificate authority (CA) that you own the domain you’re creating the certificate for. This is an important step in the “trust” process and ensures that attackers can’t impersonate your servers. The result of the process is one or more files, which is the HTTPS certificate you need to configure for your app.

**TIP** The specifics of how to obtain a certificate vary by provider and by your OS platform, so follow your provider’s documentation carefully. The vagaries and complexities of this process are one of the reasons I strongly favor the SSL/TLS-offloading or “one-click” approaches described previously. Those approaches mean my apps don’t need to deal with certificates, and I don’t need to use the approaches described in this section; I delegate that responsibility to another piece of the network, or to the underlying platform.

Once you have a certificate, you need to configure Kestrel to use it to serve HTTPS traffic. In chapter 16 you saw how to set the port your application listens on with the ASPNETCORE\_URLS environment variable or via the command line, and you saw that you could provide an HTTPS URL. As you didn’t provide any certificate configuration, Kestrel used the development certificate by default. In production you need to tell Kestrel which certificate to use.

Kestrel is very configurable, allowing you to configure your certificates in multiple ways. You can use different certificates for different ports, you can load from a .pfx file or from the OS certificate store, or you can have a different configuration for each URL endpoint you expose. For full details, see the “Endpoint configuration” section in Microsoft’s “Kestrel web server implementation in ASP.NET Core” documentation: <http://mng.bz/KMdX>.

The following listing shows one possible way to set a custom HTTPS certificate for your production app, by configuring the default certificate Kestrel uses for HTTPS connections. You can add the “Kestrel:Certificates:Default” section to your appsettings.json file (or using any other configuration source, as described in chapter 11) to define the .pfx file of the certificate to use. You must also provide the password for accessing the certificate.

**Listing 18.1 Configuring the default HTTPS certificate for Kestrel using a .pfx file**

```
{  
  "Kestrel": {  
    "Certificates": {  
      "Default": {  
        "Path": "localhost.pfx",  
        "Password": "testpassword"  
      }  
    }  
  }  
}
```

Create a configuration section at Kestrel:Certificates:Default.

The relative or absolute path to the certificate

The password for opening the certificate

The preceding example is the simplest way to replace the HTTPS certificate, as it doesn't require changing any of Kestrel's defaults. You can use a similar approach to load the HTTPS certificate from the OS certificate store (on Windows or macOS), as shown in the "Endpoint configuration" documentation mentioned previously (<http://mng.bz/KMdX>).

**WARNING** Listing 18.1 has hardcoded the certificate filename and password for simplicity, but you should either load these from a configuration store like user-secrets, as you saw in chapter 11, or load the certificate from the local store. *Never* put production passwords in your appsettings.json files.

All the default ASP.NET Core templates configure your application to serve both HTTP and HTTPS traffic, and with the configuration you've seen so far, you can ensure your application can handle both HTTP and HTTPS in development and in production.

However, whether you use HTTP or HTTPS may depend on the URL users click when they first browse to your app. For example, if your app listens using the default URLs, `http://localhost:5000` for HTTP traffic and `https://localhost:5001` for HTTPS traffic, if a user navigates to the HTTP URL, their traffic will be unencrypted. Seeing as you've gone to all the trouble to set up HTTPS, it's probably best that you force users to use it.

### 18.1.3 Enforcing HTTPS for your whole app

Enforcing HTTPS across your whole website is practically required these days. Browsers are beginning to explicitly label HTTP pages as insecure; for security reasons you *must* use TLS any time you're transmitting sensitive data across the internet, and, thanks to HTTP/2, adding TLS can *improve* your app's performance.<sup>4</sup>

There are multiple approaches to enforcing HTTPS for your application. If you're using a reverse proxy with SSL/TLS-offloading, it might be handled for you anyway, without having to worry about it within your apps. Nevertheless, it doesn't hurt to enforce SSL/TLS in your applications too, regardless of what the reverse proxy may be doing.

**NOTE** If you're building a Web API, rather than a Razor Pages app, it's common to just reject HTTP requests, without using the approaches described in this section. These protections apply primarily when building apps to be consumed in a browser. For more details, see Microsoft's "Enforce HTTPS in ASP.NET Core" documentation: <http://mng.bz/j46a>.

One approach to improving the security of your app is to use *HTTP security headers*. These are HTTP headers sent as part of your HTTP response that tell the browser how

---

<sup>4</sup> HTTP/2 offers many performance improvements over HTTP/1.x, and all modern browsers require HTTPS to enable it. For a great introduction to HTTP/2, see Google's "Introduction to HTTP/2": <http://mng.bz/9M8j>.

it should behave. There are many different headers available, most of which restrict the features your app can use in exchange for increased security.<sup>5</sup> In the next chapter you'll see how to add your own custom headers to your HTTP responses by creating custom middleware.

One of these security headers, the HTTP Strict Transport Security (HSTS) header, can help ensure browsers use HTTPS where it's available, instead of defaulting to HTTP.

### ENFORCING HTTPS WITH HTTP STRICT TRANSPORT SECURITY HEADERS

It's unfortunate, but by default, browsers always load apps over HTTP, unless otherwise specified. That means your apps typically must support both HTTP and HTTPS, even if you don't want to serve any traffic over HTTP. One mitigation for this (and a security best practice), is to add HTTP Strict Transport Security headers to your responses.

**DEFINITION** HTTP Strict Transport Security (HSTS) is a header that instructs the browser to use HTTPS for all *subsequent* requests to your application. The browser will no longer send HTTP requests to your app and will only use HTTPS instead. It can only be sent with responses to HTTPS requests. It is only relevant for requests originating from a browser—it has no effect on server-to-server communication.

HSTS headers are strongly recommended for *production* apps. You generally don't want to enable them for local development, as that would mean you could never run a non-HTTPS app locally. In a similar fashion, you should only use HSTS on sites for which you *always* intend to use HTTPS, as it's hard (sometimes impossible) to turn-off HTTPS once it's enforced with HSTS.

ASP.NET Core comes with built-in middleware for setting HSTS headers, which is included in some of the default templates automatically. The following listing shows how you can configure the HSTS headers for your application using the `HstsMiddleware` in `Startup.cs`.

#### Listing 18.2 Using `HstsMiddleware` to add HSTS headers to an application

```
public class Startup
{
    public void ConfigureServices(IServiceCollection services)
    {
        services.AddRazorPages();
        services.AddHsts(options =>
        {
            options.MaxAge = TimeSpan.FromHours(1);
        });
    }
}
```

Configure your HSTS header settings. This changes the MaxAge from the default of 30 days.

<sup>5</sup> Scott Helme has some great guidance on this and other security headers you can add to your site, such as the Content Security Policy (CSP) header. See "Hardening your HTTP response headers" on his website: <https://scotthelme.co.uk/hardening-your-http-response-headers/>.

```

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    if (env.IsProduction())
    {
        app.UseHsts();           ← You shouldn't use HSTS
                                in local environments.

    }                         ← Adds the HstsMiddleware

    app.UseStaticFiles();
    app.UseRouting();
    app.UseAuthorization();
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
    });
}
}

```

The HstsMiddleware  
should be very early in  
the middleware pipeline.

**TIP** The preceding example shows how to change the `MaxAge` sent in the HSTS header. It's a good idea to start with a small value initially. Once you're sure your app's HTTPS is functioning correctly, increase the age for greater security. For more details on HSTS see Scott Helme's article, "HSTS—The missing link in Transport Layer Security": <https://scotthelme.co.uk/hsts-the-missing-link-in-tls/>.

HSTS is a great option for forcing users to use HTTPS on your website. But one problem with the header is that it can only be added to HTTPS requests. That means you must have *already* made an HTTPS request before HSTS kicks in: if the *initial* request is HTTP, no HSTS header is sent, and you *stay on HTTP!* That's unfortunate, but you can mitigate it by redirecting insecure requests to HTTPS immediately.

#### REDIRECTING FROM HTTP TO HTTPS WITH THE HTTPS REDIRECTION MIDDLEWARE

The `HstsMiddleware` should generally be used in conjunction with middleware that redirects all HTTP requests to HTTPS.

**TIP** It's possible to apply HTTPS redirection to only parts of your application, such as to specific Razor Pages, but I don't recommend that, as it's too easy to open up a security hole in your application.

ASP.NET Core comes with `HttpsRedirectionMiddleware`, which you can use to enforce HTTPS across your whole app. You add it to the middleware pipeline in the `Configure` section of `Startup`, and it ensures that any requests that pass through it are secure. If an HTTP request reaches the `HttpsRedirectionMiddleware`, the middleware immediately short-circuits the pipeline with a redirect to the HTTPS version of the request. The browser will then repeat the request using HTTPS instead of HTTP.

**NOTE** The eagle-eyed among you will notice that even with the HSTS and redirection middleware, there is still an inherent weakness. By default, browsers will always make an initial, *insecure*, request over HTTP to your app. The only way to avoid this is by HSTS-preloading, which tells browsers to *always* use HTTPS. You can find a great guide to HSTS, including preloading, on the

ForwardPMX site: “The Ultimate Guide to HSTS Protocol” by Chris Herbrand, <http://mng.bz/Wdmg>.

The `HttpsRedirectionMiddleware` is added in the default ASP.NET Core templates. It is typically placed after the error handling and `HstsMiddleware`, as shown in the following listing. By default, the middleware redirects all HTTP requests to the secure endpoint, using an HTTP 307 Temporary Redirect status code.

### Listing 18.3 Using `HttpsRedirectionMiddleware` to enforce HTTPS for an application

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.UseExceptionHandler("/Error");
    if (env.IsProduction())
    {
        app.UseHsts();
    }

    app.UseHttpsRedirection();           ← Adds the HttpsRedirectionMiddleware
    app.UseStaticFiles();
    app.UseRouting();
    app.UseAuthorization();
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapRazorPages();
    });
}
```

Adds the `HttpsRedirectionMiddleware` to the pipeline. Redirects all HTTP requests to HTTPS.

The `HttpsRedirectionMiddleware` will automatically redirect HTTP requests to the first configured HTTPS endpoint for your application. If your application isn’t configured for HTTPS, the middleware *won’t* redirect and instead will log a warning:

```
warn: Microsoft.AspNetCore.HttpsPolicy.HttpsRedirectionMiddleware[3]
      Failed to determine the https port for redirect.
```

If you want the middleware to redirect to a different port than Kestrel knows about, you can configure that by setting the `ASPNETCORE_HTTPS_PORT` environment variable. This is sometimes necessary if you’re using a reverse proxy, and it can be set in alternative ways, as described in Microsoft’s “Enforce HTTPS in ASP.NET Core” documentation: <http://mng.bz/QmN4>.

#### SSL/TLS offloading, header forwarding, and detecting secure requests

At the start of section 18.1, I encouraged you to consider terminating HTTPS requests at a reverse proxy. That way, the user uses HTTPS to talk to the reverse proxy, and the reverse proxy talks to your app using HTTP. With this setup, your users are protected but your app doesn’t have to deal with TLS certificates itself.

**(continued)**

In order for the `HttpsRedirectionMiddleware` to work correctly, Kestrel needs some way of knowing whether the *original* request that the reverse proxy received was over HTTP or HTTPS. The reverse proxy communicates to your app over HTTP, so Kestrel can't figure that out without extra help.

The standard approach used by most reverse proxies (such as IIS, NGINX, and HAProxy) is to add headers to the request before forwarding it to your app. Specifically, a header called `X-Forwarded-Proto` is added, indicating whether the original request protocol was HTTP or HTTPS.

ASP.NET Core includes `ForwardedHeadersMiddleware` to look for this header (and others) and update the request accordingly, so your app treats a request that was *originally* secured by HTTPS as secure for all intents and purposes.

If you're using IIS with the `UseIisIntegration()` extension, the header forwarding is handled for you automatically. If you're using a different reverse proxy, such as NGINX or HAProxy, you can enable the middleware by setting the environment variable `ASPNETCORE_FORWARDEDHEADERS_ENABLED=true`, as you saw in chapter 16. Alternatively, you can manually add the middleware to your application, as shown in section 16.3.2.

When the reverse proxy forwards a request, `ForwardedHeadersMiddleware` will look for the `X-Forwarded-Proto` header and will update the request details as appropriate. For all subsequent middleware, the request is considered secure. When adding the middleware manually, it's important that you place `ForwardedHeadersMiddleware` before the call to `UseHsts()` or `UseHttpsRedirection()`, so that the forwarded headers are read and the request is marked secure, as appropriate.

HTTPS is one of the most basic requirements for adding security to your application these days. It can be tricky to set up initially, but once you're up and running, you can largely forget about it, especially if you're using SSL/TLS termination at a reverse proxy.

Unfortunately, most other security practices require rather more vigilance to ensure you don't accidentally introduce vulnerabilities into your app as it grows and develops. Many attacks are conceptually simple and have been known about for years, yet they're still commonly found in new applications. In the next section we'll look at one such attack and see how to defend against it when building apps using Razor Pages.

## 18.2 Defending against cross-site scripting (XSS) attacks

In this section I'll describe cross-site scripting attacks and how attackers can use them to compromise your users. I'll show how the Razor Pages framework protects you from these attacks, how to disable the protections when you need to, and what to look out for. I'll also discuss the difference between HTML encoding and JavaScript encoding, and the impact of using the wrong encoder.

Attackers can exploit a vulnerability in your app to create cross-site scripting (XSS) attacks that execute code in another user's browser.<sup>6</sup> Commonly, attackers submit content using a legitimate approach, such as an input form, which is later rendered somewhere to the page. By carefully crafting malicious input, the attacker can execute arbitrary JavaScript on a user's browser and so can steal cookies, impersonate the user, and generally do bad things.

Figure 18.5 shows a basic example of an XSS attack. Legitimate users of your app can send their name to your app by submitting a form. The app then adds the name to an internal list and renders the whole list to the page. If the names are not rendered safely, a malicious user can execute JavaScript in the browser of every other user that views the list.



**Figure 18.5** How an XSS vulnerability is exploited. An attacker submits malicious content to your app, which is displayed in the browsers of other users. If the app doesn't encode the content when writing to the page, the input becomes part of the HTML of the page and can run arbitrary JavaScript.

In figure 18.5 the user entered a snippet of HTML such as their name. When users view the list of names, the Razor template renders the names using `@Html.Raw()`, which writes the `<script>` tag directly to the document. The user's input has become part of the page's HTML structure. As soon as the page is loaded in a user's browser,

<sup>6</sup> For a detailed discussion of XSS attacks, see the “Cross Site Scripting (XSS)” article on the OWASP site: <https://owasp.org/www-community/attacks/xss/>.

the `<script>` tag executes, and the user is compromised. Once an attacker can execute arbitrary JavaScript on a user's browser, they can do pretty much anything.

The vulnerability here is due to rendering the user input in an unsafe way. If the data isn't encoded to make it safe before it's rendered, you could open your users to attack. *By default, Razor protects against XSS attacks* by HTML-encoding any data written using Tag Helpers, HTML Helpers, or the `@` syntax. So, generally, you should be safe, as you saw in chapter 7.

Using `@Html.Raw()` is where the danger lies—if the HTML you're rendering contains user input (even indirectly), you could have an XSS vulnerability. By rendering the user input with `@` instead, the content is encoded before it's written to the output, as shown in figure 18.6.

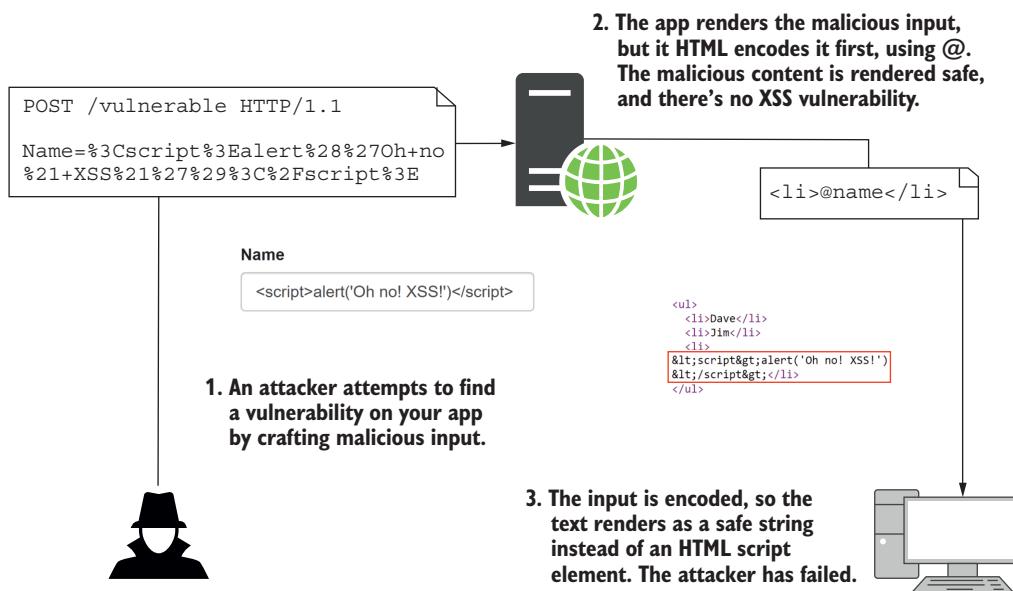


Figure 18.6 Protecting against XSS attacks by HTML-encoding user input using `@` in Razor templates. The `<script>` tag is encoded so that it is no longer rendered as HTML and can't be used to compromise your app.

This example demonstrates using HTML encoding to prevent elements being directly added to the HTML DOM, but it's not the only case you have to think about. If you're passing untrusted data to JavaScript, or using untrusted data in URL query values, you must make sure you encode the data correctly.

A common scenario is when you're using jQuery or JavaScript with Razor pages, and you want to pass a value from the server to the client. If you use the standard `@` symbol to render the data to the page, the output will be HTML-encoded. Unfortunately, if you HTML-encode a string and inject it directly into JavaScript, you probably won't get what you expect.

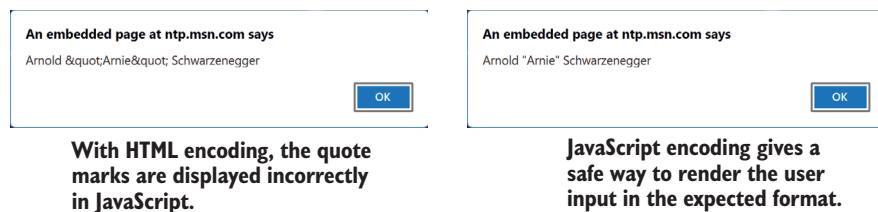
For example, if you have a variable in your Razor file called `name`, and you want to make it available in JavaScript, you might be tempted to use something like this:

```
<script>var name = '@name'</script>
```

If the `name` contains special characters, Razor will encode them using HTML encoding, which probably isn't what you want in this JavaScript context. For example, if `name` was Arnold "Arnie" Schwarzenegger, then rendering it as you did previously would give this:

```
<script>var name = 'Arnold &quot;Arnie&quot; Schwarzenegger';</script>
```

Note how the double quotation marks ("") have been HTML-encoded to `&quot;`. If you use this value in JavaScript directly, expecting it to be a "safe" encoded value, it's going to look wrong, as shown in figure 18.7.



**Figure 18.7** Comparison of alerts when using JavaScript encoding compared to HTML encoding

Instead, you should encode the variable using JavaScript encoding so that the double-quote character rendered as a safe Unicode character, `\u0022`. You can achieve this by injecting a `JavaScriptEncoder` into the view (as you saw in chapter 10) and calling `Encode()` on the `name` variable:

```
@inject System.Text.Encodings.Web.JavaScriptEncoder encoder;
<script>var name = '@encoder.Encode(name)'</script>
```

To avoid having to remember to use JavaScript encoding, I recommend you don't write values into JavaScript like this. Instead, write the value to an HTML element's attributes, and then read that into the JavaScript variable later. That avoids the need for the JavaScript encoder entirely.

#### Listing 18.4 Passing values to JavaScript by writing them to HTML attributes

Gets a reference to the HTML element

```
<div id="data" data-name="@name"></div>
<script>
  var ele = document.getElementById('data');
```

Write the value you want in JavaScript to a `data-*` attribute. This will HTML-encode the data.

```
var name = ele.getAttribute('data-name');  
/</script>
```

Reads the `data-*` attribute into  
JavaScript, which will convert it  
to JavaScript encoding

XSS attacks are still common, and it's easy to expose yourself to them whenever you allow users to input data. Validation of the incoming data can sometimes help, but it's often a tricky problem. For example, a naive name validator might require that you only use letters, which would prevent most attacks. Unfortunately, that doesn't account for users with hyphens or apostrophes in their name, let alone users with non-western names. People get (understandably) upset when you tell them their name is invalid, so be wary of this approach!

Whether or not you use strict validation, you should always encode the data when you render it to the page. Think carefully whenever you find yourself writing `@Html.Raw()`. Is there any way for a user to get malicious data into that field? If so, you'll need to find another way to display the data.

XSS vulnerabilities allow attackers to execute JavaScript on a user's browser. The next vulnerability we're going to consider lets them make requests to your API as though they're a different logged-in user, even when the user isn't using your app. Scared? I hope so!

### 18.3 Protecting from cross-site request forgery (CSRF) attacks

In this section you'll learn about cross-site request forgery attacks, how attackers can use them to impersonate a user on your site, and how to protect against them using anti-forgery tokens. Razor Pages protects you from these attacks by default, but you can disable these verifications, so it's important to understand the implications of doing so.

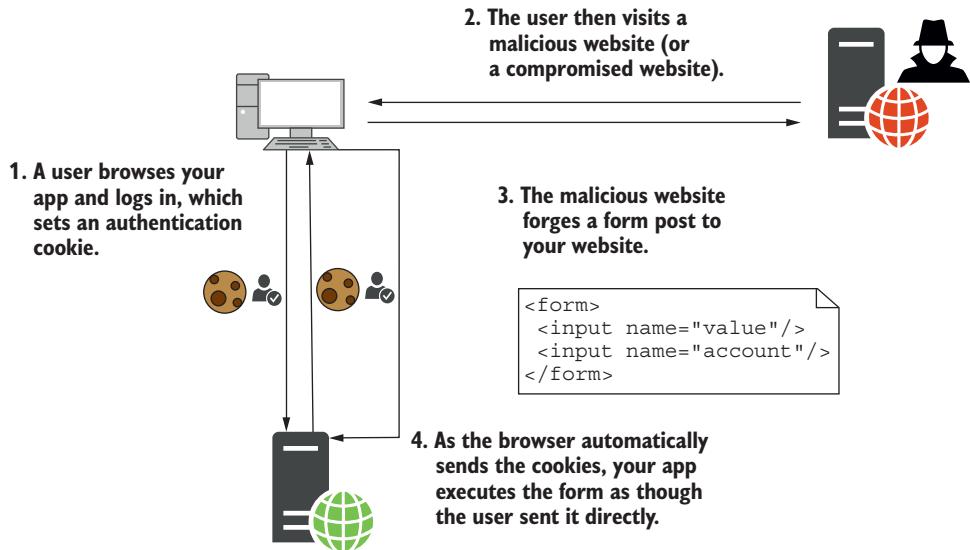
Cross-site request forgery (CSRF) attacks can be a problem for websites or APIs that use cookies for authentication. A CSRF attack involves a malicious website making an authenticated request to your API on behalf of the user, without the user initiating the request. In this section we'll explore how these attacks work and how you can mitigate them with anti-forgery tokens.

The canonical example of this attack is a bank transfer/withdrawal. Imagine you have a banking application that stores authentication tokens in a cookie, as is common (especially in traditional server-side rendered applications). Browsers automatically send the cookies associated with a domain with every request, so the app knows whether a user is authenticated.

Now imagine your application has a page that lets a user transfer funds from their account to another account using a POST request to the Balance Razor Page. You have to be logged in to access the form (you've protected the Razor Page with the `[Authorize]` attribute), but otherwise you just post a form that says how much you want to transfer, and where you want to transfer it.

Suppose a user visits your site, logs in, and performs a transaction. They then visit a second website that the attacker has control of. The attacker has embedded a form on

their website that performs a POST to your bank's website, identical to the transfer funds form on your banking website. This form does something malicious, such as transfer all the user's funds to the attacker, as shown in figure 18.8. Browsers automatically send the cookies for the application when the page does a full form post, and the banking app has no way of knowing that this is a malicious request. The unsuspecting user has given all their money to the attacker!



**Figure 18.8** A CSRF attack occurs when a logged-in user visits a malicious site. The malicious site crafts a form that matches one on your app and POSTS it to your app. The browser sends the authentication cookie automatically, so your app sees the request as a valid request from the user.

The vulnerability here revolves around the fact that browsers automatically send cookies when a page is requested (using a GET request) or a form is POSTed. There's no difference between a legitimate POST of the form in your banking app and the attacker's malicious POST. Unfortunately, this behavior is baked into the web; it's what allows you to navigate websites seamlessly after initially logging in.

A common solution to the attack is the *synchronizer token* pattern, which uses user-specific, unique, anti-forgery tokens to enforce a difference between a legitimate POST and a forged POST from an attacker.<sup>7</sup> One token is stored in a cookie and another is added to the form you wish to protect. Your app generates the tokens at runtime

<sup>7</sup> The “Cross-Site Request Forgery Prevention Cheat Sheet” article on the OWASP site gives a thorough discussion of the CSRF vulnerability, including the synchronizer token pattern: <http://mng.bz/5jRa>.

based on the current logged-in user, so there's no way for an attacker to create one for their forged form.

When the Balance Razor Page receives a form POST, it compares the value in the form with the value in the cookie. If either value is missing, or they don't match, the request is rejected. If an attacker creates a POST, the browser will post the cookie token as usual, but there won't be a token in the form itself, or the token won't be valid. The Razor Page will reject the request, protecting from the CSRF attack, as in figure 18.9.

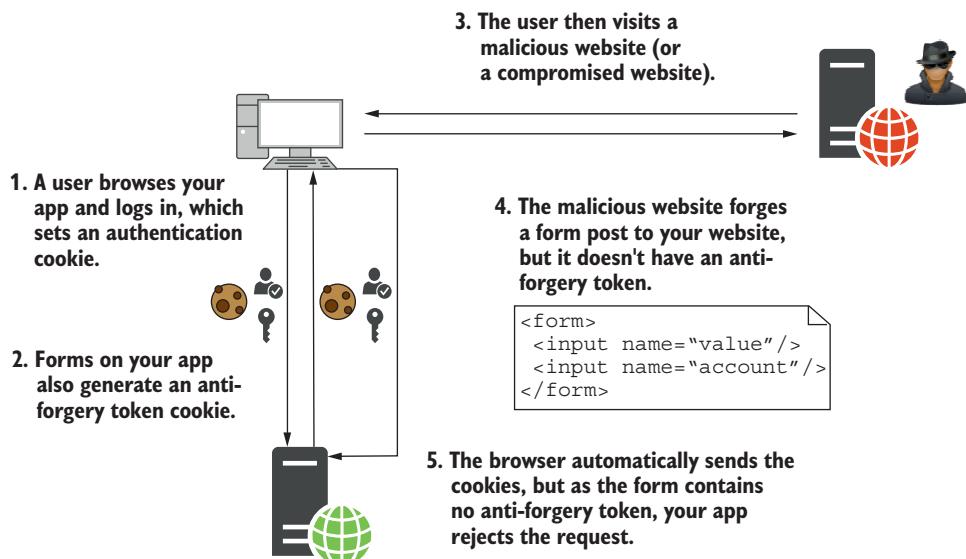


Figure 18.9 Protecting against a CSRF attack using anti-forgery tokens. The browser automatically forwards the cookie token, but the malicious site can't read it, and so can't include a token in the form. The app rejects the malicious request because the tokens don't match.

The good news is that Razor Pages automatically protects you against CSRF attacks. The Form Tag Helper automatically sets an anti-forgery token cookie and renders the token to a hidden field called `_RequestVerificationToken` for every `<form>` element in your app (unless you specifically disable them). For example, take this simple Razor template that posts back to the same Razor Page:

```
<form method="post">
  <label>Amount</label>
  <input type="number" name="amount" />
  <button type="submit">Withdraw funds</button>
</form>
```

When rendered to HTML, the anti-forgery token is stored in the hidden field and is posted back with a legitimate request:

```
<form method="post">
    <label>Amount</label>
    <input type="number" name="amount" />
    <button type="submit" >Withdraw funds</button>
    <input name="__RequestVerificationToken" type="hidden"
           value="CfDJ8Daz26qb0hBGsw7QCK"/>
</form>
```

ASP.NET Core automatically adds the anti-forgery tokens to every form, and Razor Pages automatically validates them. The framework ensures the anti-forgery tokens exist in both the cookie and the form data, ensures that they match, and will reject any requests where they don't.

If you're using MVC controllers with views instead of Razor Pages, ASP.NET Core still adds the anti-forgery tokens to every form. Unfortunately, it *doesn't* validate them for you. Instead, you have to decorate your controllers and actions with `[ValidateAntiForgeryToken]` attributes. This ensures that the anti-forgery tokens exist in both the cookie and the form data, checks that they match, and rejects any requests where they don't.

**WARNING** ASP.NET Core *doesn't* automatically validate anti-forgery tokens if you're using MVC controllers with Views. You must make sure you mark all vulnerable methods with `[ValidateAntiForgeryToken]` attributes instead, as described in the "Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core" documentation: <http://mng.bz/Xd6E>. Note that if you're using Web API controllers and are *not* using cookies for authentication, you are not vulnerable to CSRF attacks.

Generally, you only need to use anti-forgery tokens for POST, DELETE, and other dangerous request types that are used for modifying state; GET requests shouldn't be used for this purpose, so the framework doesn't require valid anti-forgery tokens to call them. Razor Pages validates anti-forgery tokens for dangerous verbs like POST and ignores safe verbs like GET. As long as you create your app following this pattern (and you should!), the framework will do the right thing to keep you safe.

If you need to explicitly ignore anti-forgery tokens on a Razor Page for some reason, you can disable the validation by applying the `[IgnoreAntiforgeryToken]` attribute to a Razor Page's `PageModel`. This bypasses the framework protections for those cases where you're doing something that you know is safe and doesn't need protecting, but in most cases it's better to play it safe and validate.

CSRF attacks can be a tricky thing to get your head around from a technical point of view, but for the most part everything *should* work without much effort on your part. Razor will add anti-forgery tokens to your forms, and the Razor Pages framework will take care of validation for you.

Where things get trickier is if you're making a lot of requests to an API using JavaScript, and you're posting JSON objects rather than form data. In these cases, you won't be able to send the verification token as part of a form (because you're sending JSON), so you'll need to add it as a header in the request instead.<sup>8</sup>

**TIP** If you're not using cookie authentication, and instead have an SPA that sends authentication tokens in a header, then good news—you don't have to worry about CSRF at all! Malicious sites can only send cookies, not headers, to your API, so they can't make authenticated requests.

### Generating unique tokens with the data protection APIs

The anti-forgery tokens used to prevent CSRF attacks rely on the ability of the framework to use strong symmetric encryption to encrypt and decrypt data. Encryption algorithms typically rely on one or more keys, which are used to initialize the encryption and to make the process reproducible. If you have the key, you can encrypt and decrypt data; without it, the data is secure.

In ASP.NET Core, encryption is handled by the data protection APIs. They're used to create the anti-forgery tokens, to encrypt authentication cookies, and to generate secure tokens in general. Crucially, they also control the management of the key files that are used for encryption.

A key file is a small XML file that contains the random key value used for encryption in ASP.NET Core apps. It's critical that it's stored securely—if an attacker got hold of it, they could impersonate any user of your app and generally do bad things!

The data protection system stores the keys in a safe location, depending on how and where you host your app. For example,

- Azure Web App—In a special synced folder, shared between regions
- IIS without user profile—Encrypted in the registry
- Account with user profile—In %LOCALAPPDATA%\ASP.NET\DataProtection-Keys on Windows, or ~/aspnet/DataProtection-Keys on Linux or macOS
- All other cases—In memory; when the app restarts, the keys will be lost

So why do you care? In order for your app to be able to read your users' authentication cookies, it must decrypt them using the same key that was used to encrypt them. If you're running in a web-farm scenario, then, by default, each server will have its own key and won't be able to read cookies encrypted by other servers.

<sup>8</sup> Exactly how you do this varies depending on the JavaScript framework you're using. Microsoft's documentation ("Prevent Cross-Site Request Forgery (XSRF/CSRF) attacks in ASP.NET Core") contains examples using JQuery and AngularJS, but you should be able to extend this to your JavaScript framework of choice: <http://mng.bz/54Sl>.

To get around this, you must configure your app to store its data protection keys in a central location. This could be a shared folder on a hard drive, a Redis instance, or an Azure blob storage instance, for example.

Microsoft's documentation on the data protection APIs is extremely detailed, but it can be overwhelming. I recommend reading the section on configuring data protection, ("Configure ASP.NET Core Data Protection," <http://mng.bz/d40i>) and configuring a key storage provider for use in a web-farm scenario ("Key storage providers in ASP.NET Core," <http://mng.bz/5pW6>).

It's worth clarifying that the CSRF vulnerability discussed in this section requires that a malicious site does a full form POST to your app. The malicious site can't make the request to your API using *client-side only* JavaScript, as browsers will block JavaScript requests to your API that are from a different origin.

This is a safety feature, but it can often cause you problems. If you're building a client-side SPA, or even if you have a little JavaScript on an otherwise server-side rendered app, you may find you need to make such *cross-origin* requests. In the next section I'll describe a common scenario you're likely to run into and show how you can modify your apps to work around it.

## 18.4 Calling your web APIs from other domains using CORS

In this section you'll learn about cross-origin resource sharing (CORS), a protocol to allow JavaScript to make requests from one domain to another. CORS is a frequent area of confusion for many developers, so this section describes why it's necessary and how CORS headers work. You'll then learn how to add CORS to both your whole application and specific Web API actions, and how to configure multiple CORS policies for your application.

As you've already seen, CSRF attacks can be powerful, but they would be even more dangerous if it weren't for browsers implementing the *same-origin* policy. This policy blocks apps from using JavaScript to call a web API at a different location unless the web API explicitly allows it.

**DEFINITION** *Origins* are deemed the same if they match the scheme (HTTP or HTTPS), domain (example.com), *and* port (80 by default for HTTP, and 443 for HTTPS). If an app attempts to access a resource using JavaScript and the origins aren't identical, the browser blocks the request.

The same-origin policy is strict—the origins of the two URLs must be identical for the request to be allowed. For example, the following origins are the same:

- <http://example.com/home>
- <http://example.com/site.css>

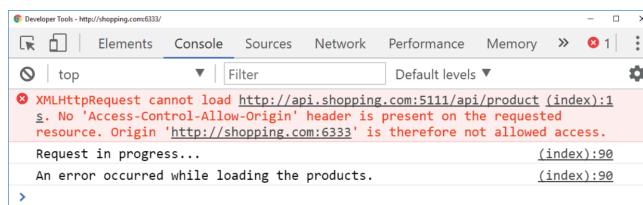
The paths are different for these two URLs (/home and /site.css), but the scheme, domain, and port (80) are identical. So if you were on the home page of your app, you could request the /site.css file using JavaScript without any issues.

In contrast, the origins of the following sites are all different, so you couldn't request any of these URLs using JavaScript from the http://example.com origin:

- `https://example.com`—Different scheme (https)
- `http://www.example.com`—Different domain (includes a subdomain)
- `http://example.com:5000`—Different port (default HTTP port is 80)

For simple apps, where you have a single web app handling all of your functionality, this limitation might not be a problem, but it's extremely common for an app to make requests to another domain.

For example, you might have an e-commerce site hosted at `http://shopping.com`, and you're attempting to load data from `http://api.shopping.com` to display details about the products available for sale. With this configuration, you'll fall foul of the same-origin policy. Any attempt to make a request using JavaScript to the API domain will fail, with an error similar to figure 18.10.



**The browser won't allow cross-origin requests by default and will block your app from accessing the response.**

Figure 18.10 The console log for a failed cross-origin request. Chrome has blocked a cross-origin request from the app `http://shopping.com:6333` to the API at `http://api.shopping.com:5111`.

The need to make cross-origin requests from JavaScript is increasingly common with the rise of client-side SPAs and the move away from monolithic apps. Luckily, there's a web standard that lets you work around this in a safe way; this standard is called cross-origin resource sharing (CORS). You can use CORS to control which apps can call your API, so you can enable scenarios like the one just described.

#### 18.4.1 Understanding CORS and how it works

CORS is a web standard that allows your Web API to make statements about who can make cross-origin requests to it. For example, you could make statements such as these:

- Allow cross-origin requests from `http://shopping.com` and `https://app.shopping.com`.
- Only allow GET cross-origin requests.

- Allow returning the `Server` header in responses to cross-origin requests.
- Allow credentials (such as authentication cookies or authorization headers) to be sent with cross-origin requests.

You can combine these rules into a *policy* and apply different policies to different endpoints of your API. You could apply a policy to your entire application, or a different policy to every API action.

CORS works using HTTP headers. When your Web API application receives a request, it sets special headers on the response to indicate whether cross-origin requests are allowed, which origins they're allowed from, and which HTTP verbs and headers the request can use—pretty much everything about the request.

In some cases, before sending a real request to your API, the browser sends a *preflight* request. This is a request sent using the `OPTIONS` verb, which the browser uses to check whether it's allowed to make the real request. If the API sends back the correct headers, the browser will send the true cross-origin request, as shown in figure 18.11.

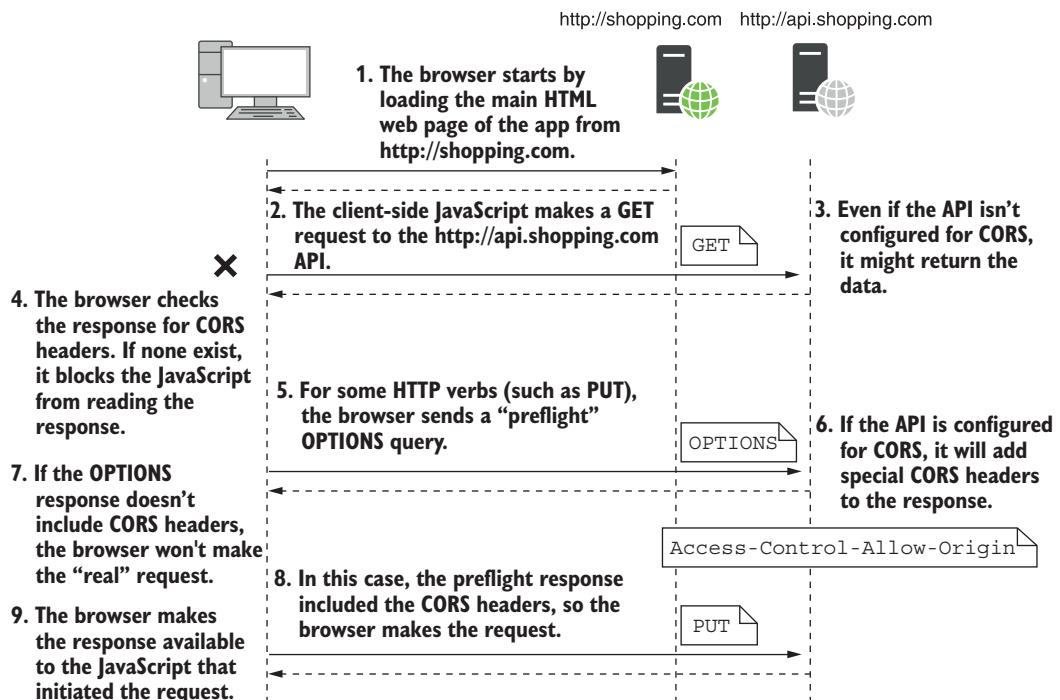


Figure 18.11 Two cross-origin requests. The response to the first request doesn't contain any CORS headers, so the browser blocks the app from reading it. The second request requires a preflight `OPTIONS` request, to check if CORS is enabled. As the response contains CORS headers, the real request can be made and the response provided to the JavaScript app.

**TIP** For a more detailed discussion of CORS, see *CORS in Action* by Monsur Hossain (Manning, 2014), available at <http://mng.bz/aD41>.

The CORS specification, like many technical documents, is pretty complicated, with a variety of headers and processes to contend with.<sup>9</sup> Thankfully, ASP.NET Core handles the details of the specification for you, so your main concern is working out exactly *who* needs to access your API, and under what circumstances.

### 18.4.2 Adding a global CORS policy to your whole app

Typically, you shouldn't set up CORS for your APIs until you need it. Browsers block cross-origin communication for a reason—it closes an avenue of attack—they're not being awkward. Wait until you have an API on a different domain than an app that needs to access it.

Adding CORS support to your application requires four things:

- Add the CORS services to your app.
- Configure at least one CORS policy.
- Add the CORS middleware to your middleware pipeline.
- Either set a default CORS policy for your entire app or decorate your Web API actions with the `[EnableCors]` attribute to selectively enable CORS for specific endpoints.

Adding the CORS services to your application involves calling `AddCors()` in your `Startup.ConfigureServices` method:

```
services.AddCors();
```

The bulk of your effort in configuring CORS will go into policy configuration. A CORS policy controls how your application will respond to cross-origin requests. It defines which origins are allowed, which headers to return, which HTTP methods to allow, and so on. You normally define your policies inline when you add the CORS services to your application.

For example, consider the previous e-commerce site example. You want your API that is hosted at <http://api.shopping.com> to be available from the main app via client-side JavaScript, hosted at <http://shopping.com>. You therefore need to configure the *API* to allow cross-origin requests.

**NOTE** Remember, it's the main app that will get errors when attempting to make cross-origin requests, but it's the *API* you're accessing that you need to add CORS to, *not* the app making the requests.

The following listing shows how to configure a policy called "AllowShoppingApp" to enable cross-origin requests from <http://shopping.com> to the API. Additionally, we

---

<sup>9</sup> If that's the sort of thing that floats your boat, you can read the spec here: <https://fetch.spec.whatwg.org/#http-cors-protocol>.

explicitly allow any HTTP verb type; without this call, only simple methods (GET, HEAD, and POST) are allowed. The policies are built up using the familiar fluent builder style you've seen throughout this book.

#### Listing 18.5 Configuring a CORS policy to allow requests from a specific origin

The `AddCors` method exposes an `Action<CorsOptions>` overload.

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddCors(options => {
        options.AddPolicy("AllowShoppingApp", policy =>
            policy.WithOrigins("http://shopping.com")
                .AllowAnyMethod());
    });
    // other service configuration
}
```

Allows all HTTP verbs to call the API

Every policy has a unique name.

The `WithOrigins` method specifies which origins are allowed. Note that the URL has no trailing /.

**WARNING** When listing origins in `WithOrigins()`, ensure that they don't have a trailing "/"; otherwise the origin will never match and your cross-origin requests will fail.

Once you've defined a CORS policy, you can apply it to your application. In the following listing, you apply the "AllowShoppingApp" policy to the whole application using `CorsMiddleware` by calling `UseCors()` in the `Configure` method of `Startup.cs`.

#### Listing 18.6 Adding the CORS middleware and configuring a default CORS policy

```
public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    app.UseRouting();           ← The CORS middleware must come
                                after the call to UseRouting().

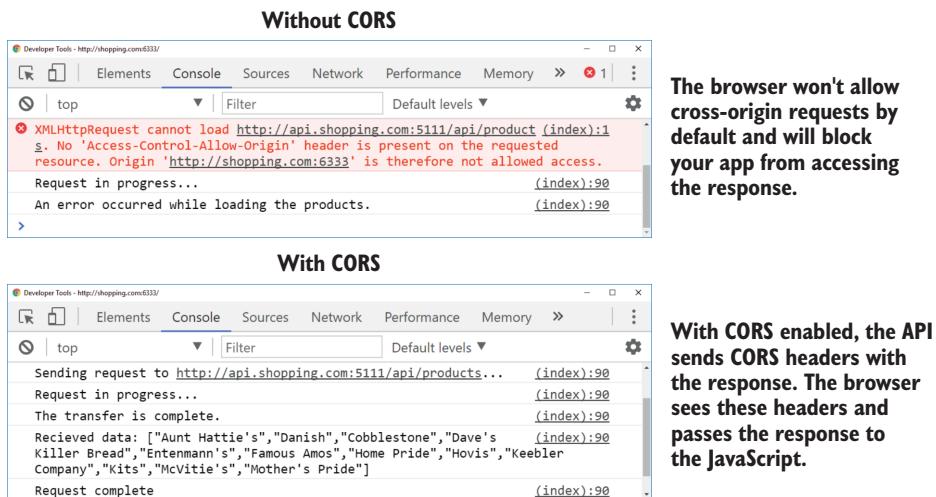
    app.UseCors("AllowShoppingApp");   ← Adds the CORS middleware
    app.UseAuthentication();          and uses AllowShoppingApp
    app.UseAuthorization();          as the default policy

    app.UseEndpoints(endpoints =>   ← Place the CORS
    {                                middleware before the
        endpoints.MapControllers();  endpoint middleware.
    });
}
```

**NOTE** As with all middleware, the order of the CORS middleware is important. You must place the call to `UseCors()` *after* `UseRouting()` and before `UseEndpoints()`. The CORS middleware needs to intercept cross-origin requests to your Web API actions so it can generate the correct responses to preflight requests and add the necessary headers. It's typical to place the CORS middleware before the call to `UseAuthentication()`.

With the CORS middleware in place for the API, the shopping app can now make cross-origin requests. You can call the API from the <http://shopping.com> site and the

browser lets the CORS request through, as shown in figure 18.12. If you make the same request from a domain other than <http://shopping.com>, the request continues to be blocked.



**Figure 18.12** With CORS enabled, as in the lower image, cross-origin requests can be made and the browser will make the response available to the JavaScript. Compare this to the upper image, in which the request was blocked.

Applying a CORS policy globally to your application in this way may be overkill. If there's only a subset of actions in your API that need to be accessed from other origins, then it's prudent to only enable CORS for those specific actions. This can be achieved with the `[EnableCors]` attribute.

#### 18.4.3 Adding CORS to specific Web API actions with `EnableCorsAttribute`

Browsers block cross-origin requests by default for good reason—they have the potential to be abused by malicious or compromised sites. Enabling CORS for your entire app may not be worth the risk if you know that only a subset of actions will ever need to be accessed cross-origin.

If that's the case, it's best to only enable a CORS policy for those specific actions. ASP.NET Core provides the `[EnableCors]` attribute, which lets you select a policy to apply to a given controller or action method. This approach lets you apply different CORS policies to different action methods. For example, you could allow GET requests access to your entire API from the <http://shopping.com> domain, but only allow other HTTP verbs for a specific controller, while allowing anyone to access your product list action method.

You define these policies in `ConfigureServices` using `AddPolicy()` and giving the policy a name, as you saw in listing 18.5. However, instead of calling `UseCors("AllowShoppingApp")` as you saw in listing 18.6, you add the middleware without a default policy by calling `UseCors()` only.

To apply a policy to a controller or an action method, apply the `[EnableCors]` attribute, as shown in the following listing. An `[EnableCors]` attribute on an action takes precedence over an attribute on a controller, or you can use the `[DisableCors]` attribute to disable cross-origin access to the method entirely.

#### Listing 18.7 Applying the `EnableCors` attribute to a controller and action

The `AllowAnyOrigin` policy is “closer” to the action, so it takes precedence.

```
[EnableCors("AllowShoppingApp")]
public class ProductController: Controller
{
    [EnableCors("AllowAnyOrigin")]
    public IActionResult GeteProducts() { /* Method */ }

    public IActionResult GeteProductPrice(int id) { /* Method */ }

    [DisableCors]
    public IActionResult DeleteProduct(int id) { /* Method */ }
}
```

The `AllowShoppingApp` policy (from the controller) will be applied.

Applies the `AllowShoppingApp` CORS policy to every action method

The `DisableCors` attribute disables CORS for the action method completely.

If you want to apply a CORS policy to most of your actions but want to use a different policy or disable CORS entirely for some actions, you can pass a default policy when configuring the middleware using `UseCors("AllowShoppingApp")`, for example. Actions decorated with `[EnableCors("OtherPolicy")]` will apply `OtherPolicy` preferentially, and actions decorated with `[DisableCors]` will not have CORS enabled at all.

Whether you choose to use a single, default CORS policy or multiple policies, you need to configure the CORS policies for your application in `ConfigureServices`. Many different options are available when configuring CORS. In the next section I'll provide an overview of the possibilities.

#### 18.4.4 Configuring CORS policies

Browsers implement the cross-origin policy for security reasons, so you should carefully consider the implications of relaxing any of the restrictions they impose. Even if you enable cross-origin requests, you can still control what data cross-origin requests can send, and what your API will return. For example, you can configure

- The origins that may make a cross-origin request to your API
- The HTTP verbs (such as GET, POST, and DELETE) that can be used
- The headers the browser can send

- The headers that the browser can read from your app's response
- Whether the browser will send authentication credentials with the request

You define all of these options when creating a CORS policy in your call to `AddCors()` using the `CorsPolicyBuilder`, as you saw in listing 18.5. A policy can set all or none of these options, so you can customize the results to your heart's content. Table 18.1 shows some of the options available, and their effects.

**Table 18.1** The methods available for configuring a CORS policy, and their effect on the policy

CorsPolicyBuilder method example	Result
<code>WithOrigins("http://shopping.com")</code>	Allows cross-origin requests from <code>http://shopping.com</code> .
<code>AllowAnyOrigin()</code>	Allows cross-origin requests from any origin. This means <i>any</i> website can make JavaScript requests to your API.
<code>WithMethods() / AllowAnyMethod()</code>	Sets the allowed methods (such as <code>GET</code> , <code>POST</code> , and <code>DELETE</code> ) that can be made to your API.
<code>WithHeaders() / AllowAnyHeader()</code>	Sets the headers that the browser may send to your API. If you restrict the headers, you must include at least <code>"Accept"</code> , <code>"Content-Type"</code> , and <code>"Origin"</code> to allow valid requests.
<code>WithExposedHeaders()</code>	Allows your API to send extra headers to the browser. By default, only the <code>Cache-Control</code> , <code>Content-Language</code> , <code>Content-Type</code> , <code>Expires</code> , <code>Last-Modified</code> , and <code>Pragma</code> headers are sent in the response.
<code>AllowCredentials()</code>	By default, the browser won't send authentication details with cross-origin requests unless you explicitly allow it. You must also enable sending credentials client-side in JavaScript when making the request.

One of the first issues in setting up CORS is realizing you have a cross-origin problem at all. Several times I've been stumped trying to figure out why a request won't work, until I realize the request is going cross-domain, or from HTTP to HTTPS, for example.

Whenever possible, I recommend avoiding cross-origin requests completely. You can end up with subtle differences in the way browsers handle them, which can cause more headaches. In particular, avoid HTTP to HTTPS cross-domain issues by running all of your applications behind HTTPS. As discussed in section 18.1, that's a best practice anyway, and it'll help avoid a whole class of CORS headaches.

Once I've established I definitely need a CORS policy, I typically start with the `WithOrigins()` method. I then expand or restrict the policy further, as need be, to

provide cross-origin lockdown of my API, while still allowing the required functionality. CORS can be tricky to work around, but remember, the restrictions are there for your safety.<sup>10</sup>

Cross-origin requests are only one of many potential avenues attackers could use to compromise your app. Many of these are trivial to defend against, but you need to be aware of them, and know how to mitigate them. In the next section we'll look at common threats and how to avoid them.

## 18.5 Exploring other attack vectors

So far in this chapter, I've described two potential ways attackers can compromise your apps—XSS and CSRF attacks—and how to prevent them. Both of these vulnerabilities regularly appear on the OWASP top ten list of most critical web app risks,<sup>11</sup> so it's important to be aware of them and to avoid introducing them into your apps. In this section I'll provide an overview of some of the other most common vulnerabilities and how to avoid them in your apps.

### 18.5.1 Detecting and avoiding open redirect attacks

A common OWASP vulnerability is due to *open redirect* attacks. An open redirect attack is where a user clicks a link to an otherwise safe app and ends up being redirected to a malicious website, such as one that serves malware. The safe app contains no direct links to the malicious website, so how does this happen?

Open redirect attacks occur where the next page is passed as a parameter to an action method. The most common example is when you're logging in to an app. Typically, apps remember the page a user is on before redirecting them to a login page by passing the current page as a `returnUrl` query string parameter. After the user logs in, the app redirects the user to the `returnUrl` to carry on where they left off.

Imagine a user is browsing an e-commerce site. They click Buy on a product and are redirected to the login page. The product page they were on is passed as the `returnUrl`, so after they log in, they're redirected to the product page instead of being dumped back to the home screen.

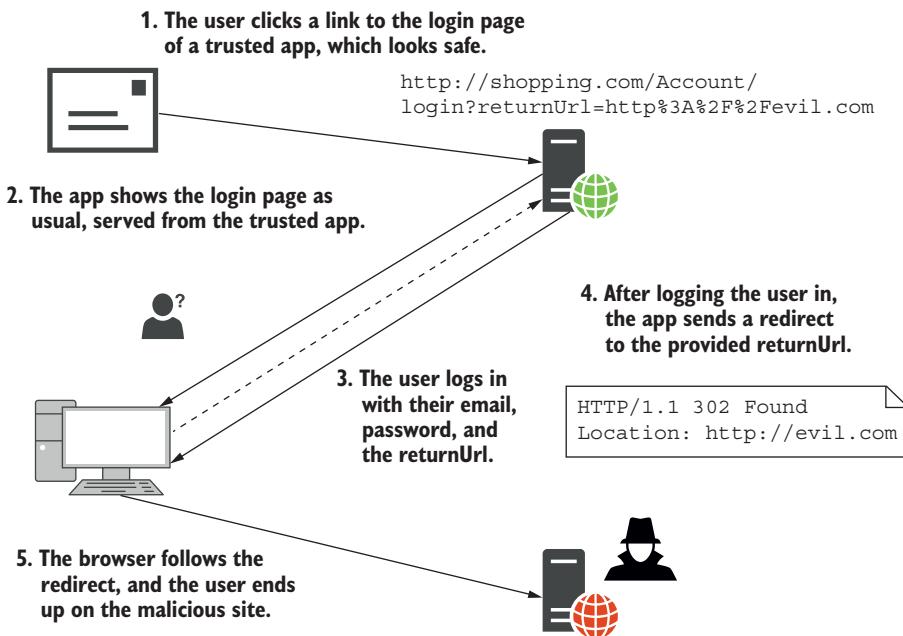
An open redirect attack takes advantage of this common pattern, as shown in figure 18.13. A malicious attacker creates a login URL where the `returnUrl` is set to the website they want to send the user to and convinces the user to click the link to your web app. After the user logs in, a vulnerable app will then redirect the user to the malicious site.

The simple solution to this attack is to always validate that the `returnUrl` is a local URL that belongs to your app *before* redirecting users to it. The default Identity UI

---

<sup>10</sup> You can find the OWASP top ten list here: <https://owasp.org/www-project-top-ten/>.

<sup>11</sup> OWASP publishes the list online, with descriptions of each attack and how to prevent those attacks. There's a cheat sheet for staying safe here: <https://cheatsheetseries.owasp.org/>.



**Figure 18.13** An open redirect makes use of the common return URL pattern. This is typically used for login pages but may be used in other areas of your app too. If your app doesn't verify that the URL is safe before redirecting the user, it could redirect users to malicious sites.

does this already, so you shouldn't have to worry about the login page if you're using Identity, as described in chapter 14.

If you have redirects in other parts of your app, ASP.NET Core provides a couple of helper methods for staying safe, the most useful of which is `Url.IsLocalUrl()`. The following listing shows how you could verify that a provided return URL is safe, and if it isn't, redirect to the app's homepage. You can also use the `LocalRedirect()` helper method on the `ControllerBase` and `Razor Page PageModel` classes, which throws an exception if the provided URL isn't local.

#### Listing 18.8 Detecting open redirect attacks by checking for local return URLs

```
[HttpPost]
public async Task<IActionResult> Login(
    LoginViewModel model, string returnUrl = null)
{
    // Verify password, and sign user in

    if (Url.IsLocalUrl(returnUrl))
    {
        return Redirect(returnUrl);
    }
}
```

The return URL is provided as an argument to the action method.

Returns true if the return URL starts with / or ~/

The URL is local, so it's safe to redirect to it.

```

    else
    {
        return RedirectToAction("Index", "Home");
    }
}

```

The URL was not local and could be an open redirect attack, so redirect to the homepage for safety.

This simple pattern protects against open redirect attacks that could otherwise expose your users to malicious content. Whenever you're redirecting to a URL that comes from a query string or other user input, you should use this pattern.

Open redirect attacks present a risk to your *users* rather than to your app directly. The next vulnerability represents a critical vulnerability in your app itself.

### 18.5.2 Avoiding SQL injection attacks with EF Core and parameterization

SQL injection attacks represent one of the most dangerous threats to your application. Attackers craft simple malicious input, which they send to your application as traditional form-based input or by customizing URLs and query strings to execute arbitrary code against your database. An SQL injection vulnerability could expose your entire database to attackers, so it's critical that you spot and remove any such vulnerabilities in your apps.

Hopefully I've scared you a little with that introduction, so now for the good news—if you're using EF Core (or pretty much any other ORM) in a standard way, you should be safe. EF Core has built-in protections against SQL injection, so as long as you're not doing anything funky, you should be fine.

SQL injection vulnerabilities occur when you build SQL statements yourself and include dynamic input that an attacker provides, even indirectly. EF Core provides the ability to create raw SQL queries using the `FromSqlRaw()` method, so you must be careful when using this method.

Imagine your recipe app has a search form that lets you search for a recipe by name. If you write the query using LINQ extension methods (as discussed in chapter 12), then you would have no risk of SQL injection attacks. However, if you decide to write your SQL query by hand, you open yourself up to such a vulnerability.

**Listing 18.9 An SQL injection vulnerability in EF Core due to string concatenation**

```

public IList<User> FindRecipe(string search)
{
    return _context.Recipes
        .FromSqlRaw("SELECT * FROM Recipes" +
                    "WHERE Name = '" + search + "'")
        .ToList();
}

```

The current EF Core `DbContext` is held in the `_context` field.

The search parameter comes from user input, so it's unsafe.

This introduces the vulnerability—including unsafe content directly in an SQL string.

You can write queries by hand using the `FromSqlRaw` extension method.

In this listing, the user input held in `search` is included *directly* in the SQL query. By crafting malicious input, users can potentially perform any operation on your database. Imagine an attacker searches your website using the text

```
'; DROP TABLE Recipes; --
```

Your app assigns this to the `search` parameter, and the SQL query executed against your database becomes

```
SELECT * FROM Recipes WHERE Name = ''; DROP TABLE Recipes; --'
```

By simply entering text into the search form of your app, the attacker has deleted the entire `Recipes` table from your app! That's catastrophic, but an SQL injection vulnerability provides more or less unfettered access to your database. Even if you've set up database permissions correctly to prevent this sort of destructive action, attackers will likely be able to read all the data from your database, including your users' details.

The simple way to avoid this happening is to avoid creating SQL queries by hand like this. If you do need to write your own SQL queries, don't use string concatenation, as in listing 18.9. Instead, use parameterized queries, in which the (potentially unsafe) input data is separate from the query itself, as shown here.

#### Listing 18.10 Avoiding SQL injection by using parameterization

```
public IList<User> FindRecipe(string search)
{
    return _context.Recipes
        .FromSqlRaw("SELECT * FROM Recipes WHERE Name = '{0}'",
                    search)
        .ToList();
}
```

**The SQL query uses a placeholder {0} for the parameter.**

**The dangerous input is passed as a parameter, separate from the query.**

Parameterized queries are not vulnerable to SQL injection attacks, so the attack presented earlier won't work. If you use EF Core (or other ORMs) to access data using standard LINQ queries, you won't be vulnerable to injection attacks. EF Core will automatically create all SQL queries using parameterized queries to protect you.

**NOTE** I've only talked about SQL injection attacks in terms of a relational database, but this vulnerability can appear in NoSQL and document databases too. Always use parameterized queries (or the equivalent), and don't craft queries by concatenating strings with user input.

Injection attacks have been the number one vulnerability on the web for over a decade, so it's crucial that you're aware of them and how they arise. Whenever you need to write raw SQL queries, make sure you always use parameterized queries.

The next vulnerability is also related to attackers accessing data they shouldn't be able to. It's a little subtler than a direct injection attack but is trivial to perform—the only skill the attacker needs is the ability to count.

### 18.5.3 Preventing insecure direct object references

*Insecure direct object reference* is a bit of a mouthful, but it means users accessing things they shouldn't by noticing patterns in URLs. Let's revisit our old friend the recipe app. As a reminder, the app shows you a list of recipes. You can view any of them, but you can only edit recipes you created yourself. When you view someone else's recipe, there's no Edit button visible.

For example, a user clicks the Edit button on one of their recipes and notices the URL is `/Recipes/Edit/120`. That "120" is a dead giveaway as the underlying database ID of the entity you're editing. A simple attack would be to change that ID to gain access to a *different* entity, one that you wouldn't normally have access to. The user could try entering `/Recipes/Edit/121`. If that lets them edit or view a recipe that they shouldn't be able to, you have an insecure direct object reference vulnerability.

The solution to this problem is simple—you should have resource-based authentication and authorization in your action methods. If a user attempts to access an entity they're not allowed to access, they should get a permission-denied error. They shouldn't be able to bypass your authorization by typing a URL directly into the search bar of their browser.

In ASP.NET Core apps, this vulnerability typically arises when you attempt to restrict users by hiding elements from your UI, such as by hiding the Edit button. Instead, you should use resource-based authorization, as discussed in chapter 15.

**WARNING** You must always use resource-based authorization to restrict which entities a user can access. Hiding UI elements provides an improved user experience, but it isn't a security measure.

You can sidestep this vulnerability somewhat by avoiding integer IDs for your entities in the URLs; for example, using a pseudorandom GUID (for instance, C2E296BA-7EA8-4195-9CA7-C323304CCD12) instead. This makes the process of guessing other entities harder, as you can't just add one to an existing number, but it's only masking the problem rather than fixing it. Nevertheless, using GUIDs can be useful when you want to have publicly accessible pages (that don't require authentication), but you don't want their IDs to be easily discoverable.

The final section in this chapter doesn't deal with a single vulnerability. Instead, I'll discuss a separate, but related, issue: protecting your users' data.

### 18.5.4 Protecting your users' passwords and data

For many apps, the most sensitive data you'll be storing is the personal data of your users. This could include emails, passwords, address details, or payment information. You should be careful when storing any of this data. As well as presenting an inviting target for attackers, you may have legal obligations for how you handle it, such as data protection laws and PCI compliance requirements.

The easiest way to protect yourself is to not store data that you don't need. If you don't *need* your user's address, don't ask for it. That way, you can't lose it! Similarly, if

you use a third-party identity service to store user details, as described in chapter 14, you won't have to work as hard to protect your users' personal information.

If you store user details in your own app, or build your own identity provider, then you need to make sure to follow best practices when handling user information. The new project templates that use ASP.NET Core Identity follow most of these practices by default, so I highly recommend you start from one of these. You need to consider many different aspects—too many to go into detail here<sup>12</sup>—but they include the following:

- Never store user passwords anywhere directly. You should only store cryptographic hashes, computed using an expensive hashing algorithm, such as BCrypt or PBKDF2.
- Don't store more data than you need. You should never store credit card details.
- Allow users to use two-factor authentication (2FA) to sign in to your site.
- Prevent users from using passwords that are known to be weak or compromised.
- Mark authentication cookies as "http" (so they can't be read using JavaScript) and "secure" so they'll only be sent over an HTTPS connection, never over HTTP.
- Don't expose whether a user is already registered with your app or not. Leaking this information can expose you to enumeration attacks.<sup>13</sup>

These are all guidelines, but they represent the minimum you should be doing to protect your users. The most important thing is to be aware of potential security issues as you're building your app. Trying to bolt on security at the end is always harder than thinking about it from the start, so it's best to think about it earlier rather than later.

This chapter has been a whistle-stop tour of things to look out for. We've touched on most of the big names in security vulnerabilities, but I strongly encourage you to check out the other resources mentioned in this chapter. They provide a more exhaustive list of things to consider, complementing the defenses mentioned in this chapter. On top of that, don't forget about input validation and mass assignment/over-posting, as discussed in chapter 6. ASP.NET Core includes basic protections against some of the most common attacks, but you can still shoot yourself in the foot. Make sure it's not your app making headlines for being breached!

## Summary

- HTTPS is used to encrypt your app's data as it travels from the server to the browser and back. This prevents third parties from seeing or modifying it.
- HTTPS is virtually mandatory for production apps, as modern browsers like Chrome and Firefox mark non-HTTPS apps as explicitly "not secure."

---

<sup>12</sup> The NIST (National Institute of Standards and Technology) recently released their Digital Identity Guidelines on how to handle user details: <http://mng.bz/6gRA>.

<sup>13</sup> You can learn more about website enumeration in this video tutorial from Troy Hunt: <http://mng.bz/PAAA>.

- In production, you can avoid handling the TLS in your app by using SSL/TLS offloading. This is where a reverse proxy uses HTTPS to talk to the browser, but the traffic is unencrypted between your app and the reverse proxy. The reverse proxy could be on the same or a different server, such as IIS or NGINX, or it could be a third-party service, such as Cloudflare.
- You can use the ASP.NET Core developer certificate or the IIS express developer certificate to enable HTTPS during development. This can't be used for production, but it's sufficient for testing locally. You must run `dotnet dev-certs https --trust` when you first install the .NET SDK to trust the certificate.
- You can configure an HTTPS certificate for Kestrel in production using the `Kestrel:Certificates:Default` configuration section. This does not require any changes to your application—Kestrel will automatically load the certificate when your app starts and use it to serve HTTPS requests.
- You can use the `HstsMiddleware` to set HTTP Strict Transport Security (HSTS) headers for your application, to ensure the browser sends HTTPS requests to your app instead of HTTP requests. This can only be enforced once an HTTPS request is made to your app, so it's best used in conjunction with HTTP to HTTPS redirection.
- You can enforce HTTPS for your whole app using the `HttpsRedirectionMiddleware`. This will redirect HTTP requests to HTTPS endpoints.
- Cross-site scripting (XSS) attacks involve malicious users injecting content into your app, typically to run malicious JavaScript when users browse your app. You can avoid XSS injection attacks by always encoding unsafe input before writing it to a page. Razor Pages do this automatically unless you use the `@Html.Raw()` method, so use it sparingly and carefully.
- Cross-site request forgery (CSRF) attacks are a problem for apps that use cookie-based authentication, such as ASP.NET Core Identity. It relies on the fact that browsers automatically send cookies to a website. A malicious website could create a form that POSTs to your site, and the browser will send the authentication cookie with the request. This allows malicious websites to send requests as though they're the logged-in user.
- You can mitigate CSRF attacks using anti-forgery tokens. These involve writing a hidden field in every form that contains a random string based on the current user. A similar token is stored in a cookie. A legitimate request will have both parts, but a forged request from a malicious website will only have the cookie half; they cannot recreate the hidden field in the form. By validating these tokens, your API can reject forged requests.
- The Razor Pages framework automatically adds anti-forgery tokens to any forms you create using Razor and validates the tokens for inbound requests. You can disable the validation check if necessary, using the `[IgnoreAntiForgeryToken]` attribute.

- Browsers won't allow websites to make JavaScript AJAX requests from one app to others at different origins. To match the origin, the app must have the same scheme, domain, and port. If you wish to make cross-origin requests like this, you must enable cross-origin resource sharing (CORS) in your API.
- CORS uses HTTP headers to communicate with browsers and defines which origins can call your API. In ASP.NET Core, you can define multiple policies, which can be applied either globally to your whole app, or to specific controllers and actions.
- You can add the CORS middleware by calling `UseCors()` in `Startup.Configure` and optionally providing the name of the default CORS policy to apply. You can also apply CORS to a Web API action or controller by adding the `[EnableCors]` attribute and providing the name of the policy to apply.
- Open redirect attacks use the common `returnURL` mechanism after logging in to redirect users to malicious websites. You can prevent this attack by ensuring you only redirect to local URLs—URLs that belong to your app.
- Insecure direct object references are a common problem where you expose the ID of database entities in the URL. You should always verify that users have permission to access or change the requested resource by using resource-based authorization in your action methods.
- SQL injection attacks are a common attack vector when you build SQL requests manually. Always use parameterized queries when building requests, or instead use a framework like EF Core, which isn't vulnerable to SQL injection.
- The most sensitive data in your app is often the data of your users. Mitigate this risk by only storing data that you need. Ensure you only store passwords as a hash, protect against weak or compromised passwords, and provide the option for 2FA. ASP.NET Core Identity provides all of this out of the box, so it's a great choice if you need to create an identity provider.

# 10

## *Building custom components*

### **This chapter covers**

- Building custom middleware
- Creating simple endpoints that generate a response using middleware
- Using configuration values to set up other configuration providers
- Replacing the built-in DI container with a third-party container

When you’re building apps with ASP.NET Core, most of your creativity and specialization goes into the services and models that make up your business logic and the Razor Pages and controllers that expose them through views or APIs. Eventually, however, you’re likely to find that you can’t quite achieve a desired feature using the components that come out of the box. At that point, you may need to build a custom component.

This chapter shows how you can create some ASP.NET Core components that you’re likely to need as your app grows. You probably won’t need to use all of them, but each solves a specific problem you may run into.

We’ll start by looking at the middleware pipeline. You saw how to build pipelines by piecing together existing middleware in chapter 3, but in this chapter

you'll create your own custom middleware. You'll explore the basic middleware constructs of the `Map`, `Use`, and `Run` methods and learn how to create standalone middleware classes. You'll use these to build middleware components that can add headers to all your responses as well as middleware that returns responses.

In section 19.2, you'll see how to use your custom middleware to create simple endpoints using endpoint routing. By using endpoint routing, you can take advantage of the power of the routing and authorization systems that you learned about in chapters 5 and 15, without needing the additional complexity that comes from using Web API controllers.

Chapter 11 described the configuration provider system used by ASP.NET Core, but in section 19.3 we'll look at more complex scenarios. In particular, I'll show you how to handle the situation where a configuration provider itself needs some configuration values. For example, a configuration provider that reads values from a database might need a connection string. You'll also see how to use DI when configuring strongly typed `IOptions` objects—something not possible using the methods you've seen so far.

We'll stick with DI in section 19.4, where I'll show you how to replace the built-in DI container with a third-party alternative. The built-in container is fine for most small apps, but your `ConfigureServices` function can quickly get bloated as your app grows and you register more services. I'll show you how to integrate the third-party Lamar library into an existing app, so you can make use of extra features such as automatic service registration by convention.

The components and techniques shown in this chapter are common across all ASP.NET Core applications. For example, I use the subject of the first topic—custom middleware—in almost every project I build. In chapter 20 we'll look at some additional components that are specific to Razor Pages and Web API controllers.

## 19.1 **Customizing your middleware pipeline**

In this section you'll learn how to create custom middleware. You'll learn how to use the `Map`, `Run`, and `Use` extension methods to create simple middleware using lambda expressions. You'll then see how to create equivalent middleware components using dedicated classes. You'll also learn how to split the middleware pipeline into branches, and you'll find out when this is useful.

The middleware pipeline is one of the fundamental building blocks of ASP.NET Core apps, so we covered it in depth in chapter 3. Every request passes through the middleware pipeline, and each middleware component in turn gets an opportunity to modify the request, or to handle it and return a response. ASP.NET Core includes middleware out of the box for handling common scenarios. You'll find middleware for serving static files, for handling errors, for authentication, and many more.

You'll spend most of your time during development working with Razor Pages and Web API controllers. These are exposed as the endpoints for most of your app's business logic, and they call methods on your app's various business services and models.

Sometimes, however, you don't need all the power (and associated complexity) that comes with Razor Pages and Web API controllers. You might want to create a very simple app that, when called, returns the current time. Or you might want to add a health-check URL to an existing app, where calling the URL doesn't do any significant processing but checks that the app is running. Although you *could* use Web API controllers for these, you could also create small, dedicated middleware components to handle these requirements.

Other times, you might have requirements that lie outside the remit of Razor Pages and Web API controllers. For example, you might want to ensure that all responses generated by your app include a specific header. This sort of cross-cutting concern is a perfect fit for custom middleware. You could add the custom middleware early in your middleware pipeline to ensure that every response from your app includes the required header, whether it comes from the static-file middleware, the error handling middleware, or a Razor Page.

In this section, I'll show three ways to create custom middleware components, as well as how to create branches in your middleware pipeline where a request can flow down either one branch or another. By combining the methods demonstrated in this section, you'll be able to create custom solutions to handle your specific requirements.

We'll start by creating a middleware component that returns the current time as plain text, whenever the app receives a request. From there we'll look at branching the pipeline, creating general-purpose middleware components, and finally, how to encapsulate your middleware into standalone classes. In section 19.2 you'll see an alternative approach to exposing response-generating middleware using endpoint routing.

### 19.1.1 Creating simple endpoints with the Run extension

As you've seen in previous chapters, you define the middleware pipeline for your app in the `Configure` method of your `Startup` class. You add middleware to a provided `IApplicationBuilder` object, typically using extension methods. For example,

```
public void Configure(IApplicationBuilder)
{
    app.UseDeveloperExceptionPage();
    app.UseStaticFiles();
}
```

When your app receives a request, the request passes through each middleware, each getting a chance to modify the request, or to handle it by generating a response. If a middleware component generates a response, it effectively short-circuits the pipeline; no subsequent middleware in the pipeline will see the request. The response passes back through the earlier middleware components on its way back to the browser.

You can use the `Run` extension method to build a simple middleware component that always generates a response. This extension takes a single lambda function that runs whenever a request reaches the component. The `Run` extension always generates

a response, so no middleware placed after it will ever execute. For that reason, you should always place the `Run` middleware last in a middleware pipeline.

**TIP** Remember, middleware runs in the order you add them to the pipeline. If a middleware component handles a request and generates a response, later middleware will never see the request.

The `Run` extension method provides access to the request in the form of the `HttpContext` object you saw in chapter 3. This contains all the details of the request in the `Request` property, such as the URL path, the headers, and the body of the request. It also contains a `Response` property you can use to return a response.

The following listing shows how you could build a simple middleware component that returns the current time. It uses the provided `HttpContext` context object and the `Response` property to set the `Content-Type` header of the response (not strictly necessary in this case, as `text/plain` is used if an alternative content type is not set) and writes the body of the response using `WriteAsync(text)`.

#### Listing 19.1 Creating simple middleware using the `Run` extension

```
public void Configure(IApplicationBuilder app)
{
    app.Run(async (HttpContext context) => {
        context.Response.ContentType = "text/plain";
        await context.Response.WriteAsync(
            DateTimeOffset.UtcNow.ToString());
    });
    app.UseStaticFiles();
}
```

Annotations for Listing 19.1:

- A callout points to the first `Run` extension with the text: "Uses the `Run` extension to create a simple middleware that always returns a response".
- A callout points to the `context.Response.ContentType` assignment with the text: "You should set the content-type of the response you're generating—`text/plain` is the default value.".
- A callout points to the `app.UseStaticFiles()` call with the text: "Any middleware added after the `Run` extension will never execute.".
- A callout points to the `context.Response.WriteAsync` call with the text: "Returns the time as a string in the response. The 200 OK status code is used if not explicitly set."

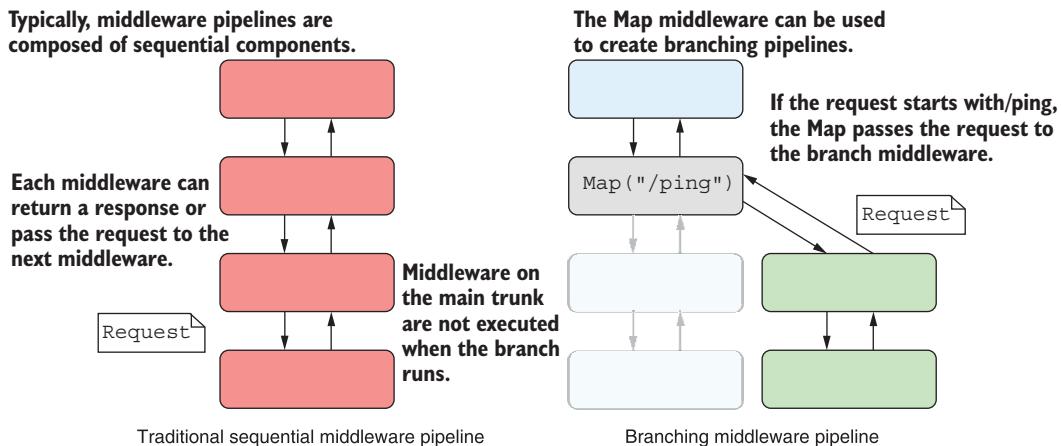
The `Run` extension is useful for building simple middleware. You can use it to create very basic endpoints that always generate a response. But as the component always generates some sort of response, you must always place it at the end of the pipeline, as no middleware placed after it will execute.

A more common scenario is where you want your middleware component to only respond to a specific URL path. In the next section, you'll see how you can combine `Run` with the `Map` extension method to create simple branching middleware pipelines.

#### 19.1.2 Branching middleware pipelines with the `Map` extension

So far, when discussing the middleware pipeline, we've always considered it as a single pipeline of sequential components. Each request passes through every middleware component until one component generates a response, which passes back through the previous middleware.

The `Map` extension method lets you change that simple pipeline into a branching structure. Each branch of the pipeline is independent; a request passes through one branch or the other, but not both, as shown in figure 19.1. The `Map` extension method looks at the path of the request’s URL. If the path matches the required pattern, the request travels down the branch of the pipeline; otherwise it remains on the main trunk. This lets you have completely different behavior in different branches of your middleware pipeline.



**Figure 19.1** A sequential middleware pipeline compared to a branching pipeline created with the `Map` extension. In branching middleware, requests only pass through one of the branches at most. Middleware on the other branch never see the request and aren’t executed.

**NOTE** The URL-matching used by `Map` is conceptually similar to the routing you’ve seen since chapter 6, but it is much more basic, with many limitations. For example, it uses a simple string-prefix match, and you can’t use route parameters. Generally, you should favor creating *endpoints* instead of branching using `Map`, as you’ll see in section 19.2.

For example, imagine you want to add a simple health-check endpoint to your existing app. This endpoint is a simple URL you can call that indicates whether your app is running correctly. You could easily create a health-check middleware using the `Run` extension, as you saw in listing 19.1, but then that’s *all* your app can do. You only want the health-check to respond to a specific URL, `/ping`. Your Razor Pages should handle all other requests as normal.

**TIP** The health-check scenario is a simple example for demonstrating the `Map` method, but ASP.NET Core includes built-in support for health-check endpoints, which you should use instead of creating your own. You can learn more about creating health checks in Microsoft’s “Health checks in ASP.NET Core” documentation: <http://mng.bz/nMA2>.

One solution would be to create a branch using the `Map` extension method and to place the health-check middleware on that branch, as shown in figure 19.1. Only those requests that match the `Map` pattern `/ping` will execute the branch; all other requests will be handled by the standard routing middleware and Razor Pages on the main trunk instead.

### Listing 19.2 Using the `Map` extension to create branching middleware pipelines

```
public void Configure(IApplicationBuilder app)
{
    app.UseDeveloperExceptionPage();                                ← Every request will pass
                                                                through this middleware.

    app.Map("/ping", (IApplicationBuilder branch) =>           ← The Map extension
    {                                                               method will branch if a
        branch.UseExceptionHandler();                            request starts with /ping.

        branch.Run(async (HttpContext context) =>
        {
            context.Response.ContentType = "text/plain";
            await context.Response.WriteAsync("pong");
        });
    });

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

This middleware will only run for requests matching the `/ping` branch.

The rest of the middleware pipeline will run for requests that don't match the `/ping` branch.

The Run extension always returns a response, but only on the `/ping` branch.

The `Map` middleware creates a completely new `IApplicationBuilder` (called `branch` in the listing), which you can customize as you would your main app pipeline. Middleware added to the `branch` builder are only added to the `branch` pipeline, not the main trunk pipeline.

In this example, you add the `Run` middleware to the `branch`, so it will only execute for requests that start with `/ping`, such as `/ping`, `/ping/go`, or `/ping?id=123`. Any requests that don't start with `/ping` are ignored by the `Map` extension. Those requests stay on the main trunk pipeline and execute the next middleware in the pipeline after `Map` (in this case, the `StaticFilesMiddleware`).

If you need to, you can create sprawling branched pipelines using `Map`, where each branch is independent of every other. You could also nest calls to `Map`, so you have branches coming off branches.

The `Map` extension can be useful, but if you try to get too elaborate, it can quickly get confusing. Remember, you should use middleware for implementing cross-cutting concerns or very simple endpoints. The endpoint routing mechanism of controllers and Razor Pages is better suited to more complex routing requirements, so don't be afraid to use it.

**TIP** In section 19.2 you'll see how to create endpoints that use the endpoint routing system.

One situation where `Map` can be useful is when you want to have two “independent” sub-applications but don’t want the hassle of multiple deployments. You can use `Map` to keep these pipelines separate, with separate routing and endpoints inside each branch of the pipeline. Just be aware that these branches will both share the same configuration and DI container, so they’re only independent from the middleware pipeline’s point of view.<sup>1</sup>

The final point you should be aware of when using the `Map` extension is that it modifies the effective `Path` seen by middleware on the branch. When it matches a URL prefix, the `Map` extension cuts off the matched segment from the path, as shown in figure 19.2. The removed segments are stored on a property of `HttpContext` called `PathBase`, so they’re still accessible if you need them.

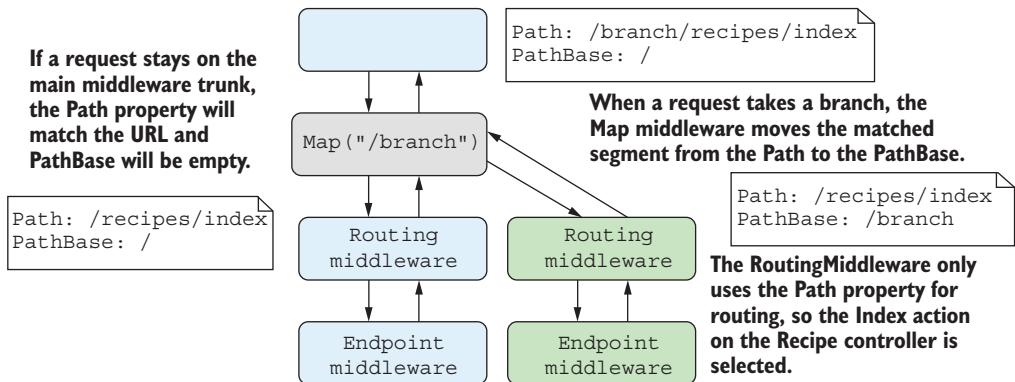


Figure 19.2 When the `Map` extension diverts a request to a branch, it removes the matched segment from the `Path` property and adds it to the `PathBase` property.

**NOTE** ASP.NET Core’s link generator (used in Razor, for example, as discussed in chapter 5) uses `PathBase` to ensure it generates URLs that include the `PathBase` as a prefix.

You’ve seen the `Run` extension, which always returns a response, and the `Map` extension which creates a branch in the pipeline. The next extension we’ll look at is the general-purpose `Use` extension.

<sup>1</sup> Achieving truly independent branches in the same application requires a lot more effort. See Filip W’s blog post, “Running multiple independent ASP.NET Core pipelines side by side in the same application,” for guidance: <http://mng.bz/vzA4>.

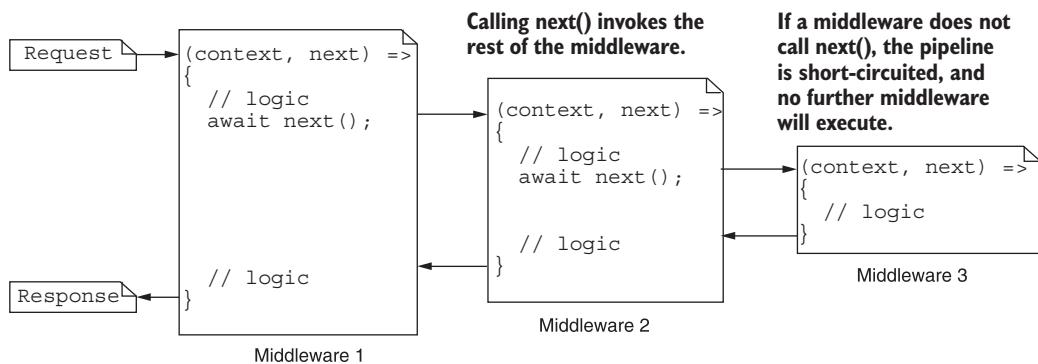
### 19.1.3 Adding to the pipeline with the Use extension

You can use the `Use` extension method to add a general-purpose piece of middleware. You can use it to view and modify requests as they arrive, to generate a response, or to pass the request on to subsequent middleware in the pipeline.

Similar to the `Run` extension, when you add the `Use` extension to your pipeline, you specify a lambda function that runs when a request reaches the middleware. The app passes two parameters to this function:

- *The `HttpContext` representing the current request and response.* You can use this to inspect the request or generate a response, as you saw with the `Run` extension.
- *A pointer to the rest of the pipeline as a `Func<Task>`.* By executing this task, you can execute the rest of the middleware pipeline.

By providing a pointer to the rest of the pipeline, you can use the `Use` extension to control exactly how and when the rest of the pipeline executes, as shown in figure 19.3. If you don't call the provided `Func<Task>` at all, the rest of the pipeline doesn't execute for the request, so you have complete control.



**Figure 19.3** Three pieces of middleware, created with the `Use` extension. Invoking the provided `Func<Task>` using `next()` invokes the rest of the pipeline. Each middleware can run code before and after calling the rest of the pipeline, or it can choose to not call `next()` at all to short-circuit the pipeline.

Exposing the rest of the pipeline as a `Func<Task>` makes it easy to conditionally short-circuit the pipeline, which opens up many different scenarios. Instead of branching the pipeline to implement the health-check middleware with `Map` and `Run`, as you did in listing 19.2, you could use a single instance of the `Use` extension. This provides the same required functionality as before but does so without branching the pipeline.

**Listing 19.3 Using the Use extension method to create a health-check middleware**

```
The Use extension takes a lambda with HttpContext  
(context) and Func<Task> (next) parameters.
```

```
public void Configure(IApplicationBuilder app)
{
    app.Use(async (HttpContext context, Func<Task> next) =>
    {
        if (context.Request.Path.StartsWithSegments("/ping"))
        {
            context.Response.ContentType = "text/plain";
            await context.Response.WriteAsync("pong");
        }
        else
        {
            await next();
        }
    });
    app.UseStaticFiles();
}
```

```
The StartsWithSegments  
method looks for the  
provided segment in  
the current path.
```

If the path matches, generate a response and short-circuit the pipeline.

If the incoming request starts with the required path segment (/ping), the middleware responds and doesn't call the rest of the pipeline. If the incoming request doesn't start with /ping, the extension calls the next middleware in the pipeline, with no branching necessary.

With the Use extension, you have control over when, and if, you call the rest of the middleware pipeline. But it's important to note that you generally shouldn't modify the Response object after calling next(). Calling next() runs the rest of the middleware pipeline, so a subsequent middleware may start streaming the response to the browser. If you try to modify the response *after* executing the pipeline, you may end up corrupting the response or sending invalid data.

**WARNING** Don't *modify* the Response object after calling next(). Also, don't call next() if you've written to the Response.Body; writing to this Stream can trigger Kestrel to start streaming the response to the browser, and you could cause invalid data to be sent. You can generally *read* from the Response object safely; for example, to inspect the final StatusCode or ContentType of the response.

Another common use for the Use extension method is to modify every request or response that passes through it. For example, there are various HTTP headers that you should send with all your applications for security reasons. These headers often disable old, insecure, legacy behaviors by browsers, or restrict the features enabled by the browser. You learned about the HSTS header in chapter 18, but there are other headers you can add for additional security.<sup>2</sup>

<sup>2</sup> You can test the security headers for your app using <https://securityheaders.com/>, which also provides information about what headers you should add to your application and why.

Imagine you've been tasked with adding one such header, X-Content-Type-Options: nosniff (which provides added protection against XSS attacks), to every response generated by your app. This sort of cross-cutting concern is perfect for middleware. You can use the `Use` extension method to intercept every request, set the response header, and then execute the rest of the middleware pipeline. No matter what response the pipeline generates, whether it's a static file, an error, or a Razor Page, the response will always have the security header.

Listing 19.4 shows a robust way to achieve this. When the middleware receives a request, it registers a callback that runs before Kestrel starts sending the response back to the browser. It then calls `next()` to run the rest of the middleware pipeline. When the pipeline generates a response, likely in some later middleware, Kestrel executes the callback and adds the header. This approach ensures the header isn't accidentally removed by other middleware in the pipeline and also that you don't try to modify the headers after the response has started streaming to the browser.

#### Listing 19.4 Adding headers to a response with the `Use` extension

```

Sets a function that should be called before
the response is sent to the browser
    public void Configure(IApplicationBuilder app)
    {
        app.Use(async (HttpContext context, Func<Task> next) =>
        {
            context.Response.OnStarting(() =>
            {
                context.Response.Headers["X-Content-Type-Options"] =
                    "nosniff";
                return Task.CompletedTask;
            });
            await next();
        });
    }

The function
passed to
OnStarting must
return a Task.
    app.UseStaticFiles();
    app.UseRouting();
    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllers();
    });
}

```

→ Adds the middleware at the start of the pipeline

→ Adds the header to the response, for added protection against XSS attacks

→ Invokes the rest of the middleware pipeline

No matter what response is generated, it'll have the security header added.

Simple cross-cutting middleware like the security header example are common, but they can quickly clutter your `Configure` method and make it difficult to understand the pipeline at a glance. Instead, it's common to encapsulate your middleware into a class that's functionally equivalent to the `Use` extension, but which can be easily tested and reused.

### 19.1.4 Building a custom middleware component

Creating middleware with the `Use` extension, as you did in listings 19.3 and 19.4, is convenient, but it's not easy to test, and you're somewhat limited in what you can do. For example, you can't easily use DI to inject scoped services inside of these basic middleware components. Normally, rather than calling the `Use` extension directly, you'll encapsulate your middleware into a class that's functionally equivalent.

Custom middleware components don't have to derive from a specific base class or implement an interface, but they have a certain shape, as shown in listing 19.5. ASP.NET Core uses reflection to execute the method at runtime. Middleware classes should have a constructor that takes a `RequestDelegate` object, which represents the rest of the middleware pipeline, and they should have an `Invoke` function with a signature similar to

```
public Task Invoke(HttpContext context);
```

The `Invoke()` function is equivalent to the lambda function from the `Use` extension, and it is called when a request is received. Here's how you could convert the headers middleware from listing 19.4 into a standalone middleware class.<sup>3</sup>

#### Listing 19.5 Adding headers to a Response using a custom middleware component

```
public class HeadersMiddleware
{
    private readonly RequestDelegate _next;
    public HeadersMiddleware(RequestDelegate next)
    {
        _next = next;
    }

    public async Task Invoke(HttpContext context)
    {
        context.Response.OnStarting(() =>
        {
            context.Response.Headers["X-Content-Type-Options"] =
                "nosniff";
            return Task.CompletedTask;
        });

        await _next(context);
    }
}
```

**Adds the header to the response as before**

**The RequestDelegate represents the rest of the middleware pipeline.**

**The Invoke method is called with HttpContext when a request is received.**

**Invokes the rest of the middleware pipeline. Note that you must pass in the provided HttpContext.**

<sup>3</sup> Using this “shape” approach makes the middleware more flexible. In particular, it means you can easily use DI to inject services into the `Invoke` method. This wouldn't be possible if the `Invoke` method were an overridden base class method or an interface. However, if you prefer, you can implement the `IMiddleware` interface, which defines the basic `Invoke` method.

This middleware is effectively identical to the example in listing 19.4, but it's encapsulated in a class called `HeadersMiddleware`. You can add this middleware to your app in `Startup.Configure` by calling

```
app.UseMiddleware<HeadersMiddleware>();
```

A common pattern is to create helper extension methods to make it easy to consume your extension method from `Startup.Configure` (so that IntelliSense reveals it as an option on the `IApplicationBuilder` instance). Here's how you could create a simple extension method for `HeadersMiddleware`.

#### Listing 19.6 Creating an extension method to expose `HeadersMiddleware`

```
public static class MiddlewareExtensions
{
    public static IApplicationBuilder UseSecurityHeaders(
        this IApplicationBuilder app)
    {
        return app.UseMiddleware<HeadersMiddleware>(); ←
    }
}
```

By convention, the extension method should return an `IApplicationBuilder` to allow chaining.

Adds the middleware to the pipeline

With this extension method, you can now add the headers middleware to your app using

```
app.UseSecurityHeaders();
```

**TIP** There is a `SecurityHeaders` NuGet package available that makes it easy to add security headers using middleware, without having to write your own. The package provides a fluent interface for adding the recommended security headers to your app. You can find instructions on how to install it on GitHub: <https://github.com/andrewlock/NetEscapades.AspNetCore.SecurityHeaders>.

Listing 19.5 is a simple example, but you can create middleware for many different purposes. In some cases you may need to use DI to inject services and use them to handle a request. You can inject singleton services into the constructor of your middleware component, or you can inject services with any lifetime into the `Invoke` method of your middleware, as demonstrated in the following listing.

#### Listing 19.7 Using DI in middleware components

```
public class ExampleMiddleware
{
    private readonly RequestDelegate _next;
    private readonly ServiceA _a;
    public HeadersMiddleware(RequestDelegate next, ServiceA a)
    {
        _next = next;
        _a = a;
    }
}
```

You can inject additional services in the constructor. These must be singletons.

```
public async Task Invoke(
    HttpContext context, ServiceB b, ServiceC c)
{
    // use services a, b, and c
    // and/or call _next.Invoke(context);
}
```

You can inject services into the `Invoke` method. These may have any lifetime.

**WARNING** ASP.NET Core creates the middleware only once for the lifetime of your app, so any dependencies injected in the constructor must be singletons. If you need to use scoped or transient dependencies, inject them into the `Invoke` method.

That covers pretty much everything you need to start building your own middleware components. By encapsulating your middleware into custom classes, you can easily test their behavior or distribute them in NuGet packages, so I strongly recommend taking this approach. Apart from anything else, it will make your `Startup.Configure()` method less cluttered and easier to understand.

## 19.2 Creating custom endpoints with endpoint routing

In this section you'll learn how to create custom endpoints from your middleware using endpoint routing. We'll take the simple middleware branches used in section 19.1 and convert them to use endpoint routing, and I'll demonstrate the additional features this enables, such as routing and authorization.

In section 19.1 I described creating a simple endpoint, using the `Map` and `Run` extension methods, that returns a plain-text pong response whenever a `/ping` request is received, by branching the middleware pipeline. This is fine because it's so simple, but what if you have more complex requirements?

Consider a basic enhancement of the ping-pong example: how would you add authorization to the request? The `AuthorizationMiddleware` added to your pipeline by `UseAuthorization()` looks for metadata on endpoints like Razor Pages to see if there's an `[Authorize]` attribute, but it doesn't know how to work with the ping-pong `Map` extension.

Similarly, what if you wanted to use more complex routing? Maybe you want to be able to call `/ping/3` and have your ping-pong middleware reply pong-pong-pong (no, I can't think why you would either!). You now have to try and parse that integer from the URL, make sure it's valid, and so on. That's sounding like a lot more work!

When your requirements start ramping up like this, one option is to move to using Web API controllers or Razor Pages. These provide the greatest flexibility in your app and have the most features, but they're also comparatively heavyweight compared to middleware. What if you want something in between?

In ASP.NET Core 3.0, the routing system was rewritten to use *endpoint routing* to provide exactly this balance. Endpoint routing allows you to create endpoints that can use the same routing and authorization framework as you get with Web API controllers and Razor Pages, but with the simplicity of middleware.

**NOTE** I discussed endpoint routing in detail in chapter 5.

In this section you'll see how to convert the simple branch-based middleware from the previous section to a custom endpoint. You'll see how taking this approach makes it easy to apply authorization to the endpoint, using the declarative approaches you're already familiar with from chapter 15.

### 19.2.1 Creating a custom endpoint routing component

As I described in chapter 5, endpoint routing splits the process of executing an endpoint into two steps implemented by two separate pieces of middleware:

- **RoutingMiddleware**—This middleware uses the incoming request to *select* an endpoint to execute. It exposes the metadata about the selected endpoint on `HttpContext`, such as authorization requirements applied using the `[Authorize]` attribute.
- **EndpointMiddleware**—This middleware *executes* the selected endpoint to generate a response.

The advantage of using a two-step process is that you can place middleware *between* the middleware that selects the endpoint and the middleware that executes it to generate a response. For example, the `AuthorizationMiddleware` uses the selected endpoint to determine whether to short-circuit the pipeline, *before* the endpoint is executed.

Let's imagine that you need to apply authorization to the simple ping-pong endpoint you created in section 19.1.2. This is much easier to achieve with endpoint routing than using simple middleware branches like `Map` or `Use`. The first step is to create a middleware component for the functionality, using the approach you saw in section 19.1.4, as shown in the following listing.

#### Listing 19.8 The PingPongMiddleware implemented as a middleware component

```
public class PingPongMiddleware
{
    public PingPongMiddleware(RequestDelegate next)
    {
    }

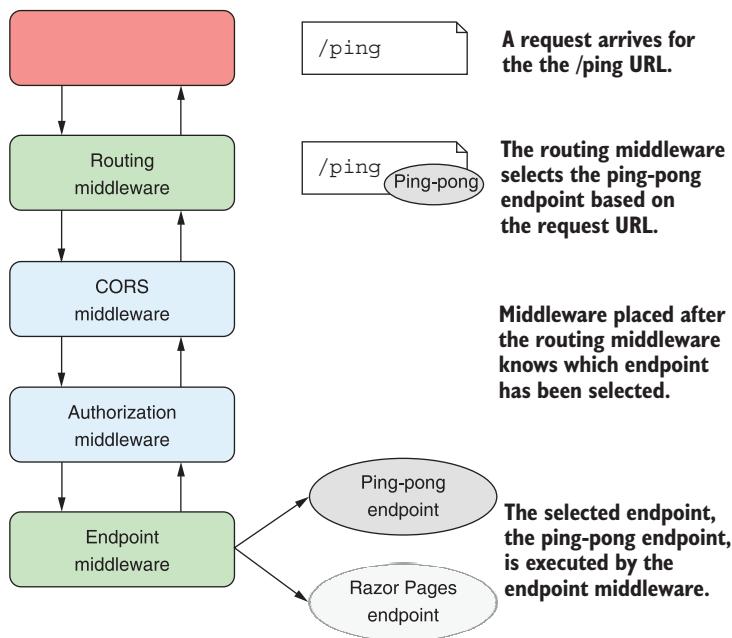
    public async Task Invoke(HttpContext context)
    {
        context.Response.ContentType = "text/plain";
        await context.Response.WriteAsync("pong");
    }
}
```

Even though it isn't used in this case, you must inject a `RequestDelegate` in the constructor.

Invoke is called to execute the middleware.

The middleware always returns a "pong" response.

Note that this middleware always returns a "pong" response, regardless of the request URL—we will configure the `"/ping"` path later. We can use this class to convert a middleware pipeline from the branching version shown in figure 19.1, to the endpoint version shown in figure 19.4.



**Figure 19.4** Endpoint routing separates the selection of an endpoint from the execution of an endpoint. The routing middleware selects an endpoint based on the incoming request and exposes metadata about the endpoint. Middleware placed before the endpoint middleware can act based on the selected endpoint, such as short-circuiting unauthorized requests. If the request is authorized, the endpoint middleware executes the selected endpoint and generates a response.

Converting the ping-pong middleware to an endpoint doesn't require any changes to the middleware itself. Instead, you need to create a mini middleware pipeline, containing your ping-pong middleware only.

**TIP** Converting response-generating middleware to an endpoint essentially requires converting it into its own mini-pipeline, so you can include additional middleware in the endpoint pipeline if you wish.

You must create your endpoint pipeline *inside* the `UseEndpoints()` lambda argument, as shown in the following listing. Use `CreateApplicationBuilder()` to create a new `IApplicationBuilder`, add your middleware that makes up your endpoint, and then call `Build()` to create the pipeline.

#### Listing 19.9 Mapping the ping-pong endpoint in `UseEndpoints`

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

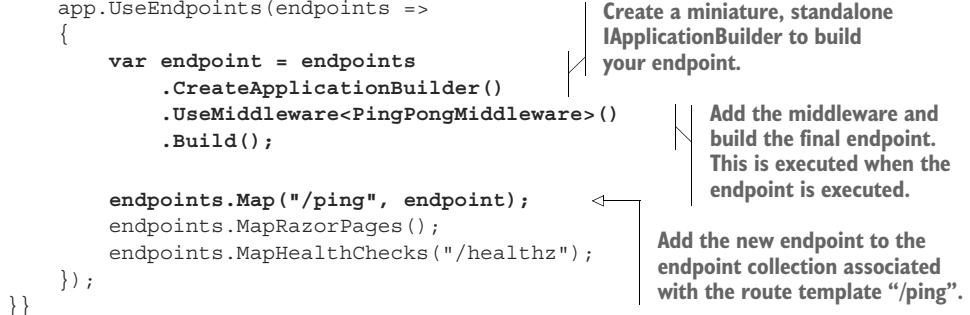
```

app.UseAuthentication();
app.UseAuthorization();

app.UseEndpoints(endpoints =>
{
    var endpoint = endpoints
        .CreateApplicationBuilder()
        .UseMiddleware<PingPongMiddleware>()
        .Build();

    endpoints.Map("/ping", endpoint);
    endpoints.MapRazorPages();
    endpoints.MapHealthChecks("/healthz");
})
}

```



Once you have a pipeline, you can associate it with a given route by calling `Map()` on the `IEndpointRouteBuilder` instance and passing in a route template.

**TIP** Note that the `Map()` function on `IEndpointRouteBuilder` creates a new endpoint (consisting of your mini-pipeline) with an associated route. Although it has the same name, this is conceptually different from the `Map` function on `IApplicationBuilder` from section 19.1.2, which is used to *branch* the middleware pipeline.

If you have many custom endpoints, the `UseEndpoints()` method can quickly get cluttered. I like to extract this functionality into an extension method to make the `UseEndpoints()` method cleaner and easier to read. The following listing extracts the code to create an endpoint from listing 19.9 into a separate class, taking the route template to use as a method parameter.

#### Listing 19.10 An extension method for using the PingPongMiddleware as an endpoint

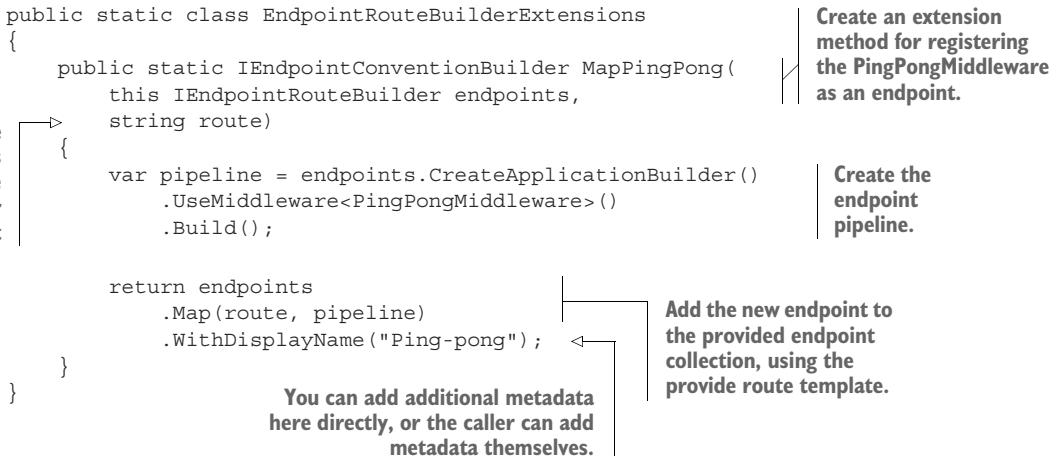
```

public static class EndpointRouteBuilderExtensions
{
    public static IEndpointConventionBuilder MapPingPong(
        this IEndpointRouteBuilder endpoints,
        string route)
    {
        var pipeline = endpoints.CreateApplicationBuilder()
            .UseMiddleware<PingPongMiddleware>()
            .Build();

        return endpoints
            .Map(route, pipeline)
            .WithDisplayName("Ping-pong");
    }
}

Allows the caller to pass in a route template for the endpoint

```



Now that you have an extension method, `MapPingPong()`, you can update your `UseEndpoints()` method in `Startup.Configure()` to be simpler and easier to understand:

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapPingPong("/ping");
    endpoints.MapRazorPages();
    endpoints.MapHealthChecks("/healthz");
});
```

Congratulations, you've created your first custom endpoint! You haven't added any additional *functionality* yet, but by using the endpoint routing system it's now much easier to satisfy the additional authorization requirements, as you'll see in section 19.2.2.

**TIP** This example used a very basic route template, `"/ping"`, but you can also use templates that contain route parameters, such as `"/ping/{count}"`, using the same routing framework you learned in chapter 5. For examples of how to access this data from your middleware, as well as best practice advice, see my blog entry titled "Accessing route values in endpoint middleware in ASP.NET Core 3.0": <http://mng.bz/4ZRj>.

Converting existing middleware like `PingPongMiddleware` to work with endpoint routing can be useful when you have already implemented that middleware, but it's a lot of boilerplate to write if you want to create a new simple endpoint. In ASP.NET Core 5.0, this process is a lot easier.

## 19.2.2 Creating simple endpoints with `MapGet` and `WriteJsonAsync`

With ASP.NET Core 5.0 it's easier than ever to create simple endpoints that use routing and return JSON. ASP.NET Core 5.0 builds on top of the endpoint routing primitives introduced in ASP.NET Core 3.0 and adds helper methods for serializing and deserializing JSON. In many cases, this is all you need to create simple endpoints, when you don't need the full power of Web API controllers.

For example, instead of converting the `PingPongMiddleware` to an endpoint and having to create a dedicated pipeline, as I showed in section 19.2.1, you could create an equivalent endpoint directly. The following listing shows how to use the `MapGet` extension method to write a lambda handler method inline, which is functionally identical to the approach shown in section 19.2.1.

### Listing 19.11 Creating a custom endpoint directly using `MapGet`

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

    app.UseAuthentication();
    app.UseAuthorization();
```

```

app.UseEndpoints(endpoints =>
{
    endpoints.MapGet("/ping", (HttpContext ctx) =>
        ctx.Response.WriteAsync("pong"));
    endpoints.MapRazorPages();
    endpoints.MapHealthChecks("/healthz");
});
}

```

Configure a lambda to execute when a GET request is routed to the /ping endpoint

This is equivalent to the example in section 19.2.1 but is clearly much simpler—there are no additional classes or helper methods required! There are alternative extension methods for other HTTP verbs, such as `MapPost` and `MapPut`, so you can create multiple endpoints inline in this way.

**TIP** As in section 19.2.1, this example uses a very basic route template, `"/ping"`, but you can also use templates that contain route parameters, such as `"/ping/{count}"`. See the associated source code for a simple example.

It's important to understand the difference between the `MapGet` *endpoint* extension method shown here, and the `Map` *application builder* extension method discussed in section 19.1:

- *Use Map to create separate branches of the middleware pipeline.* Use `MapGet` to register an endpoint with the routing middleware that generates a response.
- *Map uses simple URL segments to decide which branch to take; it does not use route parameters.* `MapGet` uses route templates, the same as Razor Pages and Web API controllers, as discussed in chapter 5.

The `MapGet` and `MapPost` extension methods were available in ASP.NET Core 3.0 too, but ASP.NET Core 5.0 introduced some extra extension methods that allow easy reading and writing of JSON using the `System.Text.Json` serializer. The following listing shows a simple “echo” endpoint that reads JSON from the request body and then writes it to the response.

#### Listing 19.12 Writing JSON from a custom endpoint using `WriteAsJsonAsync`

```

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

    app.UseAuthentication();
    app.UseAuthorization();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapPost("/echo", async (HttpContext ctx =>
        {
            MyCustomType model = await ctx.Request
                .ReadFromJsonAsync<MyCustomType>();
            await context.Response.WriteAsJsonAsync(model);
        }));
    });
}

```

Configure a lambda to execute when a GET request is routed to the /ping endpoint.

Use System.Text.Json to serialize the model object as JSON to the response body.

**Use System.Text.Json to deserialize the body of the request to a `MyCustomType` object.**

```
        endpoints.MapRazorPages();
        endpoints.MapHealthChecks("/healthz");
    });
}}
```

The `ReadFromJsonAsync` function allows you to deserialize the body of a request into a POCO object—`MyCustomType` in this example. This is *similar* to the model binding you saw in chapter 6, but it is much more rudimentary. There's no built-in validation or reading from route values here, and any errors in the data of the request will cause an exception to be thrown.

Similarly, `WriteAsJsonAsync` serializes any object passed to it to the response as JSON. This is very efficient compared to the `ActionResult` approach used in Razor Pages and Web API controllers, but it has far fewer features. There's no filter pipeline, no content-negotiation, and no error handling built in.

**TIP** If all you need is a quick, very simple API, the `Map*` extension methods and JSON helpers are a great option. If you need more features like validation, model binding, supporting multiple formats, or OpenAPI integration, stick to Web API controllers.

One of the advantages of endpoint routing, whether you use the `Map*` extension methods or the existing-middleware approach from section 19.2.1 is that you get the full power of route templates. However, using simple `Map` branches can be faster than using the routing infrastructure, so in some cases it may be best to avoid endpoint routing.

A good example of this trade-off is the built-in `StaticFileMiddleware`. This middleware serves static files based on the request's URL, but it *doesn't* use endpoint routing due to the performance impact of adding many (potentially hundreds) of routes for each static file in your application. The downside to that choice is that adding authorization to static files is not easy to achieve; if endpoint routing were used, adding authorization would be simple.

### 19.2.3 Applying authorization to endpoints

One of the main advantages of endpoint routing is the ability to easily apply authorization to your endpoint. For Razor Pages and Web API controllers, this is achieved by adding the `[Authorize]` attribute, as you saw in chapter 15.

For other endpoints, such as the ping-pong endpoint you created in section 19.2.1, you can apply authorization declaratively when you add the endpoint to your application, by calling `RequireAuthorization()` on the `IEndpointConventionBuilder`, as shown in the following listing.

#### Listing 19.13 Applying authorization to an endpoint using `RequireAuthorization()`

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

```

app.UseAuthentication();
app.UseAuthorization();

app.UseEndpoints(endpoints =>
{
    endpoints.MapPingPong("/ping")
        .RequireAuthorization();
    endpoints.MapRazorPages();
    endpoints.MapHealthChecks("/healthz")
        .RequireAuthorization("HealthCheckPolicy");
});
}

```

Require authorization.  
This is equivalent to  
applying the [Authorize]  
attribute.
Require authorization  
using a specific policy,  
HealthCheckPolicy.

Listing 19.13 shows two examples of applying authorization to endpoints:

- `RequireAuthorization()`—If you don't provide a method argument, this applies the default authorization policy to the endpoint. It is equivalent to applying the `[Authorize]` attribute to a Razor Page or Web API controller endpoint.
- `RequireAuthorization(policy)`—If you provide a policy name, the chosen authorization policy will be used. The policy must be configured in `Configure-Services`, as you saw in chapter 15. This is equivalent to applying `[Authorize]("HealthCheckPolicy")` to a Razor Page or Web API controller endpoint.

If you are globally applying authorization to your application (as described in chapter 15), you can punch a hole in the global policy with the complementary `AllowAnonymous()` method; for example,

```
endpoints.MapPingPong("/ping").AllowAnonymous();
```

This is equivalent to using the `[AllowAnonymous]` attribute on your Razor Pages and actions.

**NOTE** The `AllowAnonymous()` method for endpoints is new in .NET 5.0.

Authorization is the canonical example of adding metadata to endpoints to add functionality, but there are other options available too. Out of the box you can use the following methods:

- `RequireAuthorization()`—Applies authorization policies to the endpoint, as you've already seen.
- `AllowAnonymous()`—Overrides a global authorization policy to allow anonymous access to an endpoint.
- `RequireCors(policy)`—Applies a CORS policy to the endpoint, as described in chapter 18.
- `RequireHost(hosts)`—Only allows routing to the endpoint if the incoming request matches one of the provided hostnames.
- `WithDisplayName(name)`—Sets the friendly name for the endpoint. Used primarily in logging to describe the endpoint.

- `WithMetadata(items)`—Adds arbitrary values as metadata to the endpoint. You can access these values in middleware after an endpoint is selected by the routing middleware.

These features allow various functionality, such as CORS and authorization, to work seamlessly across Razor Pages, Web API controllers, built-in endpoints like the health-check endpoints, and custom endpoints like your ping-pong middleware. They should allow you to satisfy most requirements you get around custom endpoints. And if you find you need something more complex, like model-binding, you can always fall back to using Web API controllers instead. The choice is yours!

In the next section we'll move away from the middleware pipeline and look at how to handle complex configuration requirements. In particular, you'll see how to set up complex configuration providers that require their own configuration values, and how to use DI services to build your strongly typed `IOptions` objects.

### 19.3 Handling complex configuration requirements

In this section I'll describe how to handle two complex configuration requirements: configuration providers that need configuring themselves, and using services to configure `IOptions` objects. In the first scenario, you will see how to partially build your configuration to allow building the provider. In the second scenario, you will see how to use the `IConfigureOptions` interface to allow accessing services when configuring your options objects.

In chapter 11 we discussed the ASP.NET Core configuration system in depth. You saw how an `IConfiguration` object is built from multiple layers, where subsequent layers can add to or replace configuration values from previous layers. Each layer is added by a configuration provider, which can read values from a file, from environment variables, from User Secrets, or from any number of possible locations.

You also saw how to *bind* configuration values to strongly typed POCO objects, using the `IOptions` interface. You can inject these strongly typed objects into your classes to provide easy access to your settings, instead of having to read configuration using `string` keys.

In this section, we'll look at two scenarios that are slightly more complex. In the first scenario, you're trying to use a configuration provider that requires some configuration itself. As an example, imagine you want to store some configuration in a database table. In order to load these values, you'd need some sort of connection string, which would most likely also come from your `IConfiguration`. You're stuck with a chicken-and-egg situation: you need to build the `IConfiguration` object to add the provider, but you can't add the provider without building the `IConfiguration` object!

In the second scenario, you want to configure a strongly typed `IOptions` object with values returned from a service. But the service won't be available until after you've configured all of the `IOptions` objects. In section 19.3.2, you'll see how to handle this by implementing the `IConfigureOptions` interface.

### 19.3.1 Partially building configuration to configure additional providers

ASP.NET Core includes many configuration providers, such as file and environment variable providers, that don't require anything more than basic details to set up. All you need in order to read a JSON file, for example, is the path to that file.

But the configuration system is highly extensible, and more complex configuration providers may require some degree of configuration themselves. For example, you may have a configuration provider that loads configuration values from a database, a provider that loads values from a remote API, or a provider that loads secrets from Azure Key Vault.<sup>4</sup>

Each of these providers requires some sort of configuration themselves: a connection string for the database, a URL for the remote service, or a key to decrypt the data from Key Vault. Unfortunately, this leaves you with a circular problem: you need to add the provider to build your configuration object, but you need a configuration object to add the provider!

The solution is to use a two-stage process to build your final `IConfiguration` configuration object, as shown in figure 19.5. In the first stage, you load the configuration values that are available locally, such as JSON files and environment variables, and build a temporary `IConfiguration`. You can use this object to configure the complex providers, add them to your configuration builder, and build the final `IConfiguration` for your app.

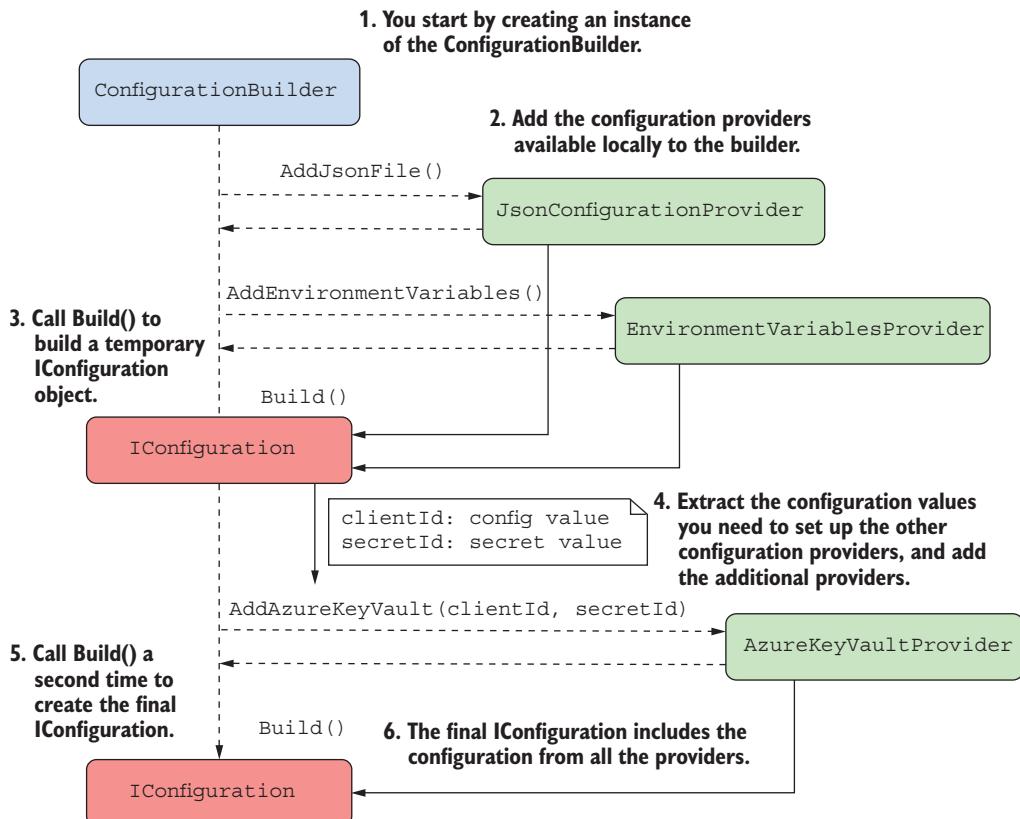
You can use this two-phase process whenever you have configuration providers that need configuration themselves. Building the configuration object twice means that the values are loaded from each of the initial configuration providers twice, but I've never found that to be a problem.

As an example of this process, listing 19.14 shows how you can create a temporary `IConfiguration` object, built using the contents of an XML file. This contains a configuration property called "SettingsFile" with the filename of a JSON file.

In the second phase of configuration, you add the JSON file provider (using the filename from the partial `IConfiguration`) and the environment variable provider. When you finally call `Build()` on the `IHostBuilder`, a new `IConfiguration` object will be built, containing the configuration values from the XML file, the JSON file, and the environment variables.

---

<sup>4</sup> Azure Key Vault is a service that lets you securely store secrets in the cloud. Your app retrieves the secrets from Azure Key Vault at runtime by calling an API and providing a client ID and a secret. The client ID and secret must come from local configuration values so that you can retrieve the rest from Azure Key Vault. Read more, including how to get started, in Microsoft's "Azure Key Vault Configuration Provider in ASP.NET Core" documentation: <http://mng.bz/Qm7v>.



**Figure 19.5** Adding a configuration provider that requires configuration. Start by adding configuration providers that you have the details for, and build a temporary `IConfiguration` object. You can use this configuration object to load the settings required by the complex provider, add the provider to your builder, and build the final `IConfiguration` object using all the providers.

#### **Listing 19.14 Using multiple IConfiguration objects to configure providers**

```
public static IHostBuilder CreateHostBuilder(string[] args) =>
    Host.CreateDefaultBuilder(args)
        .ConfigureAppConfiguration((context, config) =>
    {
        config.Sources.Clear();
        config.AddXmlFile("baseconfig.xml");
    });
    IConfiguration partialConfig = config.Build();
    string filename = partialConfig["SettingsFile"];
}

Adds an XML file to the configuration, which contains configuration for other providers
```

**Remove the default configuration sources.**

**Builds the IConfiguration to read the XML file**

**Extracts the configuration required by other providers**

```

Uses the extracted configuration to configure other providers    → config.AddJsonFile(filename)
                                                               .AddEnvironmentVariables();      ← Remember, values from
                                                               .ConfigureWebHostDefaults(webBuilder =>                                subsequent providers will
                                                               {                                     overwrite values from
                                                               webBuilder.UseStartup<Startup>(); previous providers.
                                                               });

```

This is a somewhat contrived example—it's unlikely that you'd need to load configuration values to read a JSON file—but the principle is the same no matter which providers you use. A good example of this is the Azure Key Vault provider. To load configuration values from Azure Key Vault, you need a URL, a client ID, and a secret. These must be loaded from other configuration providers, so you have to use the same two-phase process as shown in the previous listing.

Once you've loaded the configuration for your app, it's common to *bind* this configuration to strongly typed objects using the `IOptions` pattern. In the next section we'll look at other ways to configure your `IOptions` objects and how to build them using DI services.

### 19.3.2 Using services to configure `IOptions` with `IConfigureOptions`

A common and encouraged practice is to bind your configuration object to strongly typed `IOptions<T>` objects, as you saw in chapter 11. Typically, you configure this binding in `Startup.ConfigureServices` by calling `services.Configure<T>()` and providing an `IConfiguration` object or section to bind.

To bind a strongly typed object, called `CurrencyOptions`, to the "Currencies" section of an `IConfiguration` object, you'd use the following:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CurrencyOptions>(
        Configuration.GetSection("Currencies"));
}
```

This sets the properties of the `CurrencyOptions` object, based on the values in the "Currencies" section of your `IConfiguration` object. Simple binding like this is common, but sometimes you might want to customize the configuration of your `IOptions<T>` objects, or you might not want to bind them to configuration at all. The `IOptions` pattern only requires you to configure a strongly typed object before it's injected into a dependent service; it doesn't mandate that you *have* to bind it to an `IConfiguration` section.

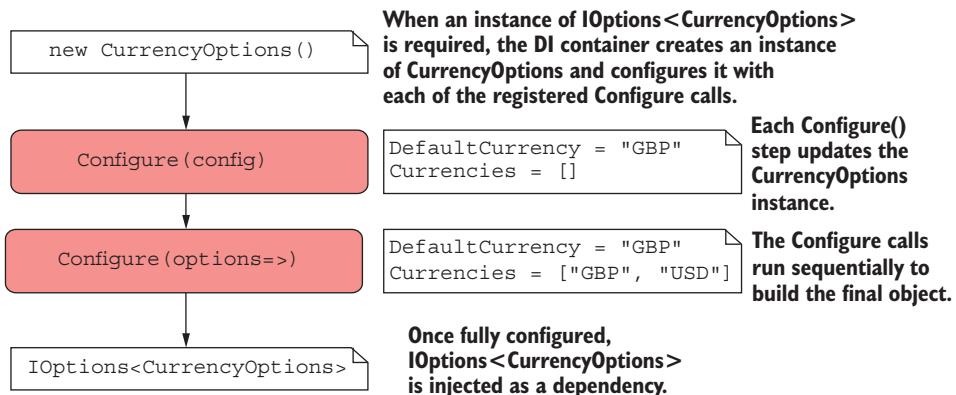
**TIP** Technically, even if you don't configure an `IOptions<T>` at all, you can still inject it into your services. In that case, the `T` object will always be created using the default constructor.

The `services.Configure<T>()` method has an overload that lets you provide a lambda function that the framework uses to configure the `CurrencyOptions` object.

For example, in the following snippet we use a lambda function to set the Currencies property on the configured CurrencyOptions object to a fixed array of strings:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CurrencyOptions>(
        Configuration.GetSection("Currencies"));
    services.Configure<CurrencyOptions>(options =>
        options.Currencies = new string[] { "GBP", "USD"});
}
```

Each call to `Configure<T>()`, both the binding to `IConfiguration` and the lambda function, adds another configuration step to the `CurrencyOptions` object. When the DI container first requires an instance of `IOptions<CurrencyOptions>`, each of the steps run in turn, as shown in figure 19.6.



**Figure 19.6 Configuring a `CurrencyOptions` object.** When the DI container needs an `IOptions<T>` instance of a strongly typed object, the container creates the object and then uses each of the registered `Configure()` methods to set the object's properties.

In the previous code snippet, you set the `Currencies` property to a static array of strings in a lambda function. But what if you don't know the correct values ahead of time? You might need to load the available currencies from a database, or from some remote service, for example.

Unfortunately, this situation, where you need a configured service to configure your `IOptions<T>`, is hard to resolve. Remember, you declare your `IOptions<T>` configuration inside `ConfigureServices` as part of the DI configuration. How can you get a fully configured instance of a currency service if you haven't registered it with the container yet?

The solution is to defer the configuration of your `IOptions<T>` object until the last moment, just before the DI container needs to create it to handle an incoming request.

At that point, the DI container will be completely configured and will know how to create the currency service.

ASP.NET Core provides this mechanism with the `IConfigureOptions<T>` interface. You implement this interface in a configuration class and use it to configure the `IOptions<T>` object in any way you need. This class can use DI, so you can easily use any other required services.

#### Listing 19.15 Implementing `IConfigureOptions<T>` to configure an options object

You can inject services that are only available after the DI is completely configured.

```
public class ConfigureCurrencyOptions : IConfigureOptions<CurrencyOptions>
{
    private readonly ICurrencyProvider _currencyProvider;
    public ConfigureCurrencyOptions(ICurrencyProvider currencyProvider)
    {
        _currencyProvider = currencyProvider;
    }

    public void Configure(CurrencyOptions options)
    {
        options.Currencies = _currencyProvider.GetCurrencies();
    }
}
```

**Configure is called when an instance of `IOptions<CurrencyOptions>` is required.**

**You can use the injected service to load the values from a remote API, for example.**

All that remains is to register this implementation in the DI container. As always, order is important, so if you want `ConfigureCurrencyOptions` to run *after* binding to configuration, you must add it *after* the first call to `services.Configure<T>()`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.Configure<CurrencyOptions>
        Configuration.GetSection("Currencies"));
    services.AddSingleton
        <IConfigureOptions<CurrencyOptions>, ConfigureCurrencyOptions>();
}
```

**WARNING** The `CurrencyConfigureOptions` object is registered as a singleton, so it will capture any injected services of scoped or transient lifetimes.<sup>5</sup>

With this configuration, when `IOptions<CurrencyOptions>` is injected into a controller or service, the `CurrencyOptions` object will first be bound to the "Currencies" section of your `IConfiguration` and will then be configured by the `ConfigureCurrencyOptions` class.

---

<sup>5</sup> If you inject a scoped service into your configuration class (for example, a `DbContext`), you need to do a bit more work to ensure it's disposed of correctly. I describe how to achieve that in my blog article titled "Access services inside ConfigureServices using `IConfigureOptions` in ASP.NET Core": <http://mng.bz/6m17>.

One piece of configuration not yet shown is `ICurrencyProvider`, used by `Configure-CurrencyOptions`. In the sample code for this chapter, I created a simple `CurrencyProvider` service and registered it with the DI container using

```
services.AddSingleton<ICurrencyProvider, CurrencyProvider>() ;
```

As your app grows and you add extra features and services, you'll probably find yourself writing more and more of these simple DI registrations, where you register a Service that implements `IService`. The built-in ASP.NET Core DI container requires you to explicitly register each of these services manually. If you find this requirement frustrating, it may be time to look at third-party DI containers that can take care of some of the boilerplate for you.

## 19.4 Using a third-party dependency injection container

In this section I'll show you how to replace the default dependency injection container with a third-party alternative, Lamar. Third-party containers often provide additional features compared to the built-in container, such as assembly scanning, automatic service registration, and property injection. Replacing the built-in container can also be useful when porting an existing app that uses a third-party DI container to ASP.NET Core.

The .NET community had been using DI containers for years before ASP.NET Core decided to include one that is built in. The ASP.NET Core team wanted a way to use DI in their own framework libraries, and they wanted to create a common abstraction<sup>6</sup> that allows you to replace the built-in container with your favorite third-party alternative, such as,

- Autofac
- StructureMap/Lamar
- Ninject
- Simple Injector
- Unity

The built-in container is intentionally limited in the features it provides, and it won't be getting many more realistically. In contrast, third-party containers can provide a host of extra features. These are some of the features available in Lamar (<https://jasperfx.github.io/lamar/documentation/ioc/>), the spiritual successor to StructureMap (<https://structuremap.github.io/>):

- Assembly scanning for interface/implementation pairs based on conventions
- Automatic concrete class registration

---

<sup>6</sup> Although the promotion of DI as a core practice has been applauded, this abstraction has seen some controversy. This post, titled "What's wrong with the ASP.NET Core DI abstraction?" from one of the maintainers of the SimpleInjector DI library describes many of the arguments and concerns: <http://mng.bz/yYAd>. You can also read more about the decisions on GitHub: <https://github.com/aspnet/DependencyInjection/issues/433>.

- Property injection and constructor selection
- Automatic Lazy<T>/Func<T> resolution
- Debugging/testing tools for viewing inside your container

None of these features are a requirement for getting an application up and running, so using the built-in container makes a lot of sense if you’re building a small app or you’re new to DI containers in general. But if, at some undefined tipping point, the simplicity of the built-in container becomes too much of a burden, it may be worth replacing.

**TIP** A middle-of-the-road approach is to use the Scrutor NuGet package, which adds some features to the built-in DI container, without replacing it entirely. For an introduction and examples, see my blog post, “Using Scrutor to automatically register your services with the ASP.NET Core DI container”: <http://mng.bz/MX7B>.

In this section, I’ll show how you can configure an ASP.NET Core app to use Lamar for dependency resolution. It won’t be a complex example, or an in-depth discussion of Lamar itself. Instead, I’ll cover the bare minimum to get you up and running.

Whichever third-party container you choose to install into an existing app, the overall process is pretty much the same:

- 1 Install the container NuGet package.
- 2 Register the third-party container with `IHostBuilder` in `Program.cs`.
- 3 Add a `ConfigureContainer` method in `Startup`.
- 4 Configure the third-party container in `ConfigureContainer` to register your services.

Most of the major .NET DI containers have been ported to work on .NET Core and include an adapter that lets you add them to ASP.NET Core apps. For details, it’s worth consulting the specific guidance for the container you’re using. For Lamar, the process looks like this:

- 1 Install the `Lamar.Microsoft.DependencyInjection` NuGet package using the NuGet package manager, by running `dotnet add package`:

```
dotnet add package Lamar.Microsoft.DependencyInjection
```

Or by adding a `<PackageReference>` to your `.csproj` file:

```
<PackageReference  
Include="Lamar.Microsoft.DependencyInjection" Version="4.4.0" />
```

- 2 Call `UseLamar()` on your `IHostBuilder` in `Program.cs`:

```
public static IHostBuilder CreateHostBuilder(string[] args) =>  
    Host.CreateDefaultBuilder(args)  
        .UseLamar()  
        .ConfigureWebHostDefaults(webBuilder =>
```

```
{
    webBuilder.UseStartup<Startup>();
}
```

- 3 Add a ConfigureContainer method to your Startup class, with the following signature:

```
public void ConfigureContainer(ServiceRegistry services) { }
```

- 4 Configure the Lamar ServiceRegistry in ConfigureContainer, as shown in the following listing. This is a basic configuration, but you can see a more complex example in the source code for this chapter.

#### Listing 19.16 Configuring Lamar as a third-party DI container

```
public void ConfigureContainer(ServiceRegistry services) {
{
    services.AddAuthorization();
    services.AddControllers()
        .AddControllersAsServices(); // You can (and should) add ASP.NET Core framework services to the Service-Registry, as usual.

    services.Scan(_ => {
        _.AssemblyContainingType(typeof(Startup));
        _.WithDefaultConventions();
    });
}
```

**Required so that Lamar is used to build your Web API controllers**

**Configure your services in ConfigureContainer instead of ConfigureServices.**

**Lamar can automatically scan your assemblies for services to register.**

In this example, I've used the default conventions to register services. This will automatically register concrete classes and services that are named following expected conventions (for example, Service implements `IService`). You can change these conventions or add other registrations in the ConfigureContainer method.

The `ServiceRegistry` passed into `ConfigureContainer` implements `IServiceCollection`, which means you can use all the built-in extension methods, such as `AddControllers()` and `AddAuthorization()`, to add framework services to your container.

**WARNING** If you're using DI in your MVC controllers (almost certainly!) and you register those dependencies with Lamar rather than the built-in container, you may need to call `AddControllersAsServices()`, as shown in listing 19.16. This is due to an implementation detail in the way your MVC controllers are created by the framework. For details, see my blog entry titled "Controller activation and dependency injection in ASP.NET Core MVC": <http://mng.bz/aogm>.

With this configuration in place, whenever your app needs to create a service, it will request it from the Lamar container, which will create the dependency tree for the class and create an instance. This example doesn't show off the power of Lamar, so be sure to check out the documentation (<https://jasperfx.github.io/lamar/>) and the

associated source code for this chapter for more examples. Even in modestly sized applications, Lamar can greatly simplify your service registration code, but its party trick is showing all the services you have registered, and any associated issues.

That brings us to the end of this chapter on custom components. In this chapter I focused on some of the most common components you will build for the configuration, dependency injection, and middleware systems of ASP.NET Core. In the next chapter you'll learn about more custom components, with a focus on Razor Pages and Web API controllers.

## Summary

- Use the `Run` extension method to create middleware components that always return a response. You should always place the `Run` extension at the end of a middleware pipeline or branch, as middleware placed after it will never execute.
- You can create branches in the middleware pipeline with the `Map` extension. If an incoming request matches the specified path prefix, the request will execute the pipeline branch; otherwise it will execute the trunk.
- When the `Map` extension matches a request path segment, it removes the segment from the request's `HttpContext.Path` and moves it to the `PathBase` property. This ensures that routing in branches works correctly.
- You can use the `Use` extension method to create generalized middleware components that can generate a response, modify the request, or pass the request on to subsequent middleware in the pipeline. This is useful for cross-cutting concerns, like adding a header to all responses.
- You can encapsulate middleware in a reusable class. The class should take a `RequestDelegate` object in the constructor and should have a public `Invoke()` method that takes an `HttpContext` and returns a `Task`. To call the next middleware in the pipeline, invoke the `RequestDelegate` with the provided `HttpContext`.
- To create endpoints that generate a response, build a miniature pipeline containing the response-generating middleware, and call `endpoints.Map(route, pipeline)`. Endpoint routing will be used to map incoming requests to your endpoint.
- Alternatively, use the `MapGet` and other `Map*` extension methods to create endpoints inline. Endpoint routing will map incoming requests to these endpoints in the same way as your other custom endpoints.
- You can read and write JSON inside a `Map*` endpoint using the `ReadFromJsonAsync` and `WriteAsJsonAsync` extension methods. These use the `System.Text.Json` serializer to deserialize and serialize objects directly. These can be useful for creating simple APIs. Be aware that this approach lacks most of the features available with Web API controllers, such as validation, the filter pipeline, and content negotiation.

- You can attach metadata to endpoints, which is made available to any middleware placed between the calls to `UseRouting()` and `UseEndpoints()`. This metadata enables functionality such as authorization and CORS.
- To add authorization to an endpoint, call `RequireAuthorization()` after mapping the endpoint. This is equivalent to using the `[Authorize]` attribute on Razor Pages and Web API controllers. You can optionally provide an authorization policy name, instead of using the default policy.
- Some configuration providers require configuration values themselves. For example, a configuration provider that loads settings from a database might need a connection string. You can load these configuration providers by partially building an `IConfiguration` object using the other providers and reading the required configuration from it. You can then configure the database provider and add it to the `ConfigurationBuilder` before rebuilding to get the final `IConfiguration`.
- If you need to use services from the DI container to configure an `IOptions<T>` object, you should create a separate class that implements `IConfigureOptions<T>`. This class can use DI in the constructor and is used to lazily build a requested `IOptions<T>` object at runtime.
- You can replace the built-in DI container with a third-party container. Third-party containers often provide additional features, such as convention-based dependency registration, assembly scanning, or property injection.
- To use a third-party container such as Lamar, install the NuGet package, enable the container on `IHostBuilder`, and implement `ConfigureContainer()` in `Startup`. Configure the third-party container in this method by registering both the required ASP.NET Core framework services and your app-specific services.



# *Building custom MVC and Razor Pages components*

---

## **This chapter covers**

- Creating custom Razor Tag Helpers
- Using view components to create complex Razor views
- Creating a custom `DataAnnotations` validation attribute
- Replacing the `DataAnnotations` validation framework with an alternative

In the previous chapter, you learned how to customize and extend some of the core systems in ASP.NET Core: configuration, dependency injection, and your middleware pipeline. These components form the basis of all ASP.NET Core apps. In this chapter we're focusing on Razor Pages and MVC/API controllers. You'll learn how to build custom components that work with Razor views. You'll also learn how to build components that work with the validation framework used by both Razor Pages and API controllers.

We'll start by looking at Tag Helpers. In section 20.1 I'll show you how to create two different Tag Helpers: one that generates HTML to describe the current machine, and one that lets you write `if` statements in Razor templates without having

to use C#. These will give you the details you need to create your own custom Tag Helpers in your own apps if the need arises.

In section 20.2 you'll learn about a new Razor concept: view components. View components are a bit like partial views, but they can contain business logic and database access. For example, on an e-commerce site you might have a shopping cart, a dynamically populated menu, and a login widget, all on one page. Each of those sections is independent of the main page content and has its own logic and data-access needs. In an ASP.NET Core app using Razor Pages, you'd implement each of those as a view component.

In section 20.3 I'll show you how to create a custom validation attribute. As you saw in chapter 6, validation is a key responsibility of Razor Page handlers and action methods, and the `DataAnnotations` attributes provide a clean, declarative way of doing so. We previously only looked at the built-in attributes, but you'll often find you need to add attributes tailored to your app's domain. In section 20.3 you'll see how to create a simple validation attribute, and how to extend it to use services registered with the DI container.

Throughout this book I've mentioned that you can easily swap out core parts of the ASP.NET Core framework if you wish. In section 20.4 you'll do just that, by replacing the built-in attribute-based validation framework with a popular alternative, FluentValidation. This open source library allows you to separate your binding models from the validation rules, which makes building certain validation logic easier. Many people prefer this approach of separating concerns to the declarative approach of `DataAnnotations`.

When you're building pages with Razor Pages, one of the best productivity features is Tag Helpers, and in the next section you'll see how you can create your own.

## 20.1 Creating a custom Razor Tag Helper

In this section you'll learn how to create your own Tag Helpers, which allow you to customize your HTML output. You'll learn how to create Tag Helpers that add new elements to your HTML markup, as well as Tag Helpers that can be used to remove or customize existing markup. You'll also see that your custom Tag Helpers integrate with the tooling of your IDE to provide rich IntelliSense in the same way as the built-in Tag Helpers.

In my opinion, Tag Helpers are one of the best additions to the venerable Razor template language in ASP.NET Core. They allow you to write Razor templates that are easier to read, as they require less switching between C# and HTML, and they *augment* your HTML tags, rather than replacing them (as opposed to the HTML Helpers used extensively in the previous version of ASP.NET).

ASP.NET Core comes with a wide variety of Tag Helpers (see chapter 8), which will cover many of your day-to-day requirements, especially when it comes to building forms. For example, you can use the Input Tag Helper by adding an `asp-for` attribute

to an `<input>` tag and passing a reference to a property on your `PageModel`, in this case `Input.Email`:

```
<input asp-for="Input.Email" />
```

The Tag Helper is activated by the presence of the attribute and gets a chance to augment the `<input>` tag when rendering to HTML. The Input Tag Helper uses the name of the property to set the `<input>` tag's `name` and `id` properties, the value of the model to set the `value` property, and the presence of attributes such as `[Required]` or `[EmailAddress]` to add attributes for validations:

```
<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."
    />
```

Tag Helpers help reduce the duplication in your code, or they can simplify common patterns. In this section I'll show how you can create your own custom Tag Helpers.

In section 20.1.1 you'll create a system information Tag Helper, which prints details about the name and operating system of the server your app is running on. In section 20.1.2 you'll create a Tag Helper that you can use to conditionally show or hide an element based on a C# Boolean property. In section 20.1.3 you'll create a Tag Helper that reads the Razor content written *inside* the Tag Helper and transforms it.

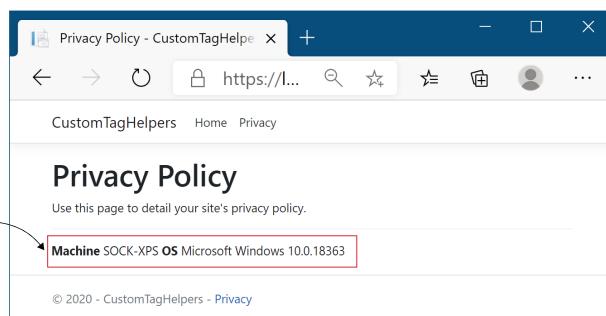
### 20.1.1 Printing environment information with a custom Tag Helper

A common problem you may run into when you start running your web applications in production, especially if you're using a server-farm setup, is working out which machine rendered the page you're currently looking at. Similarly, when deploying frequently, it can be useful to know which *version* of the application is running. When I'm developing and testing, I sometimes like to add a little "info dump" at the bottom of my layouts so I can easily work out which server generated the current page, which environment it's running in, and so on.

In this section, I'm going to show you how to build a custom Tag Helper to output system information to your layout. You'll be able to toggle the information it displays, but by default it will display the machine name and operating system on which the app is running, as shown in figure 20.1.

You can call this Tag Helper from Razor by creating a `<system-info>` element in your template:

```
<footer>
  <system-info></system-info>
</footer>
```



**Figure 20.1** The `SystemInfoTagHelper` displays the machine name and operating system on which the application is running. It can be useful for identifying which instance of your app handled the request when running in a web farm scenario.

**TIP** You probably don't want to expose this sort of information in production, so you could also wrap it in an `<environment>` Tag Helper, as you saw in chapter 8.

The easiest way to create a custom Tag Helper is to derive from the `TagHelper` base class and override the `Process()` or `ProcessAsync()` function that describes how the class should render itself. The following listing shows your complete custom Tag Helper, `SystemInfoTagHelper`, which renders the system information to a `<div>`. You could easily extend this class if you wanted to display additional fields or add additional options.

#### Listing 20.1 `SystemInfoTagHelper` to render system information to a view

Derives from the `TagHelper` base class

```
public class SystemInfoTagHelper : TagHelper
{
    private readonly HtmlEncoder _htmlEncoder;
    public SystemInfoTagHelper(HtmlEncoder htmlEncoder)
    {
        _htmlEncoder = htmlEncoder;
    }

    [HtmlAttributeName("add-machine")]
    public bool IncludeMachine { get; set; } = true;

    [HtmlAttributeName("add-os")]
    public bool IncludeOS { get; set; } = true;

    public override void Process(
        TagHelperContext context, TagHelperOutput output)
    {
        output.TagName = "div";
        output.TagMode = TagMode.StartTagAndEndTag;
        var sb = new StringBuilder();
    }
}
```

An `HtmlEncoder` is necessary when writing HTML content to the page.

Renders both the `<div>` </code> start and end tag

Decorating properties with `HtmlAttributeName` allows you to set their values from Razor markup.

The main function called when an element is rendered

Replaces the `<system-info>` element with a `<div>` element

```

If required, adds a <strong> element and the HTML-encoded machine name
    if (IncludeMachine)
    {
        sb.Append(" <strong>Machine</strong> ");
        sb.Append(_htmlEncoder.Encode(Environment.MachineName));
    }

If required, adds a <strong> element and the HTML-encoded OS name
    if (IncludeOS)
    {
        sb.Append(" <strong>OS</strong> ");
        sb.Append(
            _htmlEncoder.Encode(RuntimeInformation.OSDescription));
    }
}

output.Content.SetHtmlContent(sb.ToString());

```

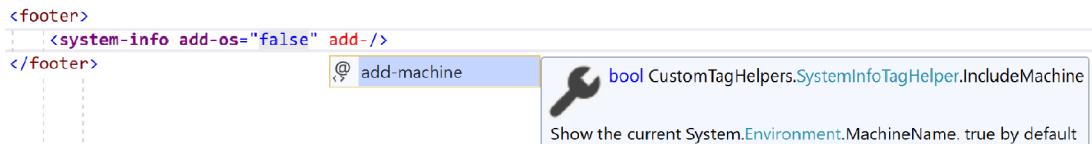
↳ **Sets the inner content of the <div> tag with the HTML-encoded value stored in the string builder**

There's a lot of new code in this example, so we'll work through it line by line. First, the class name of the Tag Helper defines the name of the element you must create in your Razor template, with the suffix removed and converted to kebab-case. As this Tag Helper is called `SystemInfoTagHelper`, you must create a `<system-info>` element.

**TIP** If you want to customize the name of the element, for example to `<env-info>`, but you want to keep the same class name, you can apply `[HtmlTargetElement]` with the desired name, such as `[HtmlTargetElement("Env-Info")]`. HTML tags are not case-sensitive, so you could use "Env-Info" or "env-info".

Inject an `HtmlEncoder` into your Tag Helper so you can HTML-encode any data you write to the page. As you saw in chapter 18, you should always HTML-encode data you write to the page to avoid XSS vulnerabilities (and to ensure the data is displayed correctly).

You've defined two properties on your Tag Helper, `IncludeMachine` and `IncludeOS`, which you'll use to control which data is written to the page. These are decorated with a corresponding `[HtmlAttributeName]`, which enables setting the properties from the Razor template. In Visual Studio, you'll even get IntelliSense and type-checking for these values, as shown in figure 20.2.



**Figure 20.2** In Visual Studio, Tag Helpers are shown in a purple font, and you get IntelliSense for properties decorated with `[HtmlAttributeName]`.

Finally, we come to the `Process()` method. The Razor engine calls this method to execute the Tag Helper when it identifies the target element in a view template. The

`Process()` method defines the type of tag to render (`<div>`), whether it should render a start and end tag (or a self-closing tag—it depends on the type of tag you’re rendering), and the HTML content of the `<div>`. You set the HTML content to be rendered inside the tag by calling `Content.SetHtmlContent()` on the provided instance of `TagHelperOutput`.

**WARNING** Always HTML-encode your output before writing to your tag with `SetHtmlContent()`. Alternatively, pass unencoded input to `SetContent()`, and the output will be automatically HTML-encoded for you.

Before you can use your new Tag Helper in a Razor template, you need to register it. You can do this in the `_ViewImports.cshtml` file, using the `@addTagHelper` directive and specifying the fully qualified name of the Tag Helper and the assembly. For example,

```
@addTagHelper CustomTagHelpers.SystemInfoTagHelper, CustomTagHelpers
```

Alternatively, you can add all the Tag Helpers from a given assembly by using the wild-card syntax, `*`, and specifying the assembly name:

```
@addTagHelper *, CustomTagHelpers
```

With your custom Tag Helper created and registered, you’re now free to use it in any of your Razor views, partial views, or layouts.

**TIP** If you’re not seeing IntelliSense for your Tag Helper in Visual Studio, and the Tag Helper isn’t rendered in the bold font used by Visual Studio, then you probably haven’t registered your Tag Helpers correctly in `_ViewImports.cshtml` using `@addTagHelper`.

The `SystemInfoTagHelper` is an example of a Tag Helper that generates content, but you can also use Tag Helpers to control how existing elements are rendered. In the next section you’ll create a simple Tag Helper that can control whether or not an element is rendered, based on an HTML attribute.

## 20.1.2 Creating a custom Tag Helper to conditionally hide elements

If you want to control whether an element is displayed in a Razor template based on some C# variable, you’d typically wrap the element in a C# `if` statement:

```
@{  
    var showContent = true;  
}  
@if(showContent)  
{  
    <p>The content to show</p>  
}
```

Falling back to C# constructs like this can be useful, as it allows you to generate any markup you like. Unfortunately, it can be mentally disruptive having to switch back

and forth between C# and HTML, and it makes it harder to use HTML editors that don't understand Razor syntax.

In this section you'll create a simple Tag Helper to avoid the cognitive dissonance problem. You can apply this Tag Helper to existing elements to achieve the same result as shown previously, but without having to fall back to C#:

```
@{
    var showContent = true;
}
<p if="showContent">
    The content to show
</p>
```

Instead of creating a new *element*, as you did for `SystemInfoTagHelper` (`<system-info>`), you'll create a Tag Helper that you apply as an *attribute* to existing HTML elements. This Tag Helper does one thing: it controls the visibility of the element it's attached to. If the value passed in the `if` attribute is `true`, the element and its content is rendered as normal. If the value passed is `false`, the Tag Helper removes the element and its content from the template. Here's how you could achieve this.

### Listing 20.2 Creating an `IfTagHelper` to conditionally render elements

```
[HtmlTargetElement(Attributes = "if")]
public class IfTagHelper : TagHelper
{
    [HtmlAttributeName("if")]
    public bool RenderContent { get; set; } = true;

    public override void Process(
        TagHelperContext context, TagHelperOutput output)
    {
        if(RenderContent == false)
        {
            output.TagName = null;
            output.SuppressOutput();
        }
    }

    public override int Order => int.MinValue;
}
```

**Binds the value of the if attribute to the RenderContent property**

**If the RenderContent property evaluates to false, removes the element**

**Setting the Attributes property ensures the Tag Helper is triggered by an if attribute.**

**The Razor engine calls Process() to execute the Tag Helper.**

**Sets the element the Tag Helper resides on to null, removing it from the page**

**Doesn't render or evaluate the inner content of the element**

**Ensures this Tag Helper runs before any others attached to the element**

Instead of a standalone `<if>` element, the Razor engine executes the `IfTagHelper` whenever it finds an element with an `if` attribute. This can be applied to any HTML element: `<p>`, `<div>`, `<input>`, whatever you need. You should define a Boolean property specifying whether you should render the content, which is bound to the value in the `if` attribute.

The `Process()` function is much simpler here. If `RenderContent` is `false`, it sets the `TagHelperOutput.TagName` to `null`, which removes the element from the page.

You also call `SuppressOutput()`, which prevents any content *inside* the attributed element from being rendered. If `RenderContent` is true, you skip these steps and the content is rendered as normal.

One other point of note is the overridden `Order` property. This controls the order in which Tag Helpers run when multiple Tag Helpers are applied to an element. By setting `Order` to `int.MinValue`, you ensure that `IfTagHelper` will run first, removing the element if required, before other Tag Helpers execute. There's generally no point running other Tag Helpers if the element is going to be removed from the page anyway.

**NOTE** Remember to register your custom Tag Helpers in `_ViewImports.cshtml` with the `@addTagHelper` directive.

With a simple HTML attribute, you can now conditionally render elements in Razor templates without having to fall back to C#. This Tag Helper can show and hide content without needing to know what the content is. In the next section we'll create a Tag Helper that *does* need to know the content.

### 20.1.3 Creating a Tag Helper to convert Markdown to HTML

The two Tag Helpers shown so far are agnostic to the content written *inside* the Tag Helper, but it can also be useful to create Tag Helpers that inspect, retrieve, and modify this content. In this section you'll see an example of one such Tag Helper that converts Markdown content written inside it into HTML.<sup>1</sup>

**DEFINITION** Markdown is a commonly used text-based markup language that is easy to read but can also be converted into HTML. It is the common format used by README files on GitHub, and I use it to write blog posts, for example. For an introduction to Markdown, see the GitHub guide: <https://guides.github.com/features/mastering-markdown/>.

We'll use the popular Markdig library (<https://github.com/xoofx/markdig>) to create the Markdown Tag Helper. This library converts a string containing Markdown into an HTML string. You can install Markdig using Visual Studio by running `dotnet add package Markdig`, or by adding a `<PackageReference>` to your `.csproj` file:

```
<PackageReference Include="Markdig" Version="0.22.0" />
```

The Markdown Tag Helper that we'll create shortly can be used by adding `<markdown>` elements to your Razor Page, as shown in the following listing.

---

<sup>1</sup> The open source library WebAPIContrib.Core includes a similar tag helper, as well as other helpful utilities. You can find it on NuGet and GitHub at <https://github.com/WebApiContrib/WebAPIContrib.Core>.

### Listing 20.3 Using a Markdown Tag Helper in a Razor Page

```
@page
@model IndexModel

<markdown>
## This is a markdown title
This is a markdown list:
* Item 1
* Item 2
</markdown>

<div if="showContent">
    Content is shown when showContent is true
</div>
```

The Markdown Tag Helper is added using the `<markdown>` element.

Titles can be created in Markdown using # to denote h1, ## to denote h2, and so on.

Markdown converts simple lists to HTML `<ul>` elements.

Razor content can be nested inside other Tag Helpers.

The Markdown Tag Helper renders content with these steps:

- 1 Render any Razor content inside the Tag Helper. This includes executing any *nested* Tag Helpers and C# code inside the Tag Helper. Listing 20.3 uses the `IfTagHelper`, for example.
- 2 Convert the resulting string to HTML using the Markdig library.
- 3 Replace the content with the rendered HTML and remove the Tag Helper `<markdown>` element.

The following listing shows a simple approach to implementing a Markdown Tag Helper using Markdig. Markdig supports many additional extensions and features that you could enable, but the overall pattern of the Tag Helper would be the same.

### Listing 20.4 Implementing a Markdown Tag Helper using Markdig

```
public class MarkdownTagHelper : TagHelper
{
    public override async Task ProcessAsync(
        TagHelperContext context, TagHelperOutput output)
    {
        TagHelperContent markdownRazorContent = await
            output.GetChildContentAsync(NullHtmlEncoder.Default);
        string markdown =
            markdownRazorContent.GetContent(NullHtmlEncoder.Default);

        string html = Markdig.Markdown.ToHtml(markdown);
```

Render the Razor contents to a string.

The Markdown Tag Helper will use the `<markdown>` element.

Retrieve the contents of the `<markdown>` element.

Convert the Markdown string to HTML using Markdig.

Write the HTML content to the output.

Remove the `<markdown>` element from the content.

When rendered to HTML, the Markdown content in listing 20.3 (when the `showContent` variable is `true`) becomes:

```
<h2>This is a markdown title</h2>
<p>This is a markdown list:</p>
<ul>
<li>Item 1</li>
<li>Item 2</li>
</ul>
<div>
    Content is shown when showContent is true
</div>
```

**NOTE** In listing 20.4 we implemented `ProcessAsync()` instead of `Process()`. That is because we call the `async` method `GetChildContentAsync()`. You must only call `async` methods from other `async` methods; otherwise you can get issues such as thread starvation. For more details, see Microsoft’s “ASP.NET Core Performance Best Practices”: <http://mng.bz/KM7X>.

The Tag Helpers in this section represent a small sample of possible avenues you could explore, but they cover the two broad categories: Tag Helpers for rendering new content, and Tag Helpers for controlling the rendering of other elements.<sup>2</sup>

Tag Helpers can be useful for providing small pieces of isolated, reusable functionality like this, but they’re not designed to provide larger, application-specific sections of an app or to make calls to business-logic services. Instead, you should use view components, as you’ll see in the next section.

## 20.2 View components: Adding logic to partial views

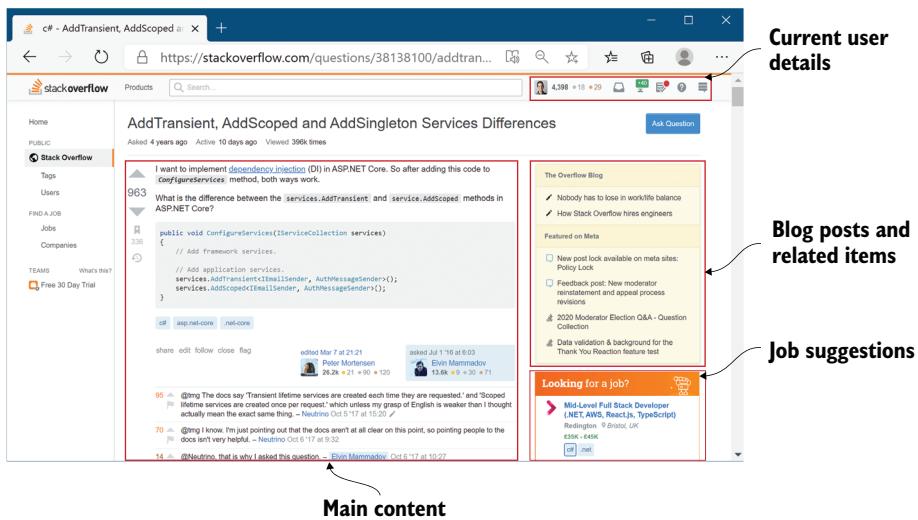
In this section you’ll learn about *view components*. View components operate independently of the main Razor Page and can be used to encapsulate complex business logic. You can use view components to keep your main Razor Page focused on a single task, rendering the main content, instead of also being responsible for other sections of the page.

If you think about a typical website, you’ll notice that they often have multiple independent dynamic sections in addition to the main content. Consider Stack Overflow, shown in figure 20.3, for example. As well as the main body of the page showing questions and answers, there’s a section showing the current logged-in user, a panel for blog posts and related items, and a section for job suggestions.

Each of these sections is effectively independent of the main content. Each section contains business logic (deciding which posts or ads to show), database access (loading the details of the posts), and rendering logic for how to display the data. In chapter 7 you saw that you can use layouts and partial views to split up the rendering of a view template into similar sections, but partial views aren’t a good fit for this example.

---

<sup>2</sup> For further details and examples, see Microsoft’s “Author Tag Helpers in ASP.NET Core” documentation at <http://mng.bz/Idb0>.



**Figure 20.3** The Stack Overflow website has multiple sections that are independent of the main content, but which contain business logic and complex rendering logic. Each of these sections could be rendered as a view component in ASP.NET Core.

Partial views let you encapsulate *view rendering* logic, but not *business logic* that's independent of the main page content.

Instead, view components provide this functionality, encapsulating both the business logic and rendering logic for displaying a small section of the page. You can use DI to provide access to a database context, and you can test view components independently of the view they generate, much like MVC and API controllers. Think of them a bit like mini-MVC controllers, or mini-Razor Pages, but you invoke them directly from a Razor view, instead of in response to an HTTP request.

**TIP** View components are comparable to *child actions* from the previous version of ASP.NET, in that they provide similar functionality. Child actions don't exist in ASP.NET Core.

### View components vs. Razor Components and Blazor

In this book I am focusing on server-side rendered applications using Razor Pages and API applications using Web API controllers. .NET Core 3.0 introduced a completely new approach to building ASP.NET Core applications: Blazor. I don't cover Blazor in this book, so I recommend reading *Blazor in Action* by Chris Sainty (Manning, 2021).

Blazor has two programming models, client-side and server-side, but both approaches use *Blazor components* (confusingly, officially called *Razor components*). Blazor components have a lot of parallels with view components, but they live in a fundamentally

different world. Blazor components can interact with each other easily, but you can't use them with Tag Helpers or view components, and it's hard to combine them with Razor Page form posts.

Nevertheless, if you need an "island" of rich client-side interactivity in a single Razor Page, you can embed a Blazor component inside a Razor Page, as shown in the "Render components in a page or view with the Component Tag Helper" section of the "Prerender and integrate ASP.NET Core Razor components" documentation: <http://mng.bz/PPen>. You could also use Blazor components as a way to replace AJAX calls in your Razor Pages, as I show in my blog entry, "Replacing AJAX calls in Razor Pages with Razor Components and Blazor": <http://mng.bz/9MJj>.

If you don't need the client-side interactivity of Blazor, view components are still the best option for isolated sections in Razor Pages. They interoperate cleanly with your Razor Pages, have no additional operational overhead, and use familiar concepts like layouts, partial views, and Tag Helpers. For more details on why you should continue to use view components, see my "Don't replace your View Components with Razor Components" blog entry: <http://mng.bz/1rKq>.

In this section, you'll see how to create a custom view component for the recipe app you built in previous chapters, as shown in figure 20.4. If the current user is logged in, the view component displays a panel with a list of links to the user's recently created recipes. For unauthenticated users, the view component displays links to the login and register actions.

The figure consists of two screenshots of a web application. Both screenshots show a header with 'Hello andrewlock.net@gmail.com! Logout'. The left sidebar contains a 'Your Recipes' section with a red border, listing 'Recent recipes you've created' such as 'Soy Garlic Salmon Couscous' and 'Spaghetti Bolognese'. The right sidebar shows a 'TimeToCook' table with three rows: 'Soy Garlic Salmon Couscous' (0hrs 35mins), 'Magret De Canard' (0hrs 0mins), and 'Tasty Chilli' (0hrs 0mins), each with a 'View' button. In the top screenshot, the user is logged in, so the 'Your Recipes' section is populated with recent recipes. In the bottom screenshot, the user is not logged in, so the 'Your Recipes' section instead displays 'Register' and 'Login' links.

**Figure 20.4** The view component displays different content based on the currently logged-in user. It includes both business logic (determining which recipes to load from the database) and rendering logic (specifying how to display the data).

This component is a great candidate for a view component as it contains database access and business logic (choosing which recipes to display), as well as rendering logic (deciding how the panel should be displayed).

**TIP** Use partial views when you want to encapsulate the rendering of a specific view model, or part of a view model. When you have rendering logic that requires business logic or database access, or if the section is logically distinct from the main page content, consider using a view component.

You invoke view components directly from Razor views and layouts using a Tag Helper-style syntax with a `vc:` prefix:

```
<vc:my-recipes number-of-recipes="3">
</vc:my-recipes>
```

Custom view components typically derive from the `ViewComponent` base class and implement an `InvokeAsync()` method, as shown in listing 20.5. Deriving from this base class allows access to useful helper methods in much the same way that deriving from the `ControllerBase` class does for API controllers. Unlike API controllers, the parameters passed to `InvokeAsync` don't come from model binding. Instead, you pass the parameters to the view component using properties on the Tag Helper element in your Razor view.

#### Listing 20.5 A custom view component to display the current user's recipes

Deriving from the `ViewComponent` base class provides useful methods like `View()`.

```
public class MyRecipesViewComponent : ViewComponent
{
    private readonly RecipeService _recipeService;
    private readonly UserManager< ApplicationUser > _userManager;
    public MyRecipesViewComponent(RecipeService recipeService,
        UserManager< ApplicationUser > userManager)
    {
        _recipeService = recipeService;
        _userManager = userManager;
    }

    public async Task< IViewComponentResult > InvokeAsync( ←
        int numberOfRecipes)
    {
        if (!User.Identity.IsAuthenticated)
        {
            return View("Unauthenticated");
        }

        var userId = _userManager.GetUserId(HttpContext.User);
        var recipes = await _recipeService.GetRecipesForUser(
            userId, numberOfRecipes);
    }
}
```

You can pass parameters to the component from the view.

Calling `View()` will render a partial view with the provided name.

You can use DI in a view component.

`InvokeAsync` renders the view component. It should return a `Task< IViewComponentResult >`.

You can use `async` external services, allowing you to encapsulate logic in your business domain.

```
        return View(recipes); } } You can pass a view model to  
the partial view. Default.cshtml  
is used by default.
```

This custom view component handles all the logic you need to render a list of recipes when the user is logged in, or a different view if the user isn't authenticated. The name of the view component is derived from the class name, like Tag Helpers. Alternatively, you can apply the `[ViewComponent]` attribute to the class and set a different name entirely.

The `InvokeAsync` method must return a `Task<IViewComponentResult>`. This is similar to the way you can return `IActionResult` from an action method or a page handler, but it's more restrictive; view components *must* render some sort of content, so you can't return status codes or redirects. You'll typically use the `View()` helper method to render a partial view template (as in the previous listing) though you can also return a string directly using the `Content()` helper method, which will HTML-encode the content and render it to the page directly.

You can pass any number of parameters to the `InvokeAsync` method. The name of the parameters (in this case, `numberOfRecipes`) is converted to kebab-case and exposed as a property in the view component's Tag Helper (`<number-of-recipes>`). You can provide these parameters when you invoke the view component from a view, and you'll get IntelliSense support, as shown in figure 20.5.



**Figure 20.5** Visual Studio provides IntelliSense support for the method parameters of a view component's `InvokeAsync` method. The parameter name, in this case `numberOfRecipes`, is converted to kebab-case for use as an attribute in the Tag Helper.

View components have access to the current request and `HttpContext`. In listing 20.5 you can see that we're checking whether the current request was from an authenticated user. You can also see that we've used some conditional logic: if the user isn't authenticated, we render the "Unauthenticated" Razor template; if they're authenticated, we render the default Razor template and pass in the view models loaded from the database.

**NOTE** If you don't specify a specific Razor view template to use in the `View()` function, view components use the template name "Default.cshtml".

The partial views for view components work similarly to other Razor partial views that you learned about in chapter 7, but they’re stored separately from them. You must create partial views for view components at one of these locations:

- Views/Shared/Components/ComponentName/TemplateName
- Pages/Shared/Components/ComponentName/TemplateName

Both locations work, so for Razor Pages apps I typically use the Pages/ folder. For the view component in listing 20.5, for example, you’d create your view templates at

- Pages/Shared/Components/MyRecipes/Default.cshtml
- Pages/Shared/Components/MyRecipes/Unauthenticated.cshtml

This was a quick introduction to view components, but it should get you a long way. View components are a simple way to embed pockets of isolated, complex logic in your Razor layouts. Having said that, be mindful of these caveats:

- View component classes must be public, non-nested, and non-abstract classes.
- Although they’re similar to MVC controllers, you can’t use filters with view components.
- You can use layouts in your view components’ views to extract rendering logic common to a specific view component. This layout may contain @sections, as you saw in chapter 7, but these sections are independent of the main Razor view’s layout.
- View components are isolated from the Razor Page they’re rendered in, so you can’t, for example, define a @section in a Razor Page layout and then add that content from a view component; the contexts are completely separate.
- When using the <vc:my-recipes> Tag Helper syntax to invoke your view component, you must import it as a custom Tag Helper, as you saw in section 20.1.
- Instead of using the Tag Helper syntax, you may invoke the view component from a view directly by using `IViewComponentHelper` Component, though I don’t recommend using this syntax. For example,

```
@await Component.InvokeAsync("MyRecipes", new { numberOfRecipes = 3 })
```

We’ve covered Tag Helpers and view components, which are both features of the Razor engine in ASP.NET Core. In the next section you’ll learn about a different, but related, topic: how to create a custom `DataAnnotations` attribute. If you’ve used previous versions of ASP.NET, this will be familiar, but ASP.NET Core has a couple of tricks up its sleeve to help you out.

## 20.3 **Building a custom validation attribute**

In this section you’ll learn how to create a custom `DataAnnotations` validation attribute that specifies specific values a `string` property may take. You’ll then learn how you can expand the functionality to be more generic by delegating to a separate service

that is configured in your DI controller. This will allow you to create custom domain-specific validations for your apps.

We looked at model binding in chapter 6, where you saw how to use the built-in `DataAnnotations` attributes in your binding models to validate user input. These provide several built-in validations, such as

- `[Required]`—The property isn't optional and must be provided.
- `[StringLength(min, max)]`—The length of the string value must be between `min` and `max` characters.
- `[EmailAddress]`—The value must have a valid email address format.

But what if these attributes don't meet your requirements? Consider the following listing, which shows a binding model from a currency converter application. The model contains three properties: the currency to convert from, the currency to convert to, and the quantity.

#### Listing 20.6 Currency converter initial binding model

```
public class CurrencyConverterModel
{
    ➤ [Required]
    [StringLength(3, MinimumLength = 3)] ←
    public string CurrencyFrom { get; set; }

    ➤ [Required]
    [StringLength(3, MinimumLength = 3)] ←
    public string CurrencyTo { get; set; }

    ➤ [Required]
    [Range(1, 1000)] ←
    public decimal Quantity { get; set; }
}
```

**All the properties are required.**

The strings must be exactly 3 characters.

The quantity can be between 1 and 1000.

There's some basic validation on this model, but during testing you identify a problem: users can enter any three-letter string for the `CurrencyFrom` and `CurrencyTo` properties. Users should only be able to choose a valid currency code, like "USD" or "GBP", but someone attacking your application could easily send "XXX" or "£\$%".

Assuming you support a limited set of currencies, say GBP, USD, EUR, and CAD, you could handle the validation in a few different ways. One way would be to validate the `CurrencyFrom` and `CurrencyTo` values within the Razor Page handler method, after model binding and attribute validation has already occurred.

Another way would be to use a `[RegularExpression]` attribute to look for the allowed strings. The approach I'm going to take here is to create a custom Validation Attribute. The goal is to have a custom validation attribute you can apply to the `CurrencyFrom` and `CurrencyTo` attributes, to restrict the range of valid values. This will look something like the following example.

### Listing 20.7 Applying custom validation attributes to the binding model

```
public class CurrencyConverterModel
{
    [Required]
    [StringLength(3, MinimumLength = 3)]
    [CurrencyCode("GBP", "USD", "CAD", "EUR")]
    public string CurrencyFrom { get; set; }

    [Required]
    [StringLength(3, MinimumLength = 3)]
    [CurrencyCode("GBP", "USD", "CAD", "EUR")]
    public string CurrencyTo { get; set; }

    [Required]
    [Range(1, 1000)]
    public decimal Quantity { get; set; }
}
```

**CurrencyCodeAttribute** validates that the property has one of the provided values.

Creating a custom validation attribute is simple; you can start with the ValidationAttribute base class, and you only have to override a single method. The next listing shows how you could implement CurrencyCodeAttribute to ensure that the currency codes provided match the expected values.

### Listing 20.8 Custom validation attribute for currency codes

**Derives from ValidationAttribute to ensure your attribute is used during validation**

```
public class CurrencyCodeAttribute : ValidationAttribute
{
    private readonly string[] _allowedCodes;
    public CurrencyCodeAttribute(params string[] allowedCodes)
    {
        _allowedCodes = allowedCodes;
    }

    protected override ValidationResult IsValid(
        object value, ValidationContext context)
    {
        var code = value as string;
        if(code == null || !_allowedCodes.Contains(code))
        {
            return new ValidationResult("Not a valid currency code");
        }
        return ValidationResult.Success;
    }
}
```

**If the value provided isn't a string, is null, or isn't an allowed code, then return an error.**

**The attribute takes in an array of allowed currency codes.**

**The IsValid method is passed the value to validate and a context object.**

**Otherwise, return a success result.**

Validation occurs in the action filter pipeline after model binding, before the action or Razor Page handler is executed. The validation framework calls `IsValid()` for each instance of `ValidationAttribute` on the model property being validated. The framework passes in `value` (the value of the property being validated) and the

ValidationContext to each attribute in turn. The context object contains details that you can use to validate the property.

Of particular note is the ObjectInstance property. You can use this to access the top-level model being validated when you validate a sub-property. For example, if the CurrencyFrom property of the CurrencyConvertModel is being validated, you can access the top-level object from the ValidationAttribute as follows:

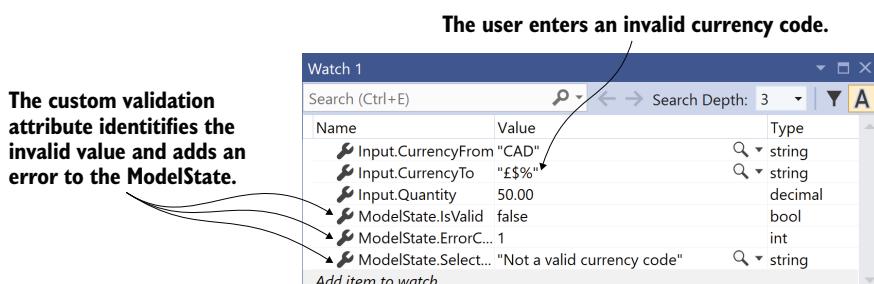
```
var model = validationContext.ObjectInstance as CurrencyConverterModel;
```

This can be useful if the validity of a property depends on the value of another property of the model. For example, you might want a validation rule that says that GBP is a valid value for CurrencyTo, *except* when CurrencyFrom is *also* GBP. The ObjectInstance makes these sorts of comparison validations easy.

**NOTE** Although using ObjectInstance makes it easy to make model-level comparisons like these, it reduces the portability of your validation attribute. In this case, you would only be able to use the attribute in the application that defines CurrencyConverterModel.

Within the IsValid method, you can cast the value provided to the required data type (in this case, string) and check against the list of allowed codes. If the code isn't allowed, the attribute returns a ValidationResult with an error message indicating that there was a problem. If the code is allowed, ValidationResult.Success is returned, and the validation succeeds.

Putting your attribute to the test in figure 20.6 shows that when CurrencyTo is an invalid value (€\$%), the validation for the property fails and an error is added to the ModelState. You could do some tidying up of this attribute to let you set a custom message, to allow nulls, or to display the name of the property that's invalid, but the important features are all there.



**Figure 20.6** The Watch window of Visual Studio showing the result of validation using the custom ValidationAttribute. The user has provided an invalid currencyTo value, €\$%. Consequently, ModelState isn't valid and contains a single error with the message "Not a valid currency code."

The main feature missing from this custom attribute is client-side validation. You've seen that the attribute works well on the server side, but if the user entered an invalid value, they wouldn't be informed until after the invalid value had been sent to the server. That's safe, and it's as much as you *need* to do for security and data-consistency purposes, but client-side validation can improve the user experience by providing immediate feedback.

You can implement client-side validation in several ways, but it's heavily dependent on the JavaScript libraries you use to provide the functionality. Currently ASP.NET Core Razor templates rely on jQuery for client-side validation. See the "Custom client-side validation" section of Microsoft's "Model validation in ASP.NET Core MVC and Razor Pages" documentation for an example of creating a jQuery Validation adapter for your attributes: <http://mng.bz/Wd6g>.

Another improvement to your custom validation attribute would be to load the list of currencies from a DI service, such as an `ICurrencyProvider`. Unfortunately, you can't use constructor DI in your `CurrencyCodeAttribute` as you can only pass *constant* values to the constructor of an Attribute in .NET. In chapter 13 we worked around this limitation for filters by using `[TypeFilter]` or `[ServiceFilter]`, but there's no such solution for `ValidationAttribute`.

Instead, for validation attributes, you must use the Service Locator pattern. As I discussed in chapter 10, this antipattern is best avoided where possible, but unfortunately it's necessary in this case. Instead of declaring an explicit dependency via a constructor, you must ask the DI container directly for an instance of the required service.

Listing 20.9 shows how you could rewrite listing 20.8 to load the allowed currencies from an instance of `ICurrencyProvider`, instead of hardcoding the allowed values in the attribute's constructor. The attribute calls the `GetService<T>()` method on `ValidationContext` to resolve an instance of `ICurrencyProvider` from the DI container. Note that `ICurrencyProvider` is a hypothetical service that would need to be registered in your application's `ConfigureServices()` method in `Startup.cs`.

#### Listing 20.9 Using the service-locator pattern to access services

```
public class CurrencyCodeAttribute : ValidationAttribute
{
    protected override ValidationResult IsValid(
        object value, ValidationContext context)
    {
        var provider = context.GetService<ICurrencyProvider>();
        var allowedCodes = provider.GetCurrencies();

        var code = value as string;
        if(code == null || !_allowedCodes.Contains(code))
        {
            return new ValidationResult("Not a valid currency code");
        }
        return ValidationResult.Success;
    }
}
```

**Validates the property as before**

**Retrieves an instance of `ICurrencyProvider` directly from the DI container**

**Fetches the currency codes using the provider**

**TIP** The generic `GetService<T>` method is an extension method available in the `Microsoft.Extensions.DependencyInjection` namespace. As an alternative, you can use the `GetService(Type type)` method.

The default `DataAnnotations` validation system can be convenient due to its declarative nature, but this has trade-offs, as shown by the dependency injection problem above. Luckily, you can completely replace the validation system your application uses, as shown in the following section.

## 20.4 Replacing the validation framework with FluentValidation

In this section you'll learn how to replace the `DataAnnotations`-based validation framework that's used by default in ASP.NET Core. You'll see the arguments for why you might want to do this and learn how to use a third-party alternative: `FluentValidation`. This open source project allows you to define the validation requirements of your models separately from the models themselves. This separation can make some types of validation easier and ensures each class in your application has a single responsibility.

Validation is an important part of the model-binding process in ASP.NET Core. Up to now, we've been using `DataAnnotations` attributes applied to properties of your binding model to define your requirements. In section 20.3 we even created a custom validation attribute.

By default, ASP.NET Core is configured to use these attributes to drive the validation portion of model binding. But the ASP.NET Core framework is very flexible and allows you to replace whole chunks of the framework if you like. The validation system is one such area that many people choose to replace.

`FluentValidation` (<https://fluentvalidation.net/>) is a popular alternative validation framework for ASP.NET Core. It is a mature library, with roots going back well before ASP.NET Core was conceived of. With `FluentValidation` you write your validation code *separately* from your binding model code. This gives several advantages:

- You're not restricted to the limitations of `Attributes`, such as the dependency injection problem we had to work around in listing 20.9.
- It's much easier to create validation rules that apply to multiple properties, such as to ensure an `EndDate` property contains a later value than a `StartDate` property. Achieving this with `DataAnnotations` attributes is possible, but difficult.
- It's generally easier to test `FluentValidation` validators than `DataAnnotations` attributes.
- The validation is strongly typed, compared to `DataAnnotations` attributes where it's possible to apply attributes in ways that don't make sense, such as applying an `[EmailAddress]` attribute to an `int` property, for example.
- Separating your validation logic from the model itself arguably better conforms to the single-responsibility-principle (SRP).<sup>3</sup>

---

<sup>3</sup> The SRP is one of the SOLID design principles: <https://en.wikipedia.org/wiki/SOLID>.

That final point is often given as a reason *not* to use FluentValidation: FluentValidation separates a binding model from its validation rules. Some people are happy to accept the limitations of DataAnnotations to keep the model and validation rules together.

Before I show you how to add FluentValidation to your application, let's see what FluentValidation validators look like.

#### 20.4.1 Comparing FluentValidation to DataAnnotations attributes

To better understand the difference between the DataAnnotations approach and FluentValidation, we'll convert the binding models from section 20.3 to use FluentValidation. The following listing shows what the binding model from listing 20.7 would look like when used with FluentValidation. It is structurally identical but has no validation attributes.

**Listing 20.10 Currency converter initial binding model for use with FluentValidation**

```
public class CurrencyConverterModel
{
    public string CurrencyFrom { get; set; }
    public string CurrencyTo { get; set; }
    public decimal Quantity { get; set; }
}
```

In FluentValidation you define your validation rules in a separate class, with a class per model to be validated. Typically, these derive from the `AbstractValidator<>` base class, which provides a set of extension methods for defining your validation rules.

The following listing shows a validator for the `CurrencyConverterModel`, which matches the validations added using attributes in listing 20.7. You create a set of validation rules for a property by calling `RuleFor()` and chaining method calls such as `NotEmpty()` from it. This style of method chaining is called a “fluent” interface, hence the name.

**Listing 20.11 A FluentValidation validator for the currency converter binding model**

```
public class CurrencyConverterModelValidator
    : AbstractValidator<CurrencyConverterModel>
{
    private readonly string[] _allowedValues
        = new []{ "GBP", "USD", "CAD", "EUR" };

    public InputValidator()
    {
        RuleFor(x => x.CurrencyFrom)
            .NotEmpty()
            .Length(3)
            .Must(value => _allowedValues.Contains(value))
            .WithMessage("Not a valid currency code");

        RuleFor(x => x.CurrencyTo)
            .NotEmpty();
    }
}
```

You define validation rules in the validator's constructor.

RuleFor is used to add a new validation rule. The lambda syntax allows for strong typing.

The validator inherits from `AbstractValidator`.

Defines the static list of currency codes that are supported

There are equivalent rules for common DataAnnotations validation attributes.

You can easily add custom validation rules without having to create separate classes.

```

    .Length(3)
    .Must(value => _allowedValues.Contains(value))
    .WithMessage("Not a valid currency code");

    RuleFor(x => x.Quantity)
        .NotNull()
        .InclusiveBetween(1, 1000);
}

}

```

Thanks to strong typing, the rules available depend on the property being validated.

Your first impression of this code might be that it's quite verbose compared to listing 20.7 but remember that listing 20.7 used a custom validation attribute, [CurrencyCode]. The validation in listing 20.11 doesn't require anything else—the logic implemented by the [CurrencyCode] attribute is right there in the validator, making it easy to reason about. The Must() method can be used to perform arbitrarily complex validations, without having the additional layers of indirection required by custom DataAnnotations attributes.

On top of that, you'll notice that you can only define validation rules that make sense for the property being validated. Previously, there was nothing to stop us applying the [CurrencyCode] attribute to the Quantity property; that's just not possible with FluentValidation.

Of course, just because you *can* write the custom [CurrencyCode] logic in-line doesn't necessarily mean you have to. If a rule is used in multiple parts of your application, it may make sense to extract it into a helper class. The following listing shows how you could extract the currency code logic into an extension method, which can be used in multiple validators.

#### Listing 20.12 An extension method for currency validation

```

public static class ValidationExtensions
{
    public static IRuleBuilderOptions<T, string>
        MustBeCurrencyCode<T>(
            this IRuleBuilder<T, string> ruleBuilder)
    {
        return ruleBuilder
            .Must(value => _allowedValues.Contains(value))
            .WithMessage("Not a valid currency code");
    }

    private static readonly string[] _allowedValues =
        new []{ "GBP", "USD", "CAD", "EUR" };
}

```

Creates an extension method that can be chained from RuleFor() for string properties

Applies the same validation logic as before

The currency code values to allow

You can then update your CurrencyConverterModelValidator to use the new extension method, removing the duplication in your validator and ensuring consistency across country-code fields:

```
RuleFor(x => x.CurrencyTo)
    .NotEmpty()
    .Length(3)
    .MustBeCurrencyCode();
```

Another advantage of the FluentValidation approach of using standalone validation classes is that they are created using dependency injection, so you can inject services into them. As an example, consider the [CurrencyCode] validation attribute from listing 20.9, which used a service, `ICurrencyProvider`, from the DI container. This requires using service location to obtain an instance of `ICurrencyProvider` using an injected context object.

With the FluentValidation library, you can just inject the `ICurrencyProvider` directly into your validator, as shown in the following listing. This requires fewer gymnastics to get the desired functionality and makes your validator's dependencies explicit.

#### **Listing 20.13 Currency converter validator using dependency injection**

```
public class CurrencyConverterModelValidator
    : AbstractValidator<CurrencyConverterModel>
{
    public CurrencyConverterModelValidator(ICurrencyProvider provider) ←
    {
        RuleFor(x => x.CurrencyFrom)
            .NotEmpty()
            .Length(3)
            .Must(value => provider
                .GetCurrencies()
                .Contains(value)) ←
            .WithMessage("Not a valid currency code");

        RuleFor(x => x.CurrencyTo)
            .NotEmpty()
            .Length(3)
            .MustBeCurrencyCode(provider.GetCurrencies()); ←
            .WithMessage("Not a valid currency code");

        RuleFor(x => x.Quantity)
            .NotNull()
            .InclusiveBetween(1, 1000);
    }
}
```

**Injecting the service using standard constructor dependency injection**

**Using the injected service in a Must() rule**

**Using the injected service with an extension method**

The final feature I'll show demonstrates how much easier it is to write validators that span multiple properties with FluentValidation. For example, imagine we want to validate that the value of `CurrencyTo` is different than `CurrencyFrom`. Using FluentValidation you can implement this with an overload of `Must()`, which provides both the model and the property being validated, as shown in the following listing.

**Listing 20.14 Using Must() to validate that two properties are different**

```

RuleFor(x => x.CurrencyTo)
    .NotEmpty()
    .Length(3)
    .MustBeCurrencyCode()
    .Must((InputModel model, string currencyTo) 
        => currencyTo != model.CurrencyFrom)
    .WithMessage("Cannot convert currency to itself");

```

The error message will be associated with the `CurrencyTo` property.

The Must function passes the top-level model being validated and the current property.

Perform the validation—the currencies must be different.

Use the provided message as the error message.

Creating a validator like this is certainly possible with `DataAnnotations` attributes, but it requires far more ceremony than the `FluentValidation` equivalent and is generally harder to test. `FluentValidation` has many more features for making it easier to write and test your validators too:

- *Complex property validations*—Validators can be applied to complex types, as well as to the primitive types like `string` and `int` shown here in this section.
- *Custom property validators*—In addition to simple extension methods, you can create your own property validators for complex validation scenarios.
- *Collection rules*—When types contain collections, such as `List<T>`, you can apply validation to each item in the list, as well as to the overall collection.
- *RuleSets*—You can create multiple collections of rules that can be applied to an object in different circumstances. These can be especially useful if you’re using `FluentValidation` in additional areas of your application.
- *Client-side validation*—`FluentValidation` is a server-side framework, but it emits the same attributes as `DataAnnotations` attributes to enable client-side validation using `jQuery`.

There are many more features in addition to these, so be sure to browse the documentation at <https://docs.fluentvalidation.net/> for details. In the next section you’ll see how to add `FluentValidation` to your ASP.NET Core application.

## 20.4.2 Adding FluentValidation to your application

Replacing the whole validation system of ASP.NET Core sounds like a big step, but the `FluentValidation` library makes it easy to add to your application. Simply follow these steps:

- 1 Install the `FluentValidation.AspNetCore` NuGet package using Visual Studio’s NuGet package manager via the CLI by running `dotnet add package FluentValidation.AspNetCore` or by adding a `<PackageReference>` to your `.csproj` file:

```
<PackageReference Include="FluentValidation.AspNetCore" Version="9.3.0" />
```

- 2 Configure the `FluentValidation` library in the `ConfigureServices` method of your `Startup` class by calling `AddFluentValidation()`. You can further configure the library as shown in listing 20.15.

- 3 Register your validators (such as the `CurrencyConverterModelValidator` from listing 20.13) with the DI container. These can be registered manually, using any scope you choose:

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

    services.AddScoped<
        IValidator<CurrencyConverterModelValidator>,
        CurrencyConverterModelValidator>();
}
```

Alternatively, you can allow FluentValidation to automatically register all your validators using the options shown in listing 20.15.

For such a mature library, FluentValidation has relatively few configuration options to decipher. The following listing shows some of the options available; in particular, it shows how to automatically register all the custom validators in your application and how to disable `DataAnnotations` validation entirely.

#### **Listing 20.15 Configuring FluentValidation in an ASP.NET Core application**

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddRazorPages()
        .AddFluentValidation(opts =>
    {
        opts.RegisterValidatorsFromAssemblyContaining<Startup>();           ←
        opts.ImplicitlyValidateChildProperties = true;                         ←
        opts.LocalizationEnabled = false;                                       ←
        opts.RunDefaultMvcValidationAfterFluentValidationExecutes
            = false;                                                 ←
    });
}
```

**Ensure that complex (nested)  
properties are validated, not  
just top-level properties.**

**Instead of manually registering  
validators, FluentValidation can  
auto-register them for you.**

**Setting to false disables DataAnnotations  
validation completely for model binding.**

**FluentValidation has full localization support,  
but you can disable it if you don't need it.**

It's important to understand that final point. If you don't set `RunDefaultMvcValidationAfterFluentValidationExecutes` to `false`, ASP.NET Core will run validation with *both* `DataAnnotations` and with `FluentValidation`. That may be useful if you're in the process of migrating from one system to the other, but otherwise I recommend disabling it. Having your validation split between both places seems like it would be the worst of both worlds!

One final thing to consider is where to put your validators in your solution. There are no technical requirements on this—if you've registered your validator with the DI container, it will be used correctly—so the choice is up to you. Personally, I prefer to place validators close to the models they're validating.

For Razor Pages binding-model validators, I create the validator as a nested class of the `PageModel`, in the same place as I create the `InputModel`, as I described in chapter 6. That gives a class hierarchy in the Razor Page similar to the following:

```
public class IndexPage : PageModel
{
    public class InputModel { }
    public class InputModelValidator: AbstractValidator<InputModel> { }
}
```

That's just my preference of course. You're free to adopt another approach if you prefer.

That brings us to the end of this chapter on custom Razor Pages components. When combined with the components in the previous chapter, you've got a great base for extending your ASP.NET Core applications to meet your needs. It's a testament to ASP.NET Core's design that you can swap out whole sections like the Validation framework entirely. If you don't like how some part of the framework works, see if someone has written an alternative!

## Summary

- With Tag Helpers, you can bind your data model to HTML elements, making it easier to generate dynamic HTML. 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.
- The name of a Tag Helper class dictates the name of the element in the Razor templates, so the `SystemInfoTagHelper` corresponds to the `<system-info>` element. You can choose a different element name by adding the `[HtmlTargetElement]` attribute to your Tag Helper.
- You can set properties on your Tag Helper object from Razor syntax by decorating the property with an `[HtmlAttributeName("name")]` attribute and providing a name. You can set these properties from Razor using HTML attributes; `<system-info name="value">`, for example.
- The `TagHelperOutput` parameter passed to the `Process` or `ProcessAsync` methods control the HTML that's rendered to the page. You can set the element type with the `TagName` property and set the inner content using `Content.SetContent()` or `Content.SetHtmlContent()`.
- You can prevent inner Tag Helper content being processed by calling `SuppressOutput()`, and you can remove the element entirely by setting `TagName=null`. This is useful if you only want to conditionally render elements to the response.
- You can retrieve the contents of a Tag Helper by calling `GetChildContentAsync()` on the `TagHelperOutput` parameter. You can then render this content to a string by calling `GetContent()`. This will render any Razor expressions and Tag Helpers to HTML, allowing you to manipulate the contents.

- View components are like partial views, but they allow you to use complex business and rendering logic. You can use them for sections of a page, such as the shopping cart, a dynamic navigation menu, or suggested articles.
- Create a view component by deriving from the `ViewComponent` base class and implementing `InvokeAsync()`. You can pass parameters to this function from the Razor view template using HTML attributes, in a similar way to Tag Helpers.
- View components can use DI, access the `HttpContext`, and render partial views. The partial views should be stored in the `Pages/Shared/Components/<Name>/` folder, where `Name` is the name of the view component. If not specified, view components will look for a default view named `Default.cshtml`.
- You can create a custom `DataAnnotations` attribute by deriving from `ValidationAttribute` and overriding the `IsValid` method. You can use this to decorate your binding model properties and perform arbitrary validation.
- You can't use constructor DI with custom validation attributes. If the validation attribute needs access to services from the DI container, you must use the Service Locator pattern to load them from the validation context, using the `GetService<T>` method.
- FluentValidation is an alternative validation system that can replace the default `DataAnnotations` validation system. It is not based on attributes, which makes it easier to write custom validations for your validation rules and makes those rules easier to test.
- To create a validator for a model, create a class derived from `AbstractValidator<>` and call `RuleFor<>()` in the constructor to add validation rules. You can chain multiple requirements on `RuleFor<>()` in the same way that you could add multiple `DataAnnotations` attributes to a model.
- If you need to create a custom validation rule, you can use the `Must()` method to specify a predicate. If you wish to re-use the validation rule across multiple models, encapsulate the rule as an extension method, to reduce duplication.
- To add FluentValidation to your application, install the `FluentValidation.AspNetCore` NuGet package, call `AddFluentValidation()` after your call to `AddRazorPages()` or `AddControllers()`, and register your validators with the DI container. This will add FluentValidation validations in addition to the built-in `DataAnnotations` system.
- To remove the `DataAnnotations` validation system and use FluentValidation only, set the `RunDefaultMvcValidationAfterFluentValidationExecutes` option to `false` in your call to `AddFluentValidation()`. Favor this approach where possible, to avoid running validation methods from two different systems.
- You can allow FluentValidation to automatically discover and register all the validators in your application by calling `RegisterValidatorsFromAssemblyContaining<T>()`, where `T` is a type in the assembly to scan. This means you don't have to register each validator in your application with the DI container individually.

# 21

## *Calling remote APIs with `IHttpClientFactory`*

### **This chapter covers**

- Problems caused by using `HttpClient` incorrectly to call HTTP APIs
- Using `IHttpClientFactory` to manage `HttpClient` lifetimes
- Encapsulating configuration and handling transient errors with `IHttpClientFactory`

So far in this book we've focused on creating web pages and exposing APIs. Whether that's customers browsing a Razor Pages application or client-side SPAs and mobile apps consuming your APIs, we've been writing the APIs for others to consume.

However, it's very common for your application to interact with third-party services by consuming *their* APIs. For example, an e-commerce site needs to take payments, send email and SMS messages, and retrieve exchange rates from a third-party service. The most common approach for interacting with services is using HTTP. So far in this book we've looked at how you can *expose* HTTP services, using API controllers, but we haven't looked at how you can *consume* HTTP services.

In section 21.1 you'll learn the best way to interact with HTTP services using `HttpClient`. If you have any experience with C#, it's very likely you've used this

class to send HTTP requests, but there are two gotchas to think about; otherwise your app could run into difficulties.

`IHttpClientFactory` was introduced in .NET Core 2.1; it makes creating and managing `HttpClient` instances easier and avoids the common pitfalls. In section 21.2 you'll learn how `IHttpClientFactory` achieves this by managing the `HttpClient` handler pipeline. You'll learn how to create *named clients* to centralize the configuration for calling remote APIs and how to use *typed clients* to encapsulate the remote service's behavior.

Network glitches are a fact of life when you're working with HTTP APIs, so it's important for you to handle them gracefully. In section 21.3 you'll learn how to use the open source resilience and fault-tolerance library Polly to handle common transient errors using simple retries, with the possibility for more complex policies.

Finally, in section 21.4 you'll see how you can create your own custom `HttpMessageHandler` handler managed by `IHttpClientFactory`. You can use custom handlers to implement cross-cutting concerns such as logging, metrics, or authentication, where a function needs to execute every time you call an HTTP API. You'll also see how to create a handler that automatically adds an API key to all outgoing requests to an API.

To misquote John Donne, “no app is an island,” and the most common way of interacting with other apps and services is over HTTP. In .NET, that means using `HttpClient`.

## 21.1 Calling HTTP APIs: The problem with `HttpClient`

In this section you'll learn how to use `HttpClient` to call HTTP APIs. I'll focus on two common pitfalls in using `HttpClient`—socket exhaustion and DNS rotation problems—and show why they occur. In section 21.2 you'll see how to avoid these issues by using `IHttpClientFactory`.

It's very common for an application to interact with other services to fulfill its duty. Take a typical e-commerce store, for example. In even the most basic version of the application, you will likely need to send emails and take payments using credit cards or other services. You *could* try to build that functionality yourself, but it probably wouldn't be worth the effort.

Instead, it makes far more sense to delegate those responsibilities to third-party services that specialize in that functionality. Whichever service you use, they will almost certainly expose an HTTP API for interacting with the service. For many services, that will be the *only* way.

### RESTful HTTP vs. gRPC vs. GraphQL

There are many ways to interact with third-party services, but HTTP RESTful services are still the king, decades after HTTP was first proposed. Every platform and programming language you can think of includes support for making HTTP requests and handling responses. That ubiquity makes it the go-to option for most services.

Despite their ubiquity, RESTful services are not perfect. They are relatively verbose, which means more data ends up being sent and received than with some other protocols. It can also be difficult to evolve RESTful APIs after you have deployed them. These limitations have spurred interest in two alternative protocols in particular: gRPC and GraphQL.

gRPC is intended to be an efficient mechanism for server-to-server communication. It builds on top of HTTP/2 but typically provides much higher performance than traditional RESTful APIs. gRPC support was added in .NET Core 3.0 and is receiving many performance and feature updates. For a comprehensive view of .NET support, see the documentation at <https://docs.microsoft.com/aspnet/core/grpc>.

While gRPC is primarily intended for server-to-server communication, GraphQL is best used to provide evolvable APIs to mobile and SPA apps. It has become very popular among frontend developers, as it can reduce the friction involved in deploying and using new APIs. For details, I recommend *GraphQL in Action* by Samer Buna (Manning, 2021).

Despite the benefits and improvements gRPC and GraphQL can bring, RESTful HTTP services are here to stay for the foreseeable future, so it's worth making sure you understand how to use them with `HttpClient`.

In .NET we use the `HttpClient` class for calling HTTP APIs. You can use it to make HTTP calls to APIs, providing all the headers and body to send in a request, and reading the response headers and data you get back. Unfortunately, it's hard to use correctly, and even when you do, it has limitations.

The source of the difficulty with `HttpClient` stems partly from the fact that it implements the `IDisposable` interface. In general, when you use a class that implements `IDisposable`, you should wrap the class with a `using` statement whenever you create a new instance. This ensures that unmanaged resources used by the type are cleaned up when the class is removed.

```
using (var myInstance = new MyDisposableClass())
{
    // use myInstance
}
```

That might lead you to think that the correct way to create an `HttpClient` is shown in the following listing. This shows a simple example where an API controller calls an external API to fetch the latest currency exchange rates and returns them as the response.

**WARNING** Do not use `HttpClient` as it's shown in listing 21.1. Using it this way could cause your application to become unstable, as you'll see shortly.

**Listing 21.1 The incorrect way to use HttpClient**

```
[ApiController]
public class ValuesController : ControllerBase
{
    [HttpGet("values")]
    public async Task<string> GetRates()
    {
        using (HttpClient client = new HttpClient()) ←
        {
            client.BaseAddress
                = new Uri("https://api.exchangeratesapi.io"); ←
            var response = await client.GetAsync("latest"); ←

            response.EnsureSuccessStatusCode(); ←
            return await response.Content.ReadAsStringAsync(); ←
        }
    }
}
```

**Make a GET request to the exchange rates API.**

**Wrapping the HttpClient in a using statement means it is disposed of at the end of the using block.**

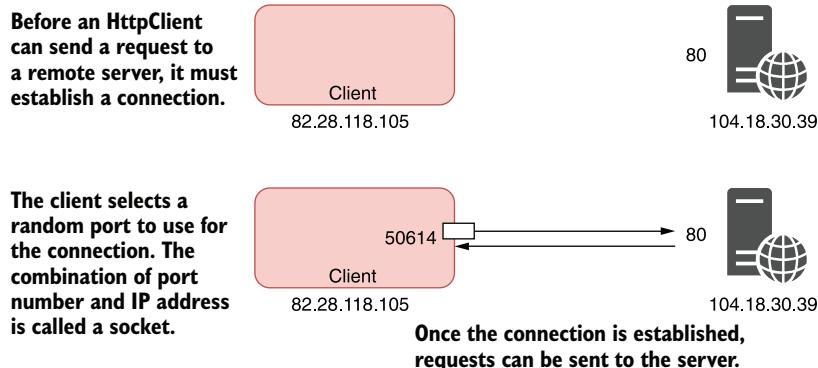
**Configure the base URL used to make requests using the HttpClient.**

**Throw an exception if the request was not successful.**

**Read the result as a string and return it from the action method.**

HttpClient is special, and you *shouldn't* use it like this! The problem is primarily due to the way the underlying protocol implementation works. Whenever your computer needs to send a request to an HTTP server, you must create a *connection* between your computer and the server. To create a connection, your computer opens a port, which has a random number between 0 and 65,535, and connects to the HTTP server's IP address and port, as shown in figure 21.1. Your computer can then send HTTP requests to the server.

**DEFINITION** The combination of IP address and port is called a *socket*.



**Figure 21.1** To create a connection, a client selects a random port and connects to the HTTP server's port and IP address. The client can then send HTTP requests to the server.

The main problem with the using statement and `HttpClient` is that it can lead to a problem called *socket exhaustion*, as illustrated in figure 21.2. This happens when all the ports on your computer have been used up making other HTTP connections, so your computer can't make any more requests. At that point, your application will hang, waiting for a socket to become free. A very bad experience!

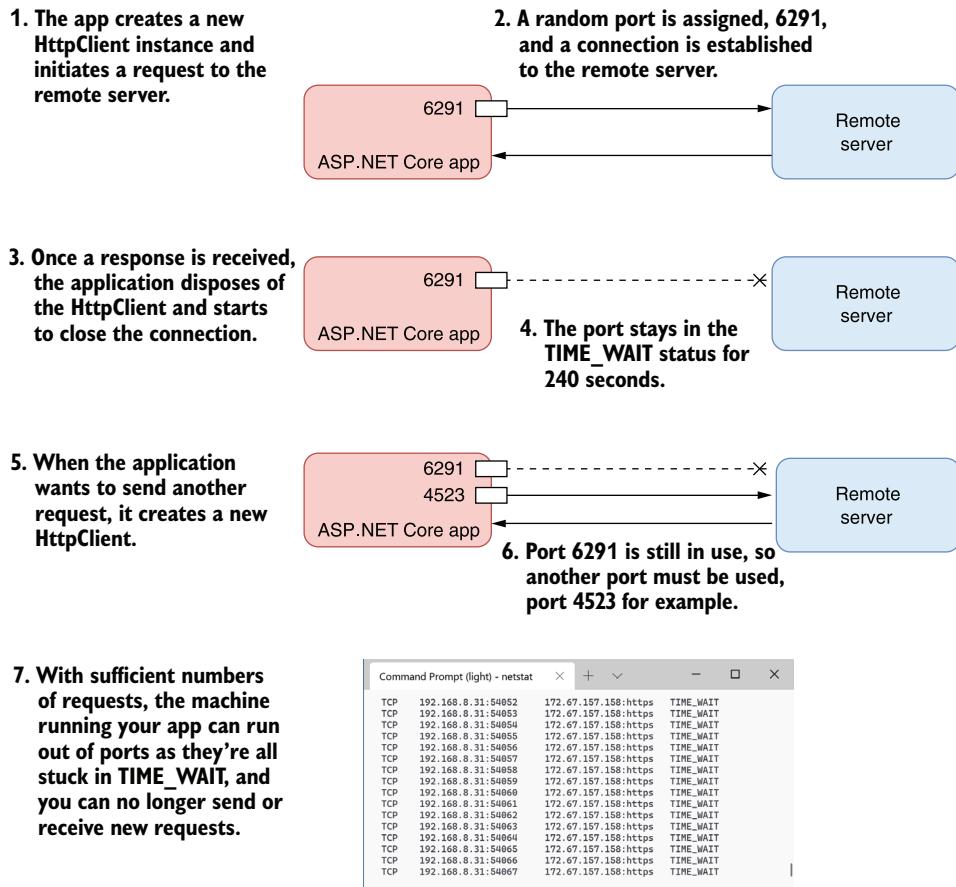


Figure 21.2 Disposing of `HttpClient` can lead to socket exhaustion. Each new connection requires the operating system to assign a new socket, and closing a socket doesn't make it available until the `TIME_WAIT` period of 240 seconds has elapsed. Eventually you can run out of sockets, at which point you can't make any outgoing HTTP requests.

Given that I said there are 65,536 different port numbers, you might think that's an unlikely situation. It's true, you will likely only run into this problem on a server that is making a lot of connections, but it's not as rare as you might think.

The problem is that when you dispose of an `HttpClient`, *it doesn't close the socket immediately*. The design of the TCP/IP protocol used for HTTP requests means that after trying to close a connection, the connection moves to a state called `TIME_WAIT`. The connection then waits for a specific period (240 seconds on Windows) before closing the socket completely.

Until the `TIME_WAIT` period has elapsed, you can't reuse the socket in another `HttpClient` to make HTTP requests. If you're making a lot of requests, that can quickly lead to socket exhaustion, as shown in figure 21.2.

**TIP** You can view the state of active ports/sockets in Windows and Linux by running the command `netstat` from the command line or a terminal window. Be sure to run `netstat -n` on Windows to skip DNS resolution.

Instead of disposing of `HttpClient`, the general advice (before `IHttpClientFactory` was introduced in .NET Core 2.1) was to use a single instance of `HttpClient`, as shown in the following listing.

### Listing 21.2 Using a singleton `HttpClient` to avoid socket exhaustion

```
[ApiController]
public class ValuesController : ControllerBase
{
    private static readonly HttpClient _client = new HttpClient
    {
        BaseAddress = new Uri("https://api.exchangeratesapi.io")
    };

    [HttpGet("values")]
    public async Task<string> GetRates()
    {
        var response = await _client.GetAsync("latest"); ←
        response.EnsureSuccessStatusCode();
        return await response.Content.ReadAsStringAsync();
    }
}
```

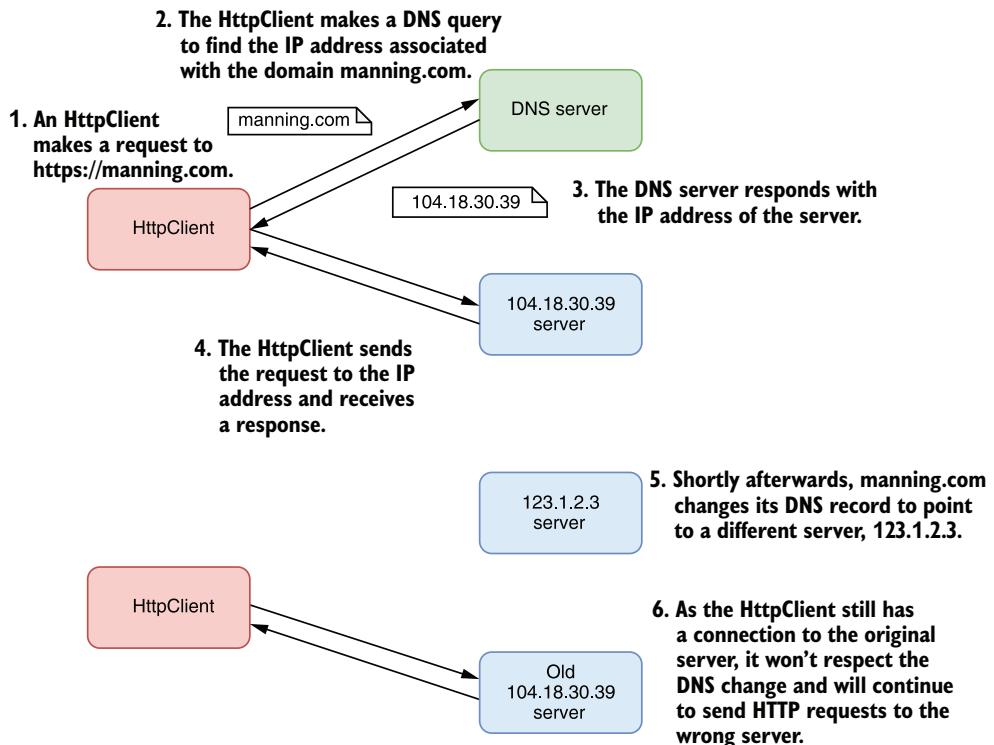
A single instance of `HttpClient` is created and stored as a static field.

Multiple requests use the same instance of `HttpClient`.

This solves the problem of socket exhaustion. As you're not disposing of the `HttpClient`, the socket is not disposed of, so you can reuse the same port for multiple requests. No matter how many times you call `GetRates()` in the preceding example, you will only use a single socket. Problem solved!

Unfortunately, this introduces a different problem, primarily around DNS. DNS is how the friendly hostnames we use, such as `manning.com`, are converted into the IP addresses that computers need. When a new connection is required, the `HttpClient` first checks the DNS record for a host to find the IP address and then makes the connection. For subsequent requests, the connection is already established, so it doesn't make another DNS call.

For singleton HttpClient instances, this can be a problem because the HttpClient won't detect DNS changes. DNS is often used in cloud environments for load balancing to do graceful rollouts of deployments.<sup>1</sup> If the DNS record of a service you're calling changes during the lifetime of your application, a singleton HttpClient will keep calling the old service, as shown in figure 21.3.



**Figure 21.3** HttpClient does a DNS lookup before establishing a connection, to determine the IP address associated with a hostname. If the DNS record for a hostname changes, a singleton HttpClient will not detect it and will continue sending requests to the original server it connected to.

**NOTE** HttpClient won't respect a DNS change while the original connection exists. If the original connection is closed (for example, if the original server goes offline), it will respect the DNS change as it must establish a new connection.

<sup>1</sup> Azure Traffic Manager, for example, uses DNS to route requests. You can read more about how it works at <https://azure.microsoft.com/en-gb/services/traffic-manager/>.

It seems like you're damned if you, and you're damned if you don't! Luckily, `IHttpClientFactory` can take care of all this for you.

## 21.2 Creating `HttpClients` with `IHttpClientFactory`

In this section you'll learn how you can use `IHttpClientFactory` to avoid the common pitfalls of `HttpClient`. I'll show several patterns you can use to create `HttpClients`:

- Using `CreateClient()` as a drop-in replacement for `HttpClient`
- Using *named clients* to centralize the configuration of an `HttpClient` used to call a specific third-party API
- Using *typed clients* to encapsulate the interaction with a third-party API for easier consumption by your code

`IHttpClientFactory` was introduced in .NET Core 2.1. It makes it easier to create `HttpClient` instances *correctly*, instead of relying on either of the faulty approaches I discussed in section 21.1. It also makes it easier to configure multiple `HttpClients` and allows you to create a middleware pipeline for outgoing requests.

Before we look at how `IHttpClientFactory` achieves all that, we will look a little closer at how `HttpClient` works under the hood.

### 21.2.1 Using `IHttpClientFactory` to manage `HttpClientHandler` lifetime

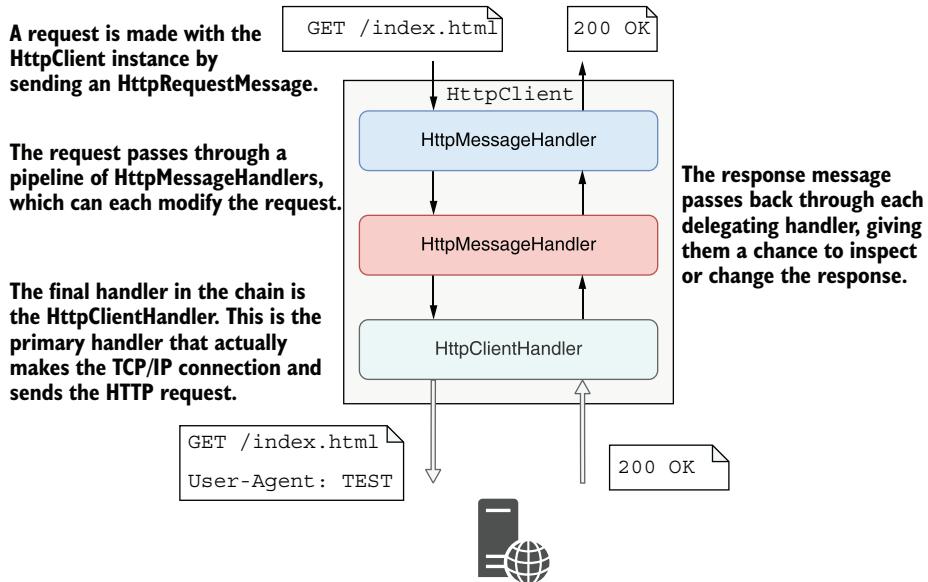
In this section we'll look at the handler pipeline used by `HttpClient`. You'll see how `IHttpClientFactory` manages the lifetime of this pipeline and how this enables the factory to avoid both socket exhaustion and DNS issues.

The `HttpClient` class you typically use to make HTTP requests is responsible for orchestrating requests, but it isn't responsible for making the raw connection itself. Instead, the `HttpClient` calls into a pipeline of `HttpMessageHandler`, at the end of which is an `HttpClientHandler`, which makes the actual connection and sends the HTTP request, as shown in figure 21.4.

This configuration is very reminiscent of the middleware pipeline used by ASP.NET Core applications, but this is an *outbound* pipeline. When an `HttpClient` makes a request, each handler gets a chance to modify the request before the final `HttpClientHandler` makes the real HTTP request. Each handler, in turn, then gets a chance to view the response after it's received.

**TIP** You'll see an example of using this handler pipeline for cross-cutting concerns in section 21.2.4 when we add a transient error handler.

The issues of socket exhaustion and DNS I described in section 21.1 are both related to the disposal of the `HttpClientHandler` at the end of the handler pipeline. By default, when you dispose of an `HttpClient`, you dispose of the handler pipeline too. `IHttpClientFactory` separates the lifetime of the `HttpClient` from the underlying `HttpClientHandler`.



**Figure 21.4** Each `HttpClient` contains a pipeline of `HttpMessageHandlers`. The final handler is an `HttpClientHandler`, which makes the connection to the remote server and sends the HTTP request. This configuration is similar to the ASP.NET Core middleware pipeline, and it allows you to make cross-cutting adjustments to outgoing requests.

Separating the lifetime of these two components enables the `IHttpClientFactory` to solve the problems of socket exhaustion and DNS rotation. It achieves this in two ways:

- *By creating a pool of available handlers*—Socket exhaustion occurs when you dispose of an `HttpClientHandler`, due to the `TIME_WAIT` problem described previously. `IHttpClientFactory` solves this by creating a *pool* of handlers.

`IHttpClientFactory` maintains an *active* handler that is used to create all `HttpClients` for two minutes. When the `HttpClient` is disposed of, the underlying handler *isn't* disposed of, so the connection isn't closed. As a result, socket exhaustion isn't a problem.

- *By periodically disposing of handlers*—Sharing handler pipelines solves the socket exhaustion problem, but it doesn't solve the DNS issue. To work around this, the `IHttpClientFactory` periodically (every two minutes) creates a new active `HttpClientHandler` that is used for each `HttpClient` created subsequently. As these `HttpClients` are using a new handler, they make a new TCP/IP connection, so DNS changes are respected.

`IHttpClientFactory` disposes of “expired” handlers periodically in the background once they are no longer used by an `HttpClient`. This ensures there

are only ever a limited number of connections in use by your application's `HttpClients`.<sup>2</sup>

Rotating handlers with `IHttpClientFactory` solves both of the issues we've discussed. Another bonus is that it's easy to replace existing uses of `HttpClient` with `IHttpClientFactory`.

`IHttpClientFactory` is included by default in ASP.NET Core; you just need to add it to your application's services in the `ConfigureServices()` method of `Startup.cs`:

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddHttpClient()
}
```

This registers the `IHttpClientFactory` as a singleton in your application, so you can inject it into any other service. For example, the following listing shows how you can replace the `HttpClient` approach from listing 21.2 with a version that uses `IHttpClientFactory`.

### Listing 21.3 Using `IHttpClientFactory` to create an `HttpClient`

```
[ApiController]
public class ValuesController : ControllerBase
{
    private readonly IHttpClientFactory _factory;
    public ValuesController(IHttpClientFactory factory)
    {
        _factory = factory;
    }

    [HttpGet("values")]
    public async Task<string> GetRates()
    {
        HttpClient client = _factory.CreateClient(); ← Create an HttpClient instance with an HttpClientHandler managed by the factory.

        client.BaseAddress =
            new Uri("https://api.exchangeratesapi.io");
        client.DefaultRequestHeaders.Add(
            HeaderNames.UserAgent, "ExchangeRateViewer"); ← Configure the HttpClient for calling the API as before.

        var response = await client.GetAsync("latest");
        response.EnsureSuccessStatusCode();
        return await response.Content.ReadAsStringAsync();
    }
}
```

Inject the `IHttpClientFactory` using DI.

← Use the `HttpClient` in exactly the same way as you would otherwise.

<sup>2</sup> I wrote a blog post that looks in depth at how `IHttpClientFactory` achieves this rotation. This is a detailed post, but it may be of interest to those who like to know how things are implemented behind the scenes. See “Exploring the code behind `IHttpClientFactory` in depth” at <http://mng.bz/8NRK>.

The immediate benefit of using `IHttpClientFactory` in this way is efficient socket and DNS handling. Minimal changes should be required to take advantage of this pattern, as the bulk of your code stays the same. This makes it a good option if you're refactoring an existing app.

#### SocketsHttpHandler vs. `IHttpClientFactory`

The limitations of `HttpClient` described in section 21.1 apply specifically to the `HttpClientHandler` at the end of the `HttpClient` handler pipeline. `IHttpClientFactory` provides a mechanism for managing the lifetime and reuse of `HttpClientHandler` instances.

In .NET Core 2.1, a replacement for the `HttpClientHandler` was introduced: `SocketsHttpHandler`. This handler has several advantages, most notably performance benefits and consistency across platforms. The `SocketsHttpHandler` can also be configured to use connection pooling and recycling, just like `IHttpClientFactory`.

So if `HttpClient` can already use connection pooling, is it worth using `IHttpClientFactory`? In most cases, I would say yes. You must manually configure connection pooling with `SocketsHttpHandler`, and `IHttpClientFactory` has additional features such as named clients and typed clients.

Nevertheless, if you're working in a non-DI scenario, where you can't use `IHttpClientFactory`, be sure to enable the `SocketsHttpHandler` connection pooling as described in this post by Steve Gordon, titled "HttpClient connection pooling in .NET Core": <http://mng.bz/E27q>.

Managing the socket issue is one big advantage of using `IHttpClientFactory` over `HttpClient`, but it's not the only benefit. You can also use `IHttpClientFactory` to clean up the client configuration, as you'll see in the next section.

### 21.2.2 Configuring named clients at registration time

In this section you'll learn how to use the Named Client pattern with `IHttpClientFactory`. This pattern encapsulates the logic for calling a third-party API in a single location, making it easier to use the `HttpClient` in your consuming code.

Using `IHttpClientFactory` solves the technical issues I described in section 21.1, but the code in listing 21.3 is still pretty messy in my eyes. That's primarily because you must configure the `HttpClient` to point to your service before you use it. If you need to create an `HttpClient` to call the API in more than one place in your application, you must *configure* it in more than one place too.

`IHttpClientFactory` provides a convenient solution to this problem by allowing you to centrally configure *named clients*. These clients have a string name and a configuration function that runs whenever an instance of the named client is requested. You can define multiple configuration functions that run in sequence to configure your new `HttpClient`.

For example, the following listing shows how to register a named client called "rates". This client is configured with the correct `BaseAddress` and sets default headers that are to be sent with each outbound request.

#### Listing 21.4 Configuring a named client using `IHttpClientFactory` in `Startup.cs`

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddHttpClient("rates", (HttpClient client) =>
    {
        client.BaseAddress =
            new Uri("https://api.exchangeratesapi.io");
        client.DefaultRequestHeaders.Add(
            HeaderNames.UserAgent, "ExchangeRateViewer");
    })
    .ConfigureHttpClient((HttpClient client) => {})
    .ConfigureHttpClient(
        (IServiceProvider provider, HttpClient client) => {});
}
```

**You can add additional configuration functions for the named client, which run in sequence.**

**Provide a name for the client and a configuration function.**

**The configuration function runs every time the named `HttpClient` is requested.**

**Additional overloads exist that allow access to the DI container when creating a named client.**

Once you have configured this named client, you can create it from an `IHttpClientFactory` instance using the name of the client, "rates". The following listing shows how you could update listing 21.3 to use the named client configured in listing 21.4.

#### Listing 21.5 Using `IHttpClientFactory` to create a named `HttpClient`

```
[ApiController]
public class ValuesController : ControllerBase
{
    private readonly IHttpClientFactory _factory;
    public ValuesController(IHttpClientFactory factory)
    {
        _factory = factory;
    }

    [HttpGet("values")]
    public async Task<string> GetRates()
    {
        HttpClient client = _factory.CreateClient("rates");

        var response = await client.GetAsync("latest");

        response.EnsureSuccessStatusCode();
        return await response.Content.ReadAsStringAsync();
    }
}
```

**Inject the `IHttpClientFactory` using DI.**

**Request the named client called "rates" and configure it as defined in `ConfigureServices()`.**

**Use the `HttpClient` in the same way as before.**

**NOTE** You can still create unconfigured clients using `CreateClient()` without a name. Be aware that if you pass an unconfigured name, such as `CreateClient("MyRates")`, the client returned will be unconfigured. Take care—client names are case-sensitive, so "rates" is a different client than "Rates".

Named clients allow you to centralize your `HttpClient` configuration in one place, removing the responsibility of *configuring* the client from your consuming code. But you're still working with raw HTTP calls at this point—for example, providing the relative URL to call (`"/latest"`) and parsing the response. `IHttpClientFactory` includes a feature that makes it easier to clean up this code.

### 21.2.3 Using typed clients to encapsulate HTTP calls

A common pattern when you need to interact with an API is to encapsulate the mechanics of that interaction into a separate service. You could easily do this with the `IHttpClientFactory` features you've already seen by extracting the body of the `GetRates()` function from listing 21.5 into a separate service. But `IHttpClientFactory` has deeper support for this pattern too.

`IHttpClientFactory` supports *typed clients*. A typed client is a class that accepts a configured `HttpClient` in its constructor. It uses the `HttpClient` to interact with the remote API and exposes a clean interface for consumers to call. All of the logic for interacting with the remote API is encapsulated in the typed client, such as which URL paths to call, which HTTP verbs to use, and the types of responses the API returns. This encapsulation makes it easier to call the third-party API from multiple places in your app by using the typed client.

For example, the following listing shows an example typed client for the exchange rates API shown in previous listings. It accepts an `HttpClient` in its constructor and exposes a `GetLatestRates()` method that encapsulates the logic for interacting with the third-party API.

#### Listing 21.6 Creating a typed client for the exchange rates API

```
public class ExchangeRatesClient
{
    private readonly HttpClient _client;
    public ExchangeRatesClient(HttpClient client)
    {
        _client = client;
    }

    public async Task<string> GetLatestRates()
    {
        var response = await _client.GetAsync("latest");
        response.EnsureSuccessStatusCode();

        return await response.Content.ReadAsStringAsync();
    }
}
```

**Inject an HttpClient using DI instead of an `IHttpClientFactory`.**

**The `GetLatestRates()` logic encapsulates the logic for interacting with the API.**

**Use the `HttpClient` the same way as before.**

We can then inject this `ExchangeRatesClient` into consuming services, and they don't need to know anything about how to make HTTP requests to the remote service; they just need to interact with the typed client. We can update listing 21.3 to use the typed client as shown in the following listing, at which point the `GetRates()` action method becomes trivial.

#### Listing 21.7 Consuming a typed client to encapsulate calls to a remote HTTP server

```
[ApiController]
public class ValuesController : ControllerBase
{
    private readonly ExchangeRatesClient _ratesClient;
    public ValuesController(ExchangeRatesClient ratesClient)
    {
        _ratesClient = ratesClient;
    }

    [HttpGet("values")]
    public async Task<string> GetRates()
    {
        return await _ratesClient.GetLatestRates(); ← Call the typed client's API. The typed client handles making the correct HTTP requests.
    }
}
```

**Inject the typed client in the constructor.**

You may be a little confused at this point: I haven't mentioned how `IHttpClientFactory` is involved yet!

The `ExchangeRatesClient` takes an `HttpClient` in its constructor. `IHttpClientFactory` is responsible for creating the `HttpClient`, configuring it to call the remote service and injecting it into a new instance of the typed client.

You can register the `ExchangeRatesClient` as a typed client and configure the `HttpClient` that is injected in `ConfigureServices`, as shown in the following listing. This is very similar to configuring a named client, so you can register additional configuration for the `HttpClient` that will be injected into the typed client.

#### Listing 21.8 Registering a typed client with `HttpClientFactory` in `Startup.cs`

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddHttpClient<ExchangeRatesClient> ← Register a typed client using the generic AddHttpClient method.
        (HttpClient client) =>
    {
        client.BaseAddress =
            new Uri("https://api.exchangeratesapi.io");
        client.DefaultRequestHeaders.Add(
            HeaderNames.UserAgent, "ExchangeRateViewer");
    });
    .ConfigureHttpClient((HttpClient client) => {}); ← As for named clients, you can provide multiple configuration methods.
}
```

**You can provide an additional configuration function for the `HttpClient` that will be injected.**

**TIP** You can think of a typed client as a wrapper around a named client. I'm a big fan of this approach as it encapsulates all the logic for interacting with a remote service in one place. It also avoids the "magic strings" that you use with named clients, removing the possibility of typos.

Another option when registering typed clients is to register an interface in addition to the implementation. This is often a good practice, as it makes it much easier to test consuming code. For example, if the typed client in listing 21.6 implemented the interface `IExchangeRatesClient`, you could register the interface and typed client implementation using

```
services.AddHttpClient<IExchangeRatesClient, ExchangeRatesClient>()
```

You could then inject this into consuming code using the interface type:

```
public ValuesController(IExchangeRatesClient ratesClient)
```

Another commonly used pattern is to not provide any configuration for the typed client in `ConfigureServices()`. Instead, you could place that logic in the constructor of your `ExchangeRatesClient` using the injected `HttpClient`:

```
public class ExchangeRatesClient
{
    private readonly HttpClient _client;
    public ExchangeRatesClient(HttpClient client)
    {
        _client = client;
        _client.BaseAddress = new Uri("https://api.exchangeratesapi.io");
    }
}
```

This is functionally equivalent to the approach shown in listing 21.8. It's a matter of taste where you'd rather put the configuration for your `HttpClient`. If you take this approach, you don't need to provide a configuration lambda in `ConfigureServices`:

```
services.AddHttpClient<ExchangeRatesClient>();
```

Named clients and typed clients are convenient for managing and encapsulating `HttpClient` configuration, but `IHttpClientFactory` brings another advantage we haven't looked at yet: it's easier to extend the `HttpClient` handler pipeline.

## 21.3 Handling transient HTTP errors with Polly

In this section you'll learn how to handle a very common scenario: "transient" errors when you make calls to a remote service, caused by an error in the remote server or temporary network issues. You'll see how to use `IHttpClientFactory` to handle cross-cutting concerns like this by adding handlers to the `HttpClient` handler pipeline.

In section 21.2.1 I described `HttpClient` as consisting of a "pipeline" of handlers. The big advantage of this pipeline, much like the middleware pipeline of your

application, is that it allows you to add cross-cutting concerns to all requests. For example, `IHttpClientFactory` automatically adds a handler to each `HttpClient` that logs the status code and duration of each outgoing request.

As well as logging, another very common requirement is to handle transient errors when calling an external API. Transient errors can happen when the network drops out, or if a remote API goes offline temporarily. For transient errors, simply trying the request again can often succeed, but having to *manually* write the code to do so is cumbersome.

ASP.NET Core includes a library called `Microsoft.Extensions.Http.Polly` that makes handling transient errors easier. It uses the popular open source library Polly (<https://github.com/App-vNext/Polly>) to automatically retry requests that fail due to transient network errors.

Polly is a mature library for handling transient errors that includes a variety of different error-handling strategies, such as simple retries, exponential backoff, circuit breaking, bulkhead isolation, and many more. Each strategy is explained in detail at <https://github.com/App-vNext/Polly>, so be sure to read about the benefits and trade-offs when selecting a strategy.

To provide a taste of what's available, we'll add a simple retry policy to the `ExchangeRatesClient` shown in section 21.2. If a request fails due to a network problem, such as a timeout or a server error, we'll configure Polly to automatically retry the request as part of the handler pipeline, as shown in figure 21.5.

To add transient error handling to a named client or `HttpClient`, follow these steps:

- 1 Install the `Microsoft.Extensions.Http.Polly` NuGet package in your project by running `dotnet add package Microsoft.Extensions.Http.Polly`, or by using the NuGet explorer in Visual Studio, or by adding a `<PackageReference>` element to your project file as follows:

```
<PackageReference Include="Microsoft.Extensions.Http.Polly"
    Version="5.0.0" />
```

- 2 Configure a named or typed client as shown in listings 21.5 and 21.7.
- 3 Configure a transient error-handling policy for your client as shown in listing 21.9.

#### Listing 21.9 Configuring a transient error-handling policy for a typed client in `Startup.cs`

```
public void ConfigureServices(IServiceCollection services)
{
    services.AddHttpClient<ExchangeRatesClient>()
        .AddTransientHttpErrorPolicy(policy =>
            policy.WaitAndRetryAsync(new[] {
                TimeSpan.FromMilliseconds(200),
                TimeSpan.FromMilliseconds(500),
                TimeSpan.FromSeconds(1)
            }));
}
```

**Configures a policy that waits and retries three times if an error occurs**

**You can add transient error handlers to named or typed clients.**

**Use the extension methods provided by the NuGet package to add transient error handlers.**

**Configure the retry policy used by the handler. There are many types of policies to choose from.**

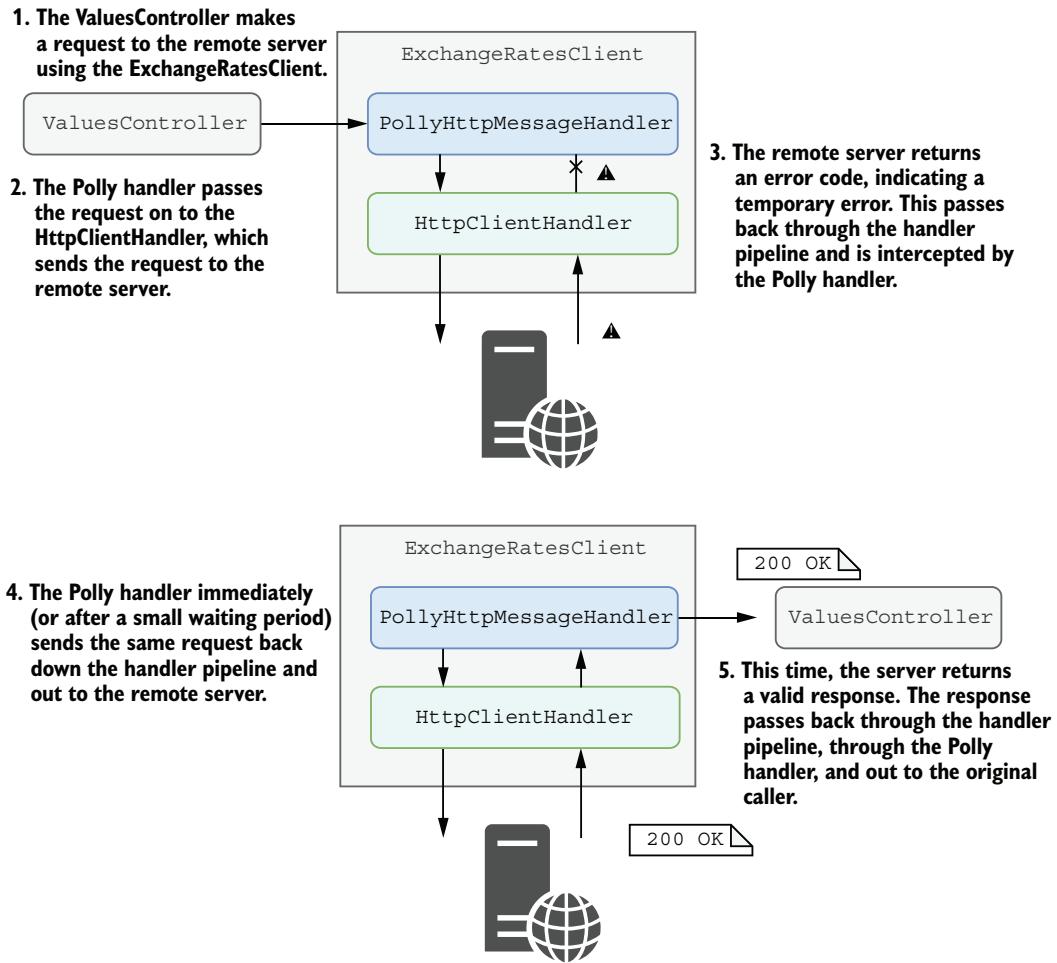


Figure 21.5 Using the `PolicyHttpMessageHandler` to handle transient errors. If an error occurs when calling the remote API, the Polly handler will automatically retry the request. If the request then succeeds, the result is passed back to the caller. The caller didn't have to handle the error themselves, making it simpler to use the `HttpClient` while remaining resilient to transient errors.

In the preceding listing we configure the error handler to catch transient errors and retry three times, waiting an increasing amount of time between requests. If the request fails on the third try, the handler will ignore the error and pass it back to the client, just as if there was no error handler at all. By default, the handler will retry any request that either

- Throws an `HttpRequestException`, indicating an error at the protocol level, such as a closed connection
- Returns an HTTP 5xx status code, indicating a server error at the API
- Returns an HTTP 408 status code, indicating a timeout

**TIP** If you want to handle more cases automatically, or to restrict the responses that will be automatically retried, you can customize the selection logic as described in the “Polly and HttpClientFactory” documentation on GitHub: <http://mng.bz/NY7E>.

Using standard handlers like the transient error handler allows you to apply the same logic across all requests made by a given `HttpClient`. The exact strategy you choose will depend on the characteristics of both the service and the request, but a good retry strategy is a must whenever you interact with potentially unreliable HTTP APIs.

The Polly error handler is an example of an optional `HttpMessageHandler` that you can plug in to your `HttpClient`, but you can also create your own custom handler. In the next section you’ll see how to create a handler that adds a header to all outgoing requests.

## 21.4 Creating a custom `HttpMessageHandler`

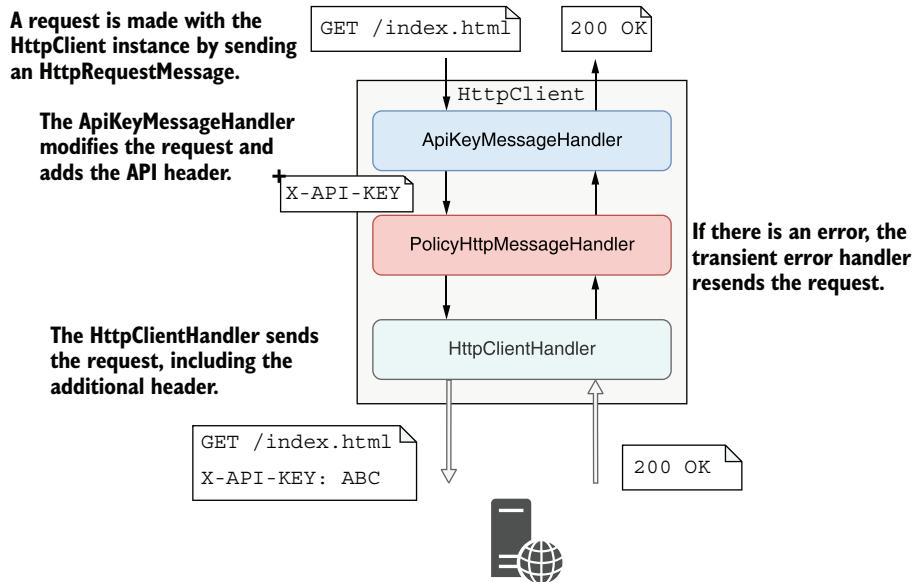
Most third-party APIs will require some form of authentication when you’re calling them. For example, many services require you attach an API key to an outgoing request, so that the request can be tied to your account. Instead of having to remember to manually add this header for every request to the API, you could configure a custom `HttpMessageHandler` to automatically attach the header for you.

**NOTE** More complex APIs may use JSON Web Tokens (JWT) obtained from an identity provider. If that’s the case, consider using the open source IdentityModel library (<https://identitymodel.readthedocs.io>), which provides integration points for ASP.NET Core Identity and `HttpClientFactory`.

You can configure a named or typed client using `IHttpClientFactory` to use your API-key handler as part of the `HttpClient`’s handler pipeline, as shown in figure 21.6. When you use the `HttpClient` to send a message, the `HttpRequestMessage` is passed through each handler in turn. The API-key handler adds the extra header and passes the request to the next handler in the pipeline. Eventually the `HttpClientHandler` makes the network request to send the HTTP request. After the response is received, each handler gets a chance to inspect (and potentially modify) the response.

To create a custom `HttpMessageHandler` and add it to a typed or named client’s pipeline, you can follow these steps:

- 1 Create a custom handler by deriving from the `DelegatingHandler` base class.
- 2 Override the `SendAsync()` method to provide your custom behavior. Call `base.SendAsync()` to execute the remainder of the handler pipeline.
- 3 Register your handler with the DI container. If your handler does not require state, you can register it as a singleton service; otherwise you should register it as a transient service.
- 4 Add the handler to one or more of your named or typed clients by calling `AddHttpMessageHandler<T>()` on an `IHttpClientBuilder`, where `T` is your



**Figure 21.6** You can use a custom `HttpMessageHandler` to modify requests before they're sent to third-party APIs. Every request passes through the custom handler before the final handler (the `HttpClientHandler`) sends the request to the HTTP API. After the response is received, each handler gets a chance to inspect and modify the response.

handler type. The order in which you register handlers dictates the order in which they will be added to the `HttpClient` handler pipeline. You can add the same handler type more than once in a pipeline if you wish, and to multiple typed or named clients.

The following listing shows an example of a custom `HttpMessageHandler` that adds a header to every outgoing request. We'll use the custom "X-API-KEY" header in this example, but the header you need will vary depending on the third-party API you're calling. This example uses strongly typed configuration to inject the secret API key, as you saw in chapter 10.

#### Listing 21.10 Creating a custom `HttpMessageHandler`

```
public class ApiKeyMessageHandler : DelegatingHandler {  
    private readonly ExchangeRateApiSettings _settings;  
    public ApiKeyMessageHandler(  
        IOptions<ExchangeRateApiSettings> settings)  
    {  
        _settings = settings.Value;  
    }  
}
```

**Inject the strongly typed configuration values using dependency injection.**

**Custom `HttpMessageHandlers` should derive from `DelegatingHandler`.**

**Override the `SendAsync` method to implement the custom behavior.**

```

protected override async Task<HttpResponseMessage> SendAsync(
    HttpRequestMessage request,
    CancellationToken cancellationToken)
{
    request.Headers.Add("X-API-KEY", _settings.ApiKey); ← Add the extra header to all outgoing requests.

    HttpResponseMessage response =
        await base.SendAsync(request, cancellationToken);

    return response; ← You could inspect or modify the response before returning it.
}

```

**Call the remainder of the pipeline and receive the response.**

To use the handler, you must register it with the DI container and add it to a named or typed client. In the following listing, we add it to the `ExchangeRatesClient`, along with the transient error handler we registered in listing 21.8. This creates a pipeline similar to that shown in figure 21.6.

#### Listing 21.11 Registering a custom handler in `Startup.ConfigureServices`

```

public void ConfigureServices(IServiceCollection services)
{
    services.AddTransient<ApiKeyMessageHandler>(); ← Register the custom handler with the DI container.

    services.AddHttpClient<ExchangeRatesClient>()
        .AddHttpMessageHandler<ApiKeyMessageHandler>() ← Configure the typed client to use the custom handler.
        .AddTransient(policy =>
            policy.WaitAndRetryAsync(new[] {
                TimeSpan.FromMilliseconds(200),
                TimeSpan.FromMilliseconds(500),
                TimeSpan.FromSeconds(1)
            })
        );
}

```

**Add the transient error handler. The order in which they are registered dictates their order in the pipeline.**

Whenever you make a request using the typed client `ExchangeRatesClient`, you can be sure the API key will be added and that transient errors will be handled automatically for you.

That brings us to the end of this chapter on `IHttpClientFactory`. Given the difficulties in using `HttpClient` correctly that I showed in section 21.1, you should always favor `IHttpClientFactory` where possible. As a bonus, `IHttpClientFactory` allows you to easily centralize your API configuration using named clients and to encapsulate your API interactions using typed clients.

### Summary

- Use the `HttpClient` class for calling HTTP APIs. You can use it to make HTTP calls to APIs, providing all the headers and body to send in a request, and reading the response headers and data you get back.

- `HttpClient` uses a pipeline of handlers, consisting of multiple `HttpMessageHandlers`, connected in a similar way to the middleware pipeline used in ASP.NET Core. The final handler is the `HttpClientHandler`, which is responsible for making the network connection and sending the request.
- `HttpClient` implements `IDisposable`, but you shouldn't typically dispose of it. When the `HttpClientHandler` that makes the TCP/IP connection is disposed of, it keeps a connection open for the `TIME_WAIT` period. Disposing of many `HttpClients` in a short period of time can lead to socket exhaustion, preventing a machine from handling any more requests.
- Prior to .NET Core 2.1, the advice was to use a single `HttpClient` for the lifetime of your application. Unfortunately, a singleton `HttpClient` will not respect DNS changes, which are commonly used for traffic management in cloud environments.
- `IHttpClientFactory` solves both these problems by managing the lifetime of the `HttpMessageHandler` pipeline. You can create a new `HttpClient` by calling `CreateClient()`, and `IHttpClientFactory` takes care of disposing of the handler pipeline when it is no longer in use.
- You can centralize the configuration of an `HttpClient` in `ConfigureServices()` using *named* clients by calling `AddHttpClient("test", c => {})`. You can then retrieve a configured instance of the client in your services by calling `IHttpClientFactory.CreateClient("test")`.
- You can create a *typed* client by injecting an `HttpClient` into a service, `T`, and configuring the client using `AddHttpClient<T>(c => {})`. Typed clients are great for abstracting the HTTP mechanics away from consumers of your client.
- You can use the `Microsoft.Extensions.Http.Polly` library to add transient HTTP error handling to your `HttpClients`. Call `AddTransientHttpErrorPolicy()` when configuring your `IHttpClientFactory` in `ConfigureServices`, and provide a `Polly` policy to control when errors should be automatically handled and retried.
- It's common to use a simple retry policy to try making a request multiple times before giving up and returning an error. When designing a policy, be sure to consider the impact of your policy; in some circumstances it may be better to fail quickly instead of retrying a request that is never going to succeed. `Polly` includes additional policies such as circuit-breakers to create more advanced approaches.
- By default, the transient error-handling middleware will handle connection errors, server errors that return a 5xx error code, and 408 (timeout) errors. You can customize this if you want to handle additional error types, but ensure that you only retry requests that are safe to do so.
- You can create a custom `HttpMessageHandler` to modify each request made through a named or typed client. Custom handlers are good for implementing cross-cutting concerns such as logging, metrics, and authentication.

- To create a custom `HttpMessageHandler`, derive from `DelegatingHandler` and override the `SendAsync()` method. Call `base.SendAsync()` to send the request to the next handler in the pipeline and finally to the `HttpClientHandler`, which makes the HTTP request.
- Register your custom handler in the DI container as either a transient or a singleton. Add it to a named or typed client using `AddHttpMessageHandler<T>()`. The order in which you register the handler in the `IHttpClientBuilder` is the order in which the handler will appear in the `HttpClient` handler pipeline.



# *Building background tasks and services*

## **This chapter covers**

- Creating tasks that run in the background for your application
- Using the generic `IHost` to create Windows Services and Linux daemons
- Using Quartz.NET to run tasks on a schedule in a clustered environment

We've covered a lot of ground in the book so far. You've learned how to create page-based applications using Razor Pages and how to create APIs for mobile clients and services. You've seen how to add authentication and authorization to your application, how to use EF Core for storing state in the database, and how to create custom components to meet your requirements.

As well as these UI-focused apps, you may find you need to build background or batch-task services. These services aren't meant to interact with users directly. Rather, they stay running in the background, processing items from a queue or periodically executing a long-running process.

For example, you might want to have a background service that sends email confirmations for e-commerce orders, or a batch job that calculates sales and losses

for retail stores after the shops close. ASP.NET Core includes support for these background tasks by providing abstractions for running a task in the background when your application starts.

In section 22.1 you'll learn about the background task support provided in ASP.NET Core by the `IHostedService` interface. You'll learn how to use the `BackgroundService` helper class to create tasks that run on a timer, and how to manage your DI lifetimes correctly in a long-running task.

In section 22.2 we'll take the background service concept one step further to create “headless” worker services using the generic `IHost`. Worker services don't use Razor Pages or API controllers; instead, they consist only of `IHostedServices` running tasks in the background. You'll also see how to configure and install a worker service app as a Windows Service or as a Linux daemon.

In section 22.3 I'll introduce the open source library Quartz.NET, which provides extensive scheduling capabilities for creating background services. You'll learn how to install Quartz.NET in your applications, how to create complex schedules for your tasks, and how to add redundancy to your worker services by using clustering.

Before we get to more complex scenarios, we'll start by looking at the built-in support for running background tasks in your apps.

## 22.1 **Running background tasks with `IHostedService`**

In most applications it's common to create tasks that happen in the background, rather than in response to a request. This could be a task to process a queue of emails, handling events published to some sort of a message bus, or running a batch process to calculate daily profits. By moving this work to a background task, your user interface can stay responsive. Instead of trying to send an email immediately, for example, you could add the request to a queue and return a response to the user immediately. The background task can consume that queue in the background at its leisure.

In ASP.NET Core, you can use the `IHostedService` interface to run tasks in the background. Classes that implement this interface are started when your application starts, shortly after your application starts handling requests, and they are stopped shortly before your application is stopped. This provides the hooks you need to perform most tasks.

**NOTE** Even the ASP.NET Core server, Kestrel, runs as an `IHostedService`. In one sense, almost everything in an ASP.NET Core app is a “background” task.

In this section you'll see how to use the `IHostedService` to create a background task that runs continuously throughout the lifetime of your app. This could be used for many different things, but in the next section you'll see how to use it to populate a simple cache. You'll also learn how to use services with a scoped lifetime in your singleton background tasks by managing container scopes yourself.

### 22.1.1 Running background tasks on a timer

In this section you'll learn how to create a background task that runs periodically on a timer, throughout the lifetime of your app. Running background tasks can be useful for many reasons, such as scheduling work to be performed later or performing work in advance.

For example, in chapter 21 we used `IHttpClientFactory` and a typed client to call a third-party service to retrieve the current exchange rate between various currencies and returned them in an API controller, as shown in the following listing.

#### Listing 22.1 Using a typed client to return exchange rates from a third-party service

```
[ApiController]
public class ValuesController : ControllerBase
{
    private readonly ExchangeRatesClient _typedClient;
    public ValuesController(ExchangeRatesClient typedClient)
    {
        _typedClient = typedClient;
    }

    [HttpGet("values")]
    public async Task<string> GetRates()
    {
        return await _typedClient.GetLatestRatesAsync();
    }
}
```

A typed client created using `IHttpClientFactory` is injected in the constructor.

The typed client is used to retrieve exchange rates from the remote API and returns them.

A simple optimization for this code might be to cache the exchange rate values for a period. There are multiple ways you could implement that, but in this section we'll use a simple cache that preemptively fetches the exchange rates in the background, as shown in figure 22.1. The API controller simply reads from the cache; it never has to make HTTP calls itself, so it remains fast.

**NOTE** An alternative approach might add caching to your strongly typed client, `ExchangeRateClient`. The downside is that when you need to update the rates, you will have to perform the request immediately, making the overall response slower. Using a background service keeps your API controller consistently fast.

You can implement a background task using the `IHostedService` interface. This consists of two methods:

```
public interface IHostedService
{
    Task StartAsync(CancellationToken cancellationToken);
    Task StopAsync(CancellationToken cancellationToken);
}
```

There are subtleties to implementing the interface correctly. In particular, the `StartAsync()` method, although asynchronous, runs inline as part of your application

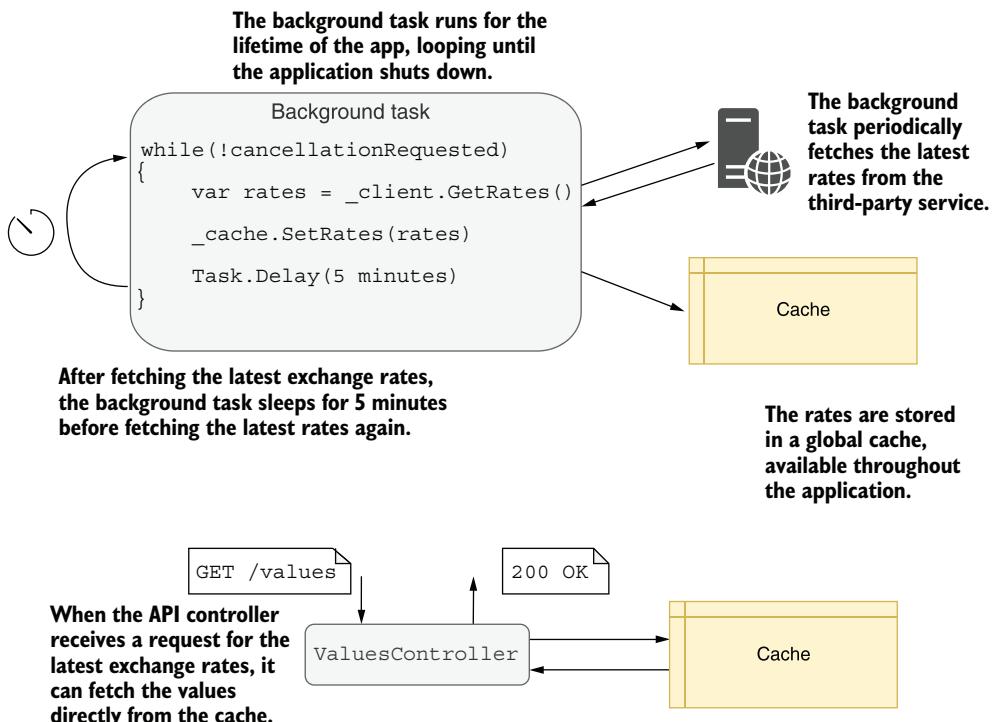


Figure 22.1 You can use a background task to cache the results from a third-party API on a schedule. The API controller can then read directly from the cache instead of calling the third-party API itself. This reduces the latency of requests to your API controller, while ensuring the data remains fresh.

startup. Background tasks that are expected to run for the lifetime of your application must return a `Task` immediately and schedule background work on a different thread.

**WARNING** Calling `await` in the `IHostedService.StartAsync()` method will block your application from starting until the method completes. This can be useful in some cases, but it's often not the desired behavior for background tasks.

To make it easier to create background services using best-practice patterns, ASP.NET Core provides the abstract base class `BackgroundService`, which implements `IHostedService` and is designed to be used for long-running tasks. To create a background task you must override a single method of this class, `ExecuteAsync()`. You're free to use `async-await` inside this method, and you can keep running the method for the lifetime of your app.

For example, the following listing shows a background service that fetches the latest interest rates using a typed client and saves them in a cache, as you saw in figure 22.1. The `ExecuteAsync()` method keeps looping and updating the cache until

the `CancellationToken` passed as an argument indicates that the application is shutting down.

### Listing 22.2 Implementing a `BackgroundService` that calls a remote HTTP API

```
Derive from BackgroundService to create a task  
that runs for the lifetime of your app.  

public class ExchangeRatesHostedService : BackgroundService  
{  
    private readonly IServiceProvider _provider;  
    private readonly ExchangeRatesCache _cache;  
    public ExchangeRatesHostedService(  
        IServiceProvider provider, ExchangeRatesCache cache)  
    {  
        _provider = provider;  
        _cache = cache;  
    }  
  
    protected override async Task ExecuteAsync(  
        CancellationToken stoppingToken)  
    {  
        while (!stoppingToken.IsCancellationRequested)  
        {  
            var client = _provider  
                .GetRequiredService<ExchangeRatesClient>();  
  
            string rates = await client.GetLatestRatesAsync();  
            _cache.SetRates(rates);  
  
            await Task.Delay(TimeSpan.FromMinutes(5), stoppingToken);  
        }  
    }  
}
```

**A simple cache for exchange rates**

**Keep looping until the application shuts down.**

**Create a new instance of the typed client so that the `HttpClient` is short-lived.**

**Store the rates in the cache.**

**You must override `ExecuteAsync` to set the service's behavior.**

**Inject an `IServiceProvider` so you can create instances of the typed client.**

**The `CancellationToken` passed as an argument is triggered when the application shuts down.**

**Fetch the latest rates from the remote API.**

**Wait for 5 minutes (or for the application to shut down) before updating the cache.**

The `ExchangeRateCache` in listing 22.2 is a simple singleton that stores the latest rates. It must be thread-safe, as it will be accessed concurrently by your API controllers. You can see a simple implementation in the source code for this chapter.

To register your background service with the DI container, use the `AddHostedService()` extension method in the `ConfigureServices()` method of `Startup.cs`, as shown in the following listing.

### Listing 22.3 Registering an `IHostedService` with the DI container

```
public void ConfigureServices(IServiceCollection services)  
{  
    services.AddHttpClient<ExchangeRatesClient>();  
    services.AddSingleton<ExchangeRatesCache>();  
    services.AddHostedService<ExchangeRatesHostedService>();  
}
```

**Register the typed client as before.**

**Add the `cache` object as a singleton, because you must share the same instance throughout your app.**

**Register `ExchangeRatesHostedService` as an `IHostedService`.**

By using a background service to fetch the exchange rates, your API controller becomes very simple. Instead of fetching the latest rates itself, it returns the value from the cache, which is kept up to date by the background service:

```
[ApiController]
public class ValuesController : ControllerBase
{
    private readonly ExchangeRatesCache _cache;
    public ValuesController(ExchangeRatesCache cache)
    {
        _cache = cache;
    }

    [HttpGet("values")]
    public string GetValues()
    {
        return _cache.GetLatestRates();
    }
}
```

One slightly messy aspect of listing 22.2 is that I've used the Service Locator pattern to retrieve the typed client. This isn't ideal, but you shouldn't inject typed clients into background services directly. Typed clients are designed to be short-lived, to ensure you take advantage of the `HttpClient` handler rotation as described in chapter 21. In contrast, background services are singletons that live for the lifetime of your application.

**TIP** If you wish, you can avoid the Service Locator pattern in listing 22.2 by using the factory pattern described in Steve Gordon's post titled "IHttpClientFactory Patterns: Using Typed Clients from Singleton Services": <http://mng.bz/opDZ>.

The need for short-lived services leads to another common question—how can you use scoped services in a background service?

### 22.1.2 Using scoped services in background tasks

Background services that implement `IHostedService` are created once when your application starts. That means they are, by necessity, singletons, as there will only ever be a single instance of the class.

That leads to a problem if you need to use services registered with a *scoped* lifetime. Any services you inject into the constructor of your singleton `IHostedService` must themselves be registered as singletons. Does that mean there's no way to use scoped dependencies in a background service?

**REMINDER** As I discussed in chapter 10, the dependencies of a service must always have a lifetime that's the same or longer than that of the service itself, to avoid captured dependencies.

For example, let's imagine a slight variation of the caching example from section 22.1.1. Instead of storing the exchange rates in a singleton cache object, you want to save the exchange rates to a database so you can look up the historic rates.

Most database providers, including EF Core's `DbContext`, register their services with scoped lifetimes. That means you need to access the *scoped* `DbContext`, from inside the *singleton* `ExchangeRatesHostedService`, which precludes injecting the `DbContext` with constructor injection. The solution is to create a new container scope every time you update the exchange rates.

In typical ASP.NET Core applications, the framework creates a new container scope every time a new request is received, just before the middleware pipeline executes. All the services that are used in that request are fetched from the scoped container. In a background service, however, there *are* no requests, so no container scopes are created. The solution is to create your own.

You can create a new container scope anywhere you have access to an `IServiceProvider` by calling `IServiceProvider.CreateScope()`. This creates a scoped container, which you can use to retrieve scoped services.

**WARNING** Always make sure to dispose of the `IServiceScope` returned by `CreateScope()` when you're finished with it, typically with a `using` statement. This disposes of any services that were created by the scoped container and prevents memory leaks.

The following listing shows a version of the `ExchangeRatesHostedService` that stores the latest exchange rates as an EF Core entity in the database. It creates a new scope for each iteration of the `while` loop and retrieves the scoped `AppDbContext` from the scoped container.

#### Listing 22.4 Consuming scoped services from an `IHostedService`

```
→ public class ExchangeRatesHostedService : BackgroundService
{
    private readonly IServiceProvider _provider;
    public ExchangeRatesHostedService(IServiceProvider provider)
    {
        _provider = provider;
    }

    protected override async Task ExecuteAsync(
        CancellationToken stoppingToken)
    {
        while (!stoppingToken.IsCancellationRequested)
        {
            Create
            a new scope
            using the root
            IServiceProvider.
            → using(IServiceScope scope = _provider.CreateScope())
            {
                var scopedProvider = scope.ServiceProvider;
                ←
                // ...
            }
        }
    }
}
```

**Background-Service is registered as a singleton.**

**The injected `IServiceProvider` can be used to retrieve singleton services, or to create scopes.**

**Create a new scope using the root `IServiceProvider`.**

**The scope exposes an `IServiceProvider` that can be used to retrieve scoped components.**

Creating scopes like this is a general solution whenever you find you need to access scoped services and you’re not running in the context of a request. A prime example is when you’re implementing `IConfigureOptions`, as you saw in chapter 19. You can take the exact same approach—creating a new scope—as shown in my blog post titled “Access services inside ConfigureServices using `IConfigureOptions` in ASP.NET Core”: <http://mng.bz/nMD5>.

**TIP** Using service location in this way always feels a bit convoluted. I typically try to extract the body of the task to a separate class and use service location to retrieve that class only. You can see an example of this approach in the “Consuming a scoped service in a background task” section of Microsoft’s “Background tasks with hosted services in ASP.NET Core” documentation: <http://mng.bz/4ZER>.

`IHostedService` is available in ASP.NET Core, so you can run background tasks in your Razor Pages or API controller applications. However, sometimes all you want is the background task and you don't need any UI. For those cases, you can use the raw `IHost` abstraction, without having to bother with HTTP handling at all.

## 22.2 Creating headless worker services using IHost

In this section you'll learn about worker services, which are ASP.NET Core applications that do not handle HTTP traffic. You'll learn how to create a new worker service from a template and compare the generated code to a traditional ASP.NET Core application. You'll also learn how to install the worker service as a Windows Service or as a systemd daemon on Linux.

In section 22.1 we cached exchange rates based on the assumption that they're being consumed directly by the UI part of your application; by Razor Pages or API controllers, for example. However, in the section 22.1.2 example we saved the rates to the database instead of storing them in-process. That raises the possibility of *other* applications with access to the database using the rates too. Taking that one step further,

could we create an application which is *only* responsible for caching these rates and has no UI at all?

Since .NET Core 3.0, ASP.NET Core has been built on top of a “generic” (as opposed to a “web”) `IHost` implementation. It is the `IHost` implementation that provides features such as configuration, logging, and dependency injection. ASP.NET Core adds the middleware pipeline for handling HTTP requests, as well as paradigms such as Razor Pages or MVC on top of that, as shown in figure 22.2.

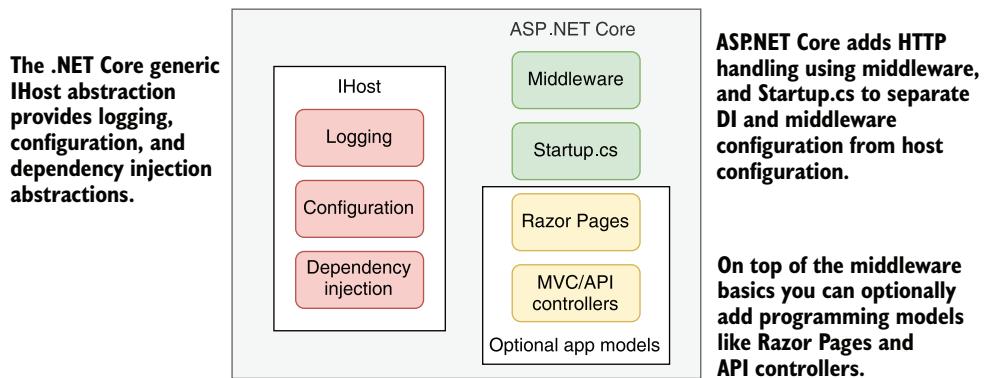


Figure 22.2 ASP.NET Core builds on the generic `IHost` implementation. `IHost` provides features such as configuration, dependency injection, and configuration. ASP.NET Core adds HTTP handling on top of that by way of the middleware pipeline, Razor Pages, and API controllers. If you don't need HTTP handling, you can use `IHost` without the additional ASP.NET Core libraries to create a smaller application.

If your application doesn't *need* to handle HTTP requests, there's no real reason to use ASP.NET Core. You can use the `IHost` implementation alone to create an application that will have a lower memory footprint, faster startup, and less surface area to worry about from a security perspective than a full ASP.NET Core application. .NET Core applications that use this approach are commonly called *worker services* or *workers*.

**DEFINITION** A *worker* is a .NET Core application that uses the generic `IHost` but doesn't include the ASP.NET Core libraries for handling HTTP requests. They are sometimes called “headless” services, as they don't expose a UI for you to interact with.

Workers are commonly used for running background tasks (`IHostedService` implementations) that don't require a UI. These tasks could be for running batch jobs, for running tasks repeatedly on a schedule, or for handling events using some sort of message bus. In the next section we'll create a worker for retrieving the latest exchange rates from a remote API, instead of adding the background task to an ASP.NET Core application.

## 22.2.1 Creating a worker service from a template

In this section you'll see how to create a basic worker service from a template. Visual Studio includes a template for creating worker services: select File > New > Project > Worker Service. You can create a similar template using the .NET CLI by running `dotnet new worker`. The resulting template consists of two C# files:

- *Worker.cs*—This is a simple `BackgroundService` implementation that writes to the log every second. You can replace this class with your own `BackgroundService` implementation, such as the example from listing 22.4.
- *Program.cs*—As in a typical ASP.NET Core application, this contains the entry point for your application, and it's where the `IHost` is built and run. In contrast to a typical ASP.NET Core app, it's *also* where you will configure the dependency injection container for your application.

The most notable difference between the worker service template and an ASP.NET Core template is that there is no `Startup.cs` file. In ASP.NET Core applications, `Startup.cs` is where you usually configure your DI container and your middleware pipeline. The worker service doesn't have a middleware pipeline (as it doesn't handle HTTP requests), but it *does* use DI, so where is that configured?

In worker service templates you configure DI in `Program.cs` using the `ConfigureServices()` method, as shown in the following listing. This method is functionally identical to the `ConfigureServices()` method in `Startup.cs`, so you can use exactly the same syntax. The following listing shows how to configure EF Core, the exchange rates typed client from chapter 21, and the background service that saves exchange rates to the database, as you saw in section 22.1.2.

### Listing 22.5 Program.cs for a worker service that saves exchange rates using EF Core

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureServices((hostContext, services) =>
            {
                services.AddHttpClient<ExchangeRatesClient>();
                services
                    .AddHostedService<ExchangeRatesHostedService>();

                services.AddDbContext<AppDbContext>(options =>
                    options.UseSqlite(
                        hostContext.Configuration
                            .GetConnectionString("SqlLiteConnectionString")));
            });
}
```

**The same Host-Builder code is used, but there is no call to ConfigureServices-WebHostDefaults.**

**IConfiguration can be accessed from the HostBuilderContext parameter.**

A worker creates an `IHostBuilder`, builds an `IHost`, and runs it, the same as an ASP.NET Core app.

Add services in `ConfigureServices`, just as you typically would in `Startup.cs`.

```

        );
    });
}

```

**TIP** You can use the `IHostBuilder.ConfigureServices()` methods in ASP.NET Core apps too, but the general convention is to use `Startup.cs` instead. The `IHostBuilder` methods are useful in some circumstances where you need to control exactly when your background tasks start, as I describe in a blog post titled “Controlling `IHostedService` execution order in ASP.NET Core 3.x”: <http://mng.bz/Qm66>.

The changes in `Program.cs`, and the lack of a `Startup.cs` file, are the most obvious differences between a worker service and an ASP.NET Core app, but there are some important differences in the `.csproj` project file too. The following listing shows the project file for a worker service that uses `IHttpClientFactory` and EF Core, and highlights some of the differences compared to a similar ASP.NET Core application.

#### Listing 22.6 Project file for a worker service

```

<Project Sdk="Microsoft.NET.Sdk.Worker">
  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
    <UserSecretsId>5088-4277-B226-DC0A790AB790</UserSecretsId>
  </PropertyGroup>

  <ItemGroup>
    <PackageReference Include="Microsoft.Extensions.Hosting"
      Version="5.0.0" />
    <PackageReference Include="Microsoft.Extensions.Http"
      Version="5.0.0" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Design"
      Version="5.0.0" PrivateAssets="All" />
    <PackageReference Include="Microsoft.EntityFrameworkCore.Sqlite"
      Version="5.0.0" />
  </ItemGroup>
</Project>

```

The target framework is the same as for ASP.NET Core apps.

If you're using `IHttpClientFactory`, you'll need to add this package in worker services.

Worker services use a different project SDK type than ASP.NET Core apps.

Worker services use configuration, so they can use `UserSecrets`, like ASP.NET Core apps.

All worker services must explicitly add this package. ASP.NET Core apps add it implicitly.

EF Core packages must be explicitly added, the same as for ASP.NET Core apps.

Some parts of the project file are the same for both worker services and ASP.NET Core apps:

- Both types of apps must specify a `<TargetFramework>`, such as `netcoreapp3.1` for .NET Core 3.1, or `net5.0` for .NET 5.0.
- Both types of apps use the configuration system, so you can use `<UserSecretsId>` to manage secrets in development, as discussed in chapter 11.
- Both types of apps must explicitly add references to the EF Core NuGet packages to use EF Core in the app.

There are also several differences in the project template:

- The `<Project>` element's `Sdk` for a worker service should be `Microsoft.NET.Sdk.Worker`, while for an ASP.NET Core app it is `Microsoft.NET.Sdk.Web`. The Web SDK includes implicit references to additional packages that are not generally required in worker services.
- The worker service *must* include an explicit `PackageReference` for the `Microsoft.Extensions.Hosting` NuGet package. This package includes the generic `IHost` implementation used by worker services.
- You may need to include additional packages to reference the same functionality, when compared to an ASP.NET Core app. An example of this is the `Microsoft.Extensions.Http` package (which provides `IHttpClientFactory`). This package is referenced implicitly in ASP.NET Core apps but must be explicitly referenced in worker services.

Running a worker service is the same as running an ASP.NET Core application: use `dotnet run` from the command line or press F5 from Visual Studio. A worker service is essentially just a console application (as are ASP.NET Core applications), so they both run the same way.

You can run worker services in most of the same places you would run an ASP.NET Core application, though as a worker service doesn't handle HTTP traffic, some options make more sense than others. In the next section we'll look at two supported ways of running your application: as a Windows Service or as a Linux systemd daemon.

### 22.2.2 Running worker services in production

In this section you'll learn how to run worker services in production. You'll learn how to install a worker service as a Windows Service so that the operating system monitors and starts your worker service automatically. You'll also see how to prepare your application for installation as a systemd daemon on Linux.

Worker services, like ASP.NET Core applications, are fundamentally just .NET Core console applications. The difference is that they are typically intended to be long-running applications. The common approach for running these types of applications on Windows is to use a Windows Service or to use a systemd daemon on Linux.

**NOTE** It's also very common to run applications in the cloud using Docker containers or dedicated platform services like Azure App Service. The process for deploying a worker service to these managed services is typically identical to deploying an ASP.NET Core application.

Adding support for Windows Services or systemd is easy, thanks to two optional NuGet packages:

- `Microsoft.Extensions.Hosting.Systemd`—Adds support for running the application as a systemd application. To enable systemd integration, call `UseSystemd()` on your `IHostBuilder` in `Program.cs`.

- *Microsoft.Extensions.Hosting.WindowsServices*—Adds support for running the application as a Windows Service. To enable the integration, call `UseWindowsService()` on your `IHostBuilder` in `Program.cs`.

These packages each add a single extension method to `IHostBuilder` that enables the appropriate integration when running as a `systemd` daemon or as a Windows Service. For example, the following listing shows how to enable Windows Service support.

#### Listing 22.7 Adding Windows Service support to a worker service

```
public class Program
{
    public static void Main(string[] args)
    {
        CreateHostBuilder(args).Build().Run();
    }

    public static IHostBuilder CreateHostBuilder(string[] args) =>
        Host.CreateDefaultBuilder(args)
            .ConfigureServices((hostContext, services) =>
            {
                services.AddHostedService<Worker>();
            })
            .UseWindowsService();      ←
    }
}
```

**Configure your worker service as you would normally.**

**Add support for running as a Windows Service.**

During development, or if you run your application as a console app, `AddWindowsService()` does nothing; your application runs exactly the same as it would without the method call. However, your application can now be installed as a Windows Service, as your app now has the required integration hooks to work with the Windows Service system.

The following basic steps show how to install a worker service app as a Windows Service:

- 1 Add the `Microsoft.Extensions.Hosting.WindowsServices` NuGet package to your application using Visual Studio by running `dotnet add package Microsoft.Extensions.Hosting.WindowsServices` in the project folder, or by adding a `<PackageReference>` to your `.csproj` file:

```
<PackageReference Include="Microsoft.Extensions.Hosting.WindowsServices"
Version="5.0.0" />
```

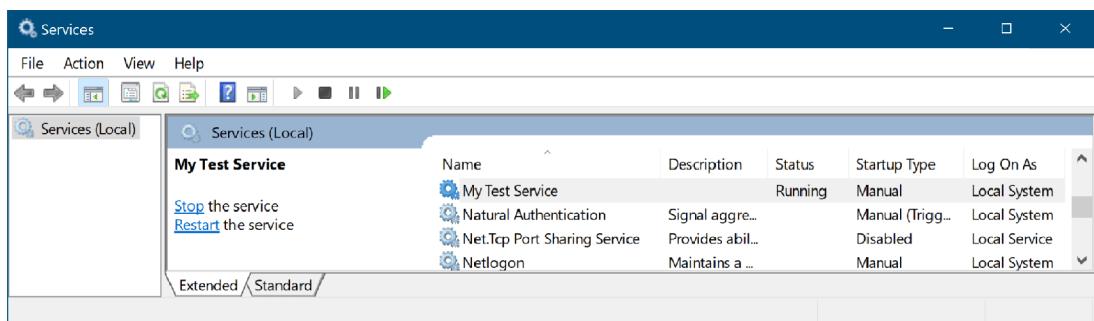
- 2 Add a call to `UseWindowsService()` on your `IHostBuilder`, as shown in listing 22.7.
- 3 Publish your application, as described in chapter 16. From the command line you could run `dotnet publish -c Release` from the project folder.

- 4 Open a command prompt as Administrator, and install the application using the Windows sc utility. You need to provide the path to your published project's .exe file and a name to use for the service, such as My Test Service:

```
sc create "My Test Service" BinPath="C:\path\to\MyService.exe"
```

- 5 You can manage the service from the Services control panel in Windows, as shown in figure 22.3. Alternatively, to start the service from the command line run sc start "My Test Service", or to delete the service run sc delete "My Test Service".

After following the preceding steps, your worker service will be running as a Windows Service.



**Figure 22.3** The Services control panel in Windows. After installing a worker service as a Windows Service using the sc utility, you can manage your worker service from here. This allows you to control when the Windows Service starts and stops, the user account the application runs under, and how to handle errors.

**WARNING** These steps are the bare minimum required to install a Windows Service. When running in production, you must consider many security aspects not covered here. For more details, see Microsoft's "Host ASP.NET Core in a Windows Service" documentation: <http://mng.bz/Xdy9>.

An interesting point of note is that installing as a Windows Service or systemd daemon isn't limited to worker services only—you can install an ASP.NET Core application in the same way. Simply follow the preceding instructions, add the call to `UseWindowsService()`, and install your ASP.NET Core app. This is thanks to the fact that the ASP.NET Core functionality is built directly on top of the generic Host functionality.

You can follow a similar process to install a worker service as a systemd daemon by installing the `Microsoft.Extensions.Hosting.Systemd` package and calling `UseSystemd()` on your `IHostBuilder`. For more details on configuring systemd, see the "Monitor the app" section in Microsoft's "Host ASP.NET Core on Linux with Nginx" documentation: <http://mng.bz/yYDp>.

So far in this chapter we've used `IHostedService` and the `BackgroundService` to run tasks that repeat on an interval, and you've seen how to install worker services as long-running applications by installing as a Windows Service.

In the final section of this chapter, we'll look at how you can create more advanced schedules for your background tasks, as well as how to add resiliency to your application by running multiple instances of your workers. To achieve that, we'll use a mature third-party library, Quartz.NET.

## 22.3 Coordinating background tasks using Quartz.NET

In this section you'll learn how to use the open source scheduler library Quartz.NET. You'll learn how to install and configure the library, and how to add a background job to run on a schedule. You'll also learn how to enable clustering for your applications, so that you can run multiple instances of your worker service and share jobs between them.

All the background tasks you've seen so far in this chapter repeat a task on an interval indefinitely, from the moment the application starts. However, sometimes you want more control of this timing. Maybe you always want to run the application at 15 minutes past each hour. Or maybe you only want to run a task on the second Tuesday of the month at 3 a.m. Additionally, maybe you want to run multiple instances of your application for redundancy, but ensure that only one of the services runs a task at any one time.

It would certainly be possible to build all this extra functionality into an application yourself, but there are excellent libraries available that already provide all this functionality for you. Two of the most well known in the .NET space are Hangfire ([www.hangfire.io](http://www.hangfire.io)) and Quartz.NET([www.quartz-scheduler.net](http://www.quartz-scheduler.net)).

Hangfire is an open source library that also has a "Pro" subscription option. One of its most popular features is a dashboard user interface that shows the state of all your running jobs, each task's history, and any errors that have occurred.

Quartz.NET is completely open source and essentially offers a beefed-up version of the `BackgroundService` functionality. It has extensive scheduling functionality, as well as support for running in a clustered environment, where multiple instances of your application coordinate to distribute the jobs amongst themselves.

**NOTE** Quartz.NET is based on a similar Java library called Quartz Scheduler. When looking for information on Quartz.NET be sure you're looking at the correct Quartz!

Quartz.NET is based around four main concepts:

- *Jobs*—These are the background tasks that implement your logic.
- *Triggers*—These control *when* a job will run, based on a schedule, such as "every five minutes" or "every second Tuesday." A job can have multiple triggers.
- *Job factory*—The job factory is responsible for creating instances of your jobs. Quartz.NET integrates with ASP.NET Core's DI container, so you can use DI in your job classes.

- *Scheduler*—The Quartz.NET scheduler keeps track of the triggers in your application, creates jobs using the job factory, and runs your jobs. The scheduler typically runs as an `IHostedService` for the lifetime of your app.

### Background services versus cron jobs

It's common to use cron jobs to run tasks on a schedule on Linux, and Windows has similar functionality through the use of Task Scheduler. These are used to periodically run an application or script file, which is typically a short-lived task.

In contrast, .NET Core apps using background services are designed to be long-lived, even if they are only used to run tasks on a schedule. This allows your application to do things like adjust its schedule as required or perform optimizations. In addition, being long-lived means your app doesn't have to just run tasks on a schedule. It can respond to ad hoc events, such as events in a message queue.

Of course, if you don't need those capabilities and would rather not have a long-running application, you can use .NET Core in combination with cron jobs. You could create a simple .NET console app that runs your task and then shuts down, and you could schedule it to execute periodically as a cron job. The choice is yours!

In this section I'll show you how to install Quartz.NET and configure a background service to run on a schedule. I'll then explain how to enable clustering, so that you can run multiple instances of your application and distribute the jobs between them.

#### 22.3.1 Installing Quartz.NET in an ASP.NET Core application

In this section I'll show you how to install the Quartz.NET scheduler into an ASP.NET Core application. Quartz.NET will run in the background in the same way as the `IHostedService` implementations do. In fact, Quartz.NET uses the `IHostedService` abstractions to schedule and run jobs.

**DEFINITION** A *job* in Quartz.NET is a task to be executed that implements the `IJob` interface. It is where you define the logic that your tasks will execute.

Quartz.NET can be installed into any .NET Core application, so you'll also see how to install Quartz.NET into a worker service. You'll install the necessary dependencies and configure the Quartz.NET scheduler to run as a background service in a worker service app. In section 22.3.2 we'll convert the exchange-rate downloader task from section 22.1 to a Quartz.NET `IJob` and configure triggers to run on a schedule.

**NOTE** The instructions in this section can be used to install Quartz.NET into either a worker service or a full ASP.NET Core application. The only difference is whether you use the `ConfigureServices()` method in `Program.cs` or `Startup.cs`.

To install Quartz.NET, follow these steps:

- 1 Install the Quartz.AspNetCore NuGet package in your project by running dotnet add package Quartz.Extensions.Hosting, by using the NuGet explorer in Visual Studio, or by adding a <PackageReference> element to your project file as follows:

```
<PackageReference Include="Quartz.Extensions.Hosting" Version="3.2.3" />
```

- 2 Add the Quartz.NET `IHostedService` scheduler by calling `AddQuartzHostedService()` on the `IServiceCollection` in `ConfigureServices`, as follows. Set `WaitForJobsToComplete=true` so that your app will wait for any jobs in progress to finish when shutting down.

```
services.AddQuartzHostedService(q => q.WaitForJobsToComplete = true);
```

- 3 Configure the required Quartz.NET services in `ConfigureServices`. The example in the following listing configures the Quartz.NET job factory to retrieve job implementations from a scoped DI container, and it adds a required service.

#### Listing 22.8 Configuring Quartz.NET in `ConfigureServices`

```
public void ConfigureServices(IServiceCollection collection)
{
    services.AddQuartz(q => {
        q.UseMicrosoftDependencyInjectionScopedJobFactory();
    });

    services.AddQuartzHostedService(
        q => q.WaitForJobsToComplete = true);
}
```

**Configure Quartz.NET to load jobs from a scoped DI container.**

**Register Quartz.NET services with the DI container.**

**Add the Quartz.NET IHostedService that runs the Quartz.NET scheduler.**

**Configure Quartz.NET to load jobs from a scoped DI container.**

**Add Quartz.NET in `Startup.cs` for ASP.NET Core apps or in `Program.cs` for worker services.**

This configuration registers all Quartz.NET's required components, so you can now run your application using `dotnet run` or by pressing F5 in Visual Studio. When your app starts, the Quartz.NET `IHostedService` starts its scheduler, as shown in figure 22.4. We haven't configured any jobs to run yet, so the scheduler doesn't have anything to schedule.

**TIP** Running your application *before* you've added any jobs is a good practice. It lets you check that you have installed and configured Quartz.NET correctly before you get to more advanced configuration.

A job scheduler without any jobs to schedule isn't a lot of use, so in the next section we'll create a job and add a trigger for it to run on a timer.

**Quartz.NET uses an in-memory store for tracking jobs and schedules by default.**

**Quartz.NET runs in non-clustered mode by default, so each running instance of your app is independent.**

**No jobs or triggers have been configured for this application.**

```

Command Prompt (light) + ×
info: Quartz.Core.SchedulerSignalerImpl[0]
    Initialized Scheduler Signaller of type: Quartz.Core.SchedulerSignalerImpl
info: Quartz.Core.QuartzScheduler[0]
    Quartz Scheduler v3.1.0.0 created.
info: Quartz.Core.QuartzScheduler[0]
    JobFactory set to: Quartz.MicrosoftDependencyInjectionScopedJobFactory
Info: Quartz.Simpl.RAMJobStore[0]
    RAMJobStore initialized.
info: Quartz.Core.QuartzScheduler[0]
    Scheduler meta-data: Quartz Scheduler (v3.1.0.0) 'QuartzScheduler' with instanceId 'NON_CLUSTERED'
    Schedules class: 'Quartz.Core.QuartzScheduler' - running locally.
    NOT STARTED.
    Currently 0 standby threads.
    Number of jobs executing: 0
    Using thread pool 'Quartz.Simpl.DefaultThreadPool' - with 10 threads.
    Using job-store 'Quartz.Simpl.RAMJobStore' - which does not support persistence. and is not clustered.

info: Quartz.Impl.StdSchedulerFactory[0]
    Quartz scheduler 'QuartzScheduler' initialized
info: Quartz.Impl.StdSchedulerFactory[0]
    Quartz scheduler version: 3.1.0.0
info: Quartz.Core.QuartzScheduler[0]
    Add 0 jobs & triggers.
info: Quartz.Core.QuartzScheduler[0]
    Scheduler QuartzScheduler_S_NON_CLUSTERED started.
info: Microsoft.Hosting.Lifetime[0]
    Application started. Press Ctrl+C to shut down.
  
```

Figure 22.4 The Quartz.NET scheduler starts on app startup and logs its configuration. The default configuration stores the list of jobs and their schedules in memory and runs in a non-clustered state. In this example, you can see that no jobs or triggers have been registered, so the scheduler has nothing to schedule yet.

### 22.3.2 Configuring a job to run on a schedule with Quartz.NET

In section 22.1 we created an `IHostedService` that downloads exchange rates from a remote service and saves the results to a database using EF Core. In this section you'll see how you can create a similar Quartz.NET `IJob` and configure it to run on a schedule.

The following listing shows an implementation of `IJob` that downloads the latest exchange rates from a remote API using a typed client, `ExchangeRatesClient`. The results are then saved using an EF Core `DbContext`, `AppDbContext`.

#### Listing 22.9 A Quartz.NET IJob for downloading and saving exchange rates

```

public class UpdateExchangeRatesJob : IJob
{
    private readonly ILogger<UpdateExchangeRatesJob> _logger;
    private readonly ExchangeRatesClient _typedClient;
    private readonly AppDbContext _dbContext;
    public UpdateExchangeRatesJob(
        ILogger<UpdateExchangeRatesJob> logger,
        ExchangeRatesClient typedClient,
        AppDbContext dbContext)
    {
        _logger = logger;
        _typedClient = typedClient;
        _dbContext = dbContext;
    }
    public async Task Execute(IJobExecutionContext context)
    {
        _logger.LogInformation("Fetching latest rates");
    }
}
  
```

IJob requires you to implement a single asynchronous method, Execute.

Quartz.NET jobs must implement the IJob interface.

You can use standard dependency injection to inject any dependencies.

```

Download the rates from the remote API.    var latestRates = await _typedClient.GetLatestRatesAsync();

                                            _dbContext.Add(latestRates);
                                            await _dbContext.SaveChangesAsync(); | Save the rates to the database.

                                            _logger.LogInformation("Latest rates updated");
}

}

```

Functionally, the `IJob` in the listing 22.9 is doing a similar task to the `BackgroundService` implementation in listing 22.4, with a few notable exceptions:

- *The `IJob` only defines the task to execute; it doesn't define timing information.* In the `BackgroundService` implementation, we also had to control how often the task was executed.
- *A new `IJob` instance is created every time the job is executed.* In contrast, the `BackgroundService` implementation is only created once, and its `Execute` method is only invoked once.
- *We can inject scoped dependencies directly into the `IJob` implementation.* To use scoped dependencies in the `IHostedService` implementation, we had to manually create our own scope and use service location to load dependencies. Quartz.NET takes care of that for us, allowing us to use pure constructor injection. Every time the job is executed, a new scope is created and is used to create a new instance of the `IJob`.

The `IJob` defines *what* to execute, but it doesn't define *when* to execute it. For that, Quartz.NET uses *triggers*. Triggers can be used to define arbitrarily complex blocks of time during which a job should be executed. For example, you can specify start and end times, how many times to repeat, and blocks of time when a job should or shouldn't run (such as only 9 a.m. to 5 p.m., Monday–Friday).

In the following listing, we register the `UpdateExchangeRatesJob` with the DI container using the `AddJob<T>()` method, and we provide a unique name to identify the job. We also configure a trigger that fires immediately, and then every five minutes, until the application shuts down.

#### Listing 22.10 Configuring a Quartz.NET `IJob` and trigger

```

public void ConfigureServices(IServiceCollection collection)
{
    services.AddQuartz(q =>
    {
        q.UseMicrosoftDependencyInjectionScopedJobFactory();
    });
}

Add the Ijob to the DI container, and associate it with the job key.    var jobKey = new JobKey("Update exchange rates");
q.AddJob<UpdateExchangeRatesJob>(opts =>
    opts.WithIdentity(jobKey));

```

Create a unique key for the job, used to associate it with a trigger.

```
q.AddTrigger(opts => opts
    .ForJob(jobKey)) | Register a trigger for the Ijob via the job key.
```

```

Fire the trigger
every five minutes
until the app
shuts down.    |
| .WithIdentity(jobKey.Name + " trigger")
| .StartNow()
| .WithSimpleSchedule(x => x
|   .WithInterval(TimeSpan.FromMinutes(5))
|   .RepeatForever())
|
| );
}
}

services.AddQuartzHostedService(q => q.WaitForJobsToComplete = true);
}

```

Provide a unique name for the trigger for use in logging and in clustered scenarios.

Fire the trigger as soon as the Quartz.NET scheduler runs on app startup.

Simple triggers like the schedule defined here are common, but you can also achieve more complex configurations using other schedules. For example, the following configuration would set a trigger to fire every week, on a Friday at 5:30 p.m.:

```

q.AddTrigger(opts => opts
    .ForJob(jobKey)
    .WithIdentity("Update exchange rates trigger")
    .WithSchedule(CronScheduleBuilder
        .WeeklyOnDayAndHourAndMinute(DayOfWeek.Friday, 17, 30))

```

You can configure a wide array of time- and calendar-based triggers with Quartz.NET. You can also control how Quartz.NET handles missed triggers—that is, triggers that should have fired, but your app wasn’t running at the time. For a detailed description of the trigger configuration options and more examples, see the Quartz.NET documentation at [www.quartz-scheduler.net/documentation/](http://www.quartz-scheduler.net/documentation/).

**TIP** A common problem people run into with long-running jobs is that Quartz.NET will keep starting new instances of the job when a trigger fires, even though it’s already running. To avoid that, tell Quartz.NET to not start another instance by decorating your `IJob` implementation with the `[DisallowConcurrentExecution]` attribute.

The ability to configure advanced schedules, the simple use of dependency injection in background tasks, and the separation of jobs from triggers are reasons enough for me to recommend Quartz.NET if you have anything more than the most basic background service needs. However, the real tipping point is when you need to scale your application for redundancy or performance reasons—that’s when Quartz.NET’s clustering capabilities make it shine.

### 22.3.3 Using clustering to add redundancy to your background tasks

In this section you’ll learn how to configure Quartz.NET to persist its configuration to a database. This is a necessary step in enabling clustering, so that multiple instances of your application can coordinate to run your Quartz.NET jobs.

As your applications become more popular, you may find you need to run more instances of your app to handle the traffic they receive. If you keep your ASP.NET Core applications stateless, the process of scaling is relatively simple: the more applications you have, the more traffic you can handle, everything else being equal.

However, scaling applications that use `IHostedService` to run background tasks might *not* be as simple. For example, imagine your application includes the `BackgroundService` that we created in section 22.1.2, which saves exchange rates to the database every five minutes. When you’re running a single instance of your app, the task runs every five minutes as expected.

But what happens if you scale your application and run 10 instances of it? Every one of those applications will be running the `BackgroundService`, and they’ll *all* be updating every five minutes from the time each instance started!

One option would be to move the `BackgroundService` to a separate worker service app. You could then continue to scale your ASP.NET Core application to handle the traffic as required, but deploy a single instance of the worker service. As only a single instance of the `BackgroundService` would be running, the exchange rates would be updated on the correct schedule again.

**TIP** Differing scaling requirements, as in this example, are one of the best reasons for splitting up bigger apps into smaller microservices. Breaking up an app like this has a maintenance overhead, however, so think about the trade-offs if you take this route. For more on this trade-off, I recommend *Microservices in .NET Core*, 2nd ed., by Christian Horsdal Gammelgaard (Manning, 2021).

However, if you take this route, you add a hard limitation that you can only *ever* have a single instance of your worker service. If you need to run more instances of your worker service to handle additional load, you’ll be stuck.

An alternative option to enforcing a single service is to use *clustering*. Clustering allows you to run multiple instances of your application, and tasks are distributed between all the instances of your application. Quartz.NET achieves clustering by using a database as a backing store. When a trigger indicates a job needs to execute, the Quartz.NET schedulers in each app attempt to obtain a lock to execute the job, as shown in figure 22.5. Only a single app can be successful, ensuring that a single app handles the trigger for the `IJob`.

Quartz.NET relies on a persistent database for its clustering functionality. Quartz.NET stores descriptions of the jobs and triggers in the database, including when the trigger last fired. It’s the locking features of the database that ensure only a single application can execute a task at a time.

**TIP** You can enable persistence without enabling clustering. This allows the Quartz.NET scheduler to catch up with missed triggers.

The following listing shows how to enable persistence for Quartz.NET, and, additionally, how to enable clustering. This example stores data in an MS SQL Server (or LocalDB) server, but Quartz.NET supports many other databases. This example uses the recommended values for enabling clustering and persistence as outlined in the documentation.<sup>1</sup>

---

<sup>1</sup> The Quartz.NET documentation discusses many configuration setting controls for persistence. See the documentation for “Job Stores”: <http://mng.bz/PP0R>.

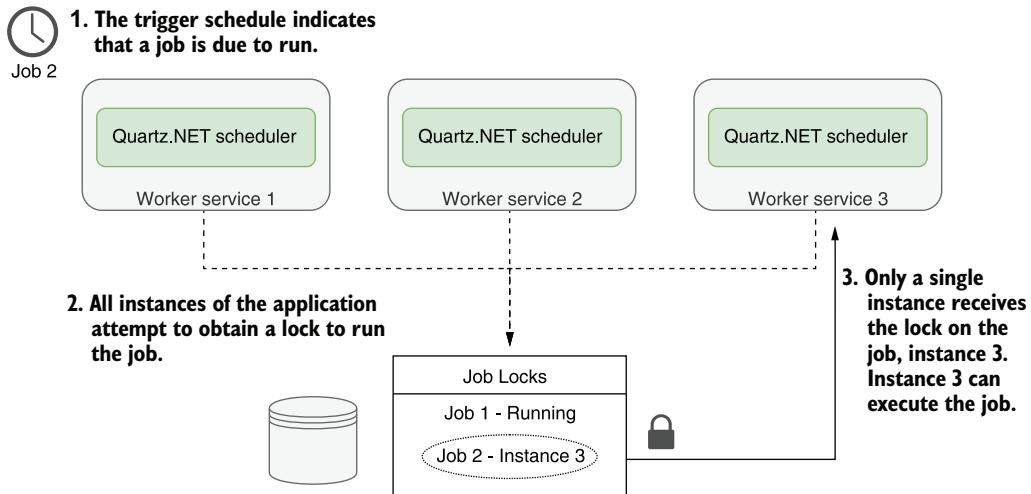


Figure 22.5 Using clustering with Quartz.NET allows horizontal scaling. Quartz.NET uses a database as a backing store, ensuring that only a single instance of the application handles a trigger at a time. This makes it possible to run multiple instances of your application to meet scalability requirements.

#### Listing 22.11 Enabling persistence and clustering for Quartz.NET

```

public void ConfigureServices(IServiceCollection collection)
{
    var connectionString = Configuration
        .GetConnectionString("DefaultConnection");

    services.AddQuartz(q =>
    {
        q.SchedulerId = "AUTO";           ← Configuration is identical
                                         for both ASP.NET Core apps
                                         and worker services.

                                         ← Each instance of your app must
                                         have a unique SchedulerId. AUTO
                                         takes care of this for you.

        q.UseMicrosoftDependencyInjectionScopedJobFactory();

        q.UsePersistentStore(s =>          ← Store the scheduler data in
                                         a SQL Server (or LocalDb)
                                         database.
        {
            s.UseSqlServer(connectionString);
            s.UseClustering();
            s.UseProperties = true;
            s.UseJsonSerializer();
        });
                                         ← Adds the recommended
                                         configuration for job persistence

    });

    var jobKey = new JobKey("Update exchange rates");
    q.AddJob<UpdateExchangeRatesJob>(opts =>
        opts.WithIdentity(jobKey));

    q.AddTrigger(opts => opts
        .ForJob(jobKey)
        .WithIdentity(jobKey.Name + " trigger")
        .StartNow());
}

```

Obtain the connection string for your database from configuration.

Enable database persistence for the Quartz.NET scheduler data.

Enables clustering between multiple instances of your app

```
        .WithSimpleSchedule(x => x
            .WithInterval(TimeSpan.FromMinutes(5))
            .RepeatForever())
    );
}) ;

services.AddQuartzHostedService(q => q.WaitForJobsToComplete = true);
}
```

With this configuration, Quartz.NET stores a list of jobs and triggers in the database and uses database locking to ensure only a single instance of your app handles a trigger and runs the associated job.

**NOTE** SQLite doesn't support the database locking primitives required for clustering. You can use SQLite as a persistence store, but you won't be able to use clustering.

Quartz.NET stores data in your database, but it doesn't attempt to create the tables it uses itself. Instead, you must manually add the required tables. Quartz.NET provides SQL scripts on GitHub for all of the supported database server types, including MS SQL Server, SQLite, PostgreSQL, MySQL, and many more: <http://mng.bz/JDeZ>.

**TIP** If you're using EF Core migrations to manage your database, I suggest using them even for ad hoc scripts like these. In the code sample associated with this chapter, you can see a migration that creates the required tables using the Quartz.NET scripts.

Clustering is one of those advanced features that is only necessary as you start to scale your application, but it's an important tool to have in your belt. It gives you the ability to safely scale your services as you add more jobs. There are some important things to bear in mind, however, so I suggest reading through the warnings in the Quartz.NET documentation: <http://mng.bz/aozj>.

That brings us to the end of this chapter on background services. In the final chapter of this book, I'll describe an important aspect of web development that, sometimes despite the best of intentions, is often left until last: testing. You'll learn how to write simple unit tests for your classes, how to design for testability, and how to build integration tests that test your whole app.

## Summary

- You can use the `IHostedService` interface to run tasks in the background of your ASP.NET Core apps. Call `AddHostedService<T>()` to add an implementation `T` to the DI container. `IHostedService` is useful for implementing long-running tasks.
- Typically, you should derive from `BackgroundService` to create an `IHostedService`, as this implements best practices required for long-running tasks. You must override a single method, `ExecuteAsync`, which is called when your

- app starts. You should run your tasks within this method until the provided `CancellationToken` indicates the app is shutting down.
- You can manually create DI scopes using `IServiceProvider.CreateScope()`. This is useful for accessing *scoped* lifetime services from within a *singleton* lifetime component, such as from an `IHostedService` implementation.
  - A *worker service* is a .NET Core application that uses the generic `IHost` but doesn't include the ASP.NET Core libraries for handling HTTP requests. They generally have a smaller memory and disk footprint than the ASP.NET Core equivalents.
  - Worker services use the same logging, configuration, and dependency injection systems as ASP.NET Core apps. However, they don't use the `Startup.cs` file, so you must configure your DI services in `IHostBuilder.ConfigureServices()`.
  - To run a worker service or ASP.NET Core app as a Windows Service, add the `Microsoft.Extensions.Hosting.WindowsServices` NuGet package, and call `AddWindowsService()` on `IHostBuilder`. You can install and manage your app with the Windows `sc` utility.
  - To install a Linux systemd daemon, add the `Microsoft.Extensions.Hosting.Systemd` NuGet package and call `AddSystemd()` on `IHostBuilder`. Both the Systemd and Windows Service integration packages do nothing when running the application as a console app, which is great for testing your app.
  - Quartz.NET runs jobs based on triggers using advanced schedules. It builds on the `IHostedService` implementation to add extra features and scalability. You can install Quartz by adding the `Quartz.AspNetCore` NuGet package and calling `AddQuartz()` and `AddQuartzHostedService()` in `ConfigureServices()`.
  - You can create a Quartz.NET job by implementing the `IJob` interface. This requires implementing a single method, `Execute`. You can enable DI for the job by calling `UseMicrosoftDependencyInjectionScopedJobFactory` in `AddQuartz()`. This allows you to directly inject scoped (or transient) services into your job without having to create your own scopes.
  - You must register your job, `T`, with DI by calling `AddJob<T>()` and providing a `JobKey` name for the job. You can add an associated trigger by calling `AddTrigger()` and providing the `JobKey`. Triggers have a wide variety of schedules available for controlling when a job should be executed.
  - By default, triggers will continue to spawn new instances of a job as often as necessary. For long-running jobs scheduled with a short interval, that will result in many instances of your app running concurrently. If you only want a trigger to execute a job when an instance is not already running, decorate your job with the `[DisallowConcurrentExecution]` attribute.
  - Quartz.NET supports database persistence for storing when triggers have executed. To enable persistence, call `UsePersistentStore()` in your `AddQuartz()` configuration method, and configure a database, using `UseSqlServer()` for

example. With persistence, Quartz.NET can persist details about jobs and triggers between application restarts.

- Enabling persistence also allows you to use clustering. Clustering enables multiple apps using Quartz.NET to coordinate, so that jobs are spread across multiple schedulers. To enable clustering, first enable database persistence and then call `UseClustering()`.

# *Testing your application*

## **This chapter covers**

- Creating unit test projects with xUnit
- Writing unit tests for custom middleware and API controllers
- Using the Test Host package to write integration tests
- Testing your real application's behavior with `WebApplicationFactory`
- Testing code dependent on EF Core with the in-memory database provider

When I first started programming, I didn't understand the benefits of automated testing. It involved writing so much more code—wouldn't it be more productive to be working on new features instead? It was only when my projects started getting bigger that I appreciated the advantages. Instead of having to manually run my app and test each scenario, I could press Play on a suite of tests and have my code tested for me automatically.

Testing is universally accepted as good practice, but how it fits into your development process can often turn into a religious debate. How many tests do you

need? Is anything less than 100% coverage of your code base adequate? Should you write tests before, during, or after the main code?

This chapter won't address any of those questions. Instead, I'll focus on the *mechanics* of testing an ASP.NET Core application. I'll show you how to use isolated *unit tests* to verify the behavior of your services in isolation, how to test your API controllers and custom middleware, and how to create *integration tests* that exercise multiple components of your application at once. Finally, I'll touch on the EF Core in-memory provider, a feature that lets you test components that depend on a `DbContext` without having to connect to a database.

**TIP** For a broader discussion of testing, or if you're brand new to unit testing, see *The Art of Unit Testing*, 3rd ed., by Roy Osherove (Manning, 2021). If you want to explore unit test best practices using C# examples, see *Unit Testing Principles, Practices, and Patterns* by Vladimir Khorikov (Manning, 2020). Alternatively, for an in-depth look at testing with xUnit in .NET Core, see *.NET Core in Action* by Dustin Metzgar (Manning, 2018).

In section 23.1 I'll introduce the .NET SDK testing framework and how you can use it to create unit testing apps. I'll describe the components involved, including the testing SDK and the testing frameworks themselves, like xUnit and MSTest. Finally, I'll cover some of the terminology I'll use throughout the chapter.

In section 23.2 you'll create your first test project. You'll be testing a simple class at this stage, but it'll allow you to come to grips with the various testing concepts involved. You'll create several tests using the xUnit test framework, make assertions about the behavior of your services, and execute the test project both from Visual Studio and the command line.

In sections 23.3 and 23.4, we'll look at how to test common features of your ASP.NET Core apps: API controllers and custom middleware. I'll show you how to write isolated unit tests for both, much like you would any other service, and I'll point out the tripping points to watch for.

To ensure components work correctly, it's important to test them in isolation. But you also need to test that they work correctly in a middleware pipeline. ASP.NET Core provides a handy `Test Host` package that lets you easily write these *integration tests* for your components. You can even go one step further with the `WebApplicationFactory` helper class, and test that your *app* is working correctly. In section 23.5 you'll see how to use `WebApplicationFactory` to simulate requests to your application and to verify that it generates the correct response.

In the final section of this chapter, I'll demonstrate how to use the SQLite database provider for EF Core with an in-memory database. You can use this provider to test services that depend on an EF Core `DbContext`, without having to use a real database. That avoids the pain of having unknown database infrastructure, of resetting the database between tests, and of different people having slightly different database configurations.

Let's start by looking at the overall testing landscape for ASP.NET Core, the options available to you, and the components involved.

## 23.1 An introduction to testing in ASP.NET Core

In this section you'll learn about the basics of testing in ASP.NET Core. You'll learn about the different types of tests you can write, such as unit tests and integration tests, and why you should write both types. Finally, you'll see how testing fits into ASP.NET Core.

If you have experience building apps with the full .NET Framework or mobile apps with Xamarin, then you might have some experience with unit testing frameworks. If you were building apps in Visual Studio, the steps for creating a test project differed between testing frameworks (xUnit, NUnit, MSTest), and running the tests in Visual Studio often required installing a plugin. Similarly, running tests from the command line varied between frameworks.

With the .NET SDK, testing in ASP.NET Core and .NET Core is now a first-class citizen, on a par with building, restoring packages, and running your application. Just as you can run `dotnet build` to build a project, or `dotnet run` to execute it, you can use `dotnet test` to execute the tests in a test project, regardless of the testing framework used.

The `dotnet test` command uses the underlying .NET SDK to execute the tests for a given project. This is exactly the same as when you run your tests using the Visual Studio test runner, so whichever approach you prefer, the results are the same.

Test projects are console apps that contain a number of *tests*. A test is typically a method that evaluates whether a given class in your app behaves as expected. The test project will typically have dependencies on at least three components:

- The .NET Test SDK
- A unit testing framework, such as xUnit, NUnit, Fixie, or MSTest
- A test-runner adapter for your chosen testing framework, so that you can execute your tests by calling `dotnet test`

These dependencies are normal NuGet packages that you can add to a project, but they allow you to hook in to the `dotnet test` command and the Visual Studio test runner. You'll see an example .csproj file from a test app in the next section.

Typically, a test consists of a method that runs a small piece of your app in isolation and checks that it has the desired behavior. If you were testing a `Calculator` class, you might have a test that checks that passing the values 1 and 2 to the `Add()` method returns the expected result, 3.

You can write lots of small, isolated tests like this for your app's classes to verify that each component is working correctly, independent of any other components. Small isolated tests like these are called *unit tests*.

Using the ASP.NET Core framework, you can build apps that you can easily unit test; you can test some aspects of your controllers in isolation from your action filters and model binding. This is because the framework

- Avoids static types
- Uses interfaces instead of concrete implementations

- Has a highly modular architecture; for example, you can test your controllers in isolation from your action filters and model binding

But just because all your components work correctly independently doesn't mean they'll work when you put them together. For that, you need *integration tests*, which test the interaction between multiple components.

The definition of an integration test is another somewhat contentious issue, but I think of integration tests as any time you're testing multiple components together, or you're testing large vertical slices of your app: testing a user manager class that can save values to a database, for example, or testing that a request made to a health-check endpoint returns the expected response. Integration tests don't necessarily include the *entire* app, but they definitely use more components than unit tests.

**NOTE** I don't cover UI tests which, for example, interact with a browser to provide true end-to-end automated testing. Selenium ([www.seleniumhq.org](http://www.seleniumhq.org)) and Cypress ([www.cypress.io](http://www.cypress.io)) are two of the best known tools for UI testing.

ASP.NET Core has a couple of tricks up its sleeve when it comes to integration testing. You can use the Test Host package to run an in-process ASP.NET Core server, which you can send requests to and inspect the responses. This saves you from the orchestration headache of trying to spin up a web server on a different process, making sure ports are available, and so on, but still allows you to exercise your whole app.

At the other end of the scale, the EF Core SQLite in-memory database provider lets you isolate your tests from the database. Interacting with and configuring a database is often one of the hardest aspects of automating tests, so this provider lets you sidestep the issue entirely. You'll see how to use it in section 23.6.

The easiest way to get to grips with testing is to give it a try, so in the next section you'll create your first test project and use it to write unit tests for a simple custom service.

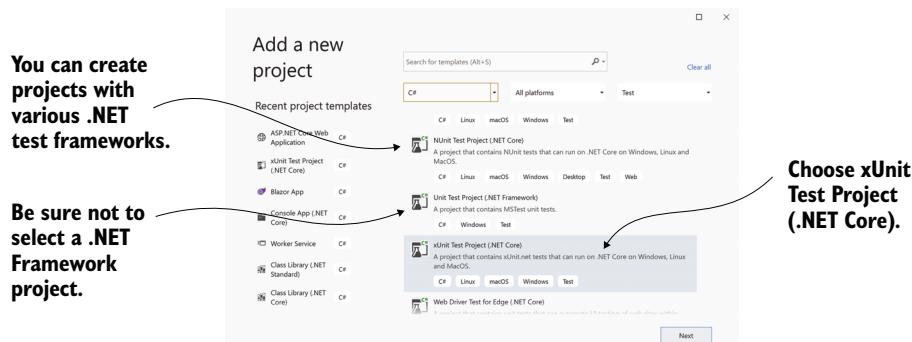
## 23.2 Unit testing with xUnit

In this section you'll learn how to create unit-test projects, how to reference classes in other projects, and how to run tests with Visual Studio or the .NET CLI. You'll create a test project and use it to test the behavior of a basic currency-converter service. You'll write some simple unit tests that check that the service returns the expected results and that it throws exceptions when you expect it to.

As I described in section 23.1, to create a test project you need to use a testing framework. You have many options, such as NUnit or MSTest, but the most commonly used test framework with .NET Core is xUnit (<https://xunit.net/>). The ASP.NET Core framework project itself uses xUnit as its testing framework, so it's become somewhat of a convention. If you're familiar with a different testing framework, then feel free to use that instead.

### 23.2.1 Creating your first test project

Visual Studio includes a template to create a .NET Core xUnit test project, as shown in figure 23.1. Choose File > New Project and choose xUnit Test Project (.NET Core) from the New Project dialog box. Alternatively, you could choose Unit Test Project (.NET Core) or NUnit Test Project (.NET Core) if you're more comfortable with those frameworks.



**Figure 23.1** The New Project dialog box in Visual Studio. Choose xUnit Test Project to create an xUnit project, or choose Unit Test Project to create an MSTest project.

Alternatively, if you're not using Visual Studio, you can create a similar template using the .NET CLI with

```
dotnet new xunit
```

Whether you use Visual Studio or the .NET CLI, the template creates a console project and adds the required testing NuGet packages to your .csproj file, as shown in the following listing. If you chose to create an MSTest (or other framework) test project, the xUnit and xUnit runner packages would be replaced with packages appropriate to your testing framework of choice.

#### Listing 23.1 The .csproj file for an xUnit test project

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net5.0</TargetFramework>
    <IsPackable>false</IsPackable>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference
      Include="Microsoft.NET.Test.Sdk" Version="16.8.0" />
    <PackageReference Include="xunit" Version="2.4.1" />
  </ItemGroup>
</Project>
```

The test project is a standard .NET Core project targeting .NET 5.0.

The .NET Test SDK, required by all test projects

The xUnit test framework

**The xUnit test adapter for the .NET Test SDK**

```

<PackageReference
    Include="xunit.runner.visualstudio" Version="2.4.3" />
<PackageReference Include="coverlet.collector" Version="1.3.0" />
</ItemGroup>
</Project>

```

An optional package that collects metrics about how much of your code base is covered by tests

In addition to the NuGet packages, the template includes a single example unit test. This doesn't *do* anything, but it's a valid xUnit test all the same, as shown in the following listing. In xUnit, a test is a method on a public class, decorated with a `[Fact]` attribute.

**Listing 23.2 An example xUnit unit test, created by the default template**

```

public class UnitTest1
{
    [Fact]
    public void Test1()
    {
        The Fact must be
        public and have no
        parameters.
    }
}

```

xUnit tests must be in public classes.

The `[Fact]` attribute indicates the method is a test method.

Even though this test doesn't test anything, it highlights some characteristics of xUnit `[Fact]` tests:

- Tests are denoted by the `[Fact]` attribute.
- The method should be public, with no method arguments.
- The method is `void`. It could also be an `async` method and return `Task`.
- The method resides inside a public, non-static class.

**NOTE** The `[Fact]` attribute, and these restrictions, are specific to the xUnit testing framework. Other frameworks will use other ways to denote test classes and have different restrictions on the classes and methods themselves.

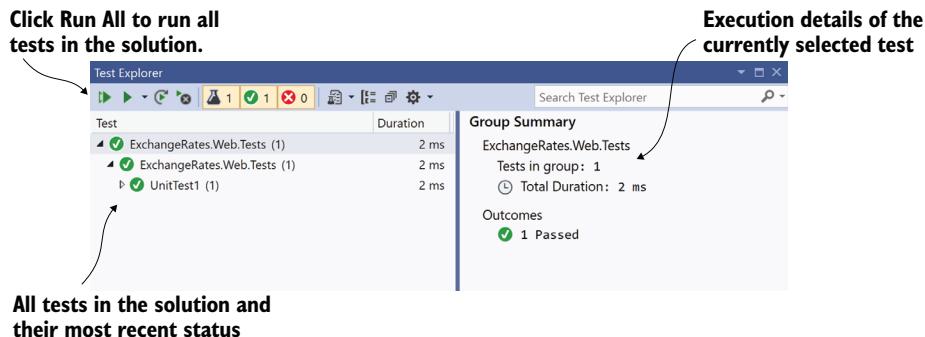
It's also worth noting that, although I said that test projects are console apps, there's no `Program` class or `static void main` method. Instead, the app looks more like a class library. This is because the test SDK automatically injects a `Program` class at build time. It's not something you have to worry about in general, but you may have issues if you try to add your own `Program.cs` file to your test project.<sup>1</sup>

Before we go any further and create some useful tests, we'll run the test project as it is, using both Visual Studio and the .NET SDK tooling, to see the expected output.

<sup>1</sup> This isn't a common thing to do, but I've seen it used occasionally. I describe this issue in detail, and how to fix it, in my blog post, "Fixing the error 'Program has more than one entry point defined' for console apps containing xUnit tests," at <http://mng.bz/w9q5>.

### 23.2.2 Running tests with dotnet test

When you create a test app that uses the .NET Test SDK, you can run your tests either with Visual Studio or using the .NET CLI. In Visual Studio, you run tests by choosing Tests > Run > All Tests from the main menu, or by clicking Run All in the Test Explorer window, as shown in figure 23.2.



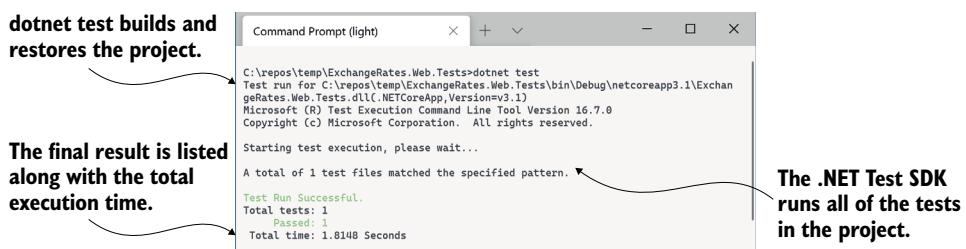
**Figure 23.2** The Test Explorer window in Visual Studio lists all tests found in the solution and their most recent pass/fail status. Click a test in the left pane to see details about the most recent test run in the right pane.

The Test Explorer window lists all the tests found in your solution and the results of each test. In xUnit, a test will pass if it doesn't throw an exception, so `Test1` passed successfully.

Alternatively, you can run your tests from the command line using the .NET CLI by running

```
dotnet test
```

from the unit-test project's folder, as shown in figure 23.3.



**Figure 23.3** You can run tests from the command line using `dotnet test`. This restores and builds the test project before executing all the tests in the project.

**NOTE** You can also run `dotnet test` from the *solution* folder. This will run all test projects referenced in the *.sln* solution file.

Calling `dotnet test` runs a restore and build of your test project and then runs the tests, as you can see from the console output in figure 23.3. Under the hood, the .NET CLI calls into the same underlying infrastructure as Visual Studio does (the .NET SDK), so you can use whichever approach better suits your development style.

You've seen a successful test run, so it's time to replace that placeholder test with something useful. First things first, though; you need something to test.

### 23.2.3 Referencing your app from your test project

In test-driven development (TDD), you typically write your unit tests before you write the actual class you're testing, but I'm going to take a more traditional route here and create the class to test first. You'll write the tests for it afterwards.

Let's assume you've created an app called `ExchangeRates.Web`, which is used to convert between different currencies, and you want to add tests for it. You've added a test project to your solution as described in section 23.2.1, so your solution looks like figure 23.4.

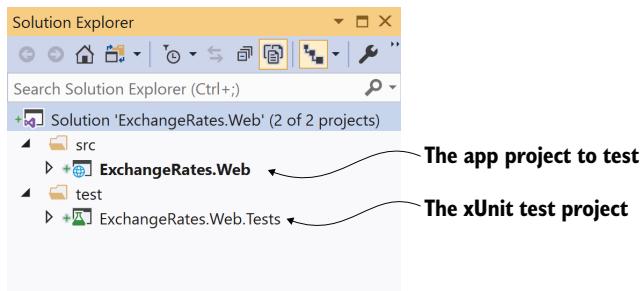


Figure 23.4 A basic solution containing an ASP.NET Core app called `ExchangeRates.Web` and a test project called `ExchangeRates.Web.Tests`.

In order for the `ExchangeRates.Web.Tests` project to be able to test the classes in the `ExchangeRates.Web` project, you need to add a reference to the web project in your test project. In Visual Studio, you can do this by right-clicking the `Dependencies` node of your test project and choosing `Add Reference`, as shown in figure 23.5. You can then select the web project from the `Add Reference` dialog box. After adding it to your project, it shows up inside the `Dependencies` node, under `Projects`.

Alternatively, you can edit the `.csproj` file directly and add a `<ProjectReference>` element inside an `<ItemGroup>` element with the relative path to the referenced project's `.csproj` file.

```
<ItemGroup>
  <ProjectReference
    Include=".\\.\\src\\ExchangeRates.Web\\ExchangeRates.Web.csproj" />
</ItemGroup>
```

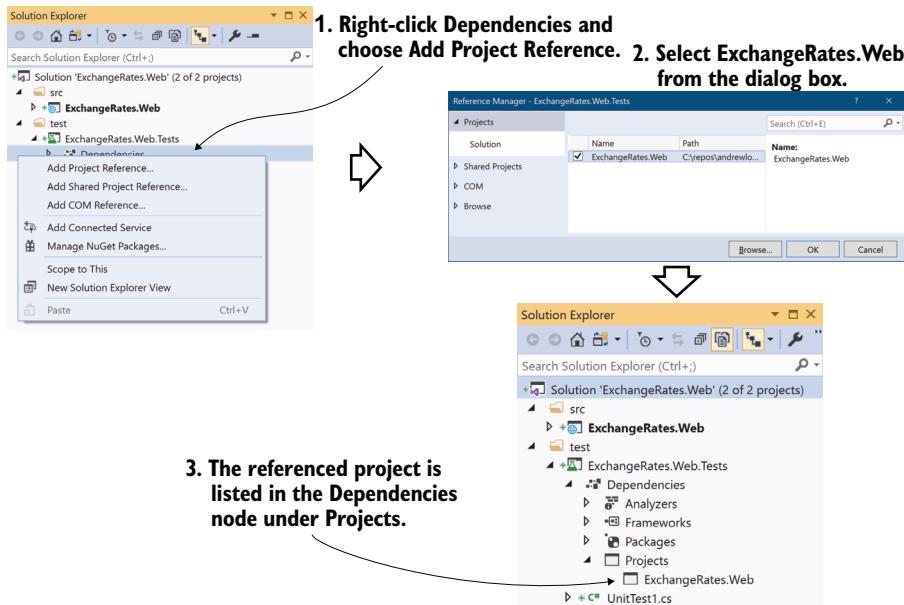


Figure 23.5 To test your app project, you need to add a reference to it from the test project. Right-click the Dependencies node and choose Add Project Reference. The app project is shown referenced inside the Dependencies node, under Projects.

Note that the path is the *relative* path. A "..." in the path means the parent folder, so the relative path shown correctly traverses the directory structure for the solution, including both the src and test folders shown in Solution Explorer in figure 23.5.

**TIP** Remember, you can edit the .csproj file directly in Visual Studio by double-clicking the project in Solution Explorer.

### Common conventions for project layout

The layout and naming of projects within a solution is completely up to you, but ASP.NET Core projects have generally settled on a couple of conventions that differ slightly from the Visual Studio File > New defaults. These conventions are used by the ASP.NET team on GitHub, as well as by many other open source C# projects.

The following figure shows an example of these layout conventions. In summary, these are as follows:

- The .sln solution file is in the root directory.
- The main projects are placed in a src subdirectory.
- The test projects are placed in a test or tests subdirectory.
- Each main project has a test project equivalent, named the same as the associated main project with a ".Test" or ".Tests" suffix.

- Other folders, such as samples, tools, or docs contain sample projects, tools for building the project, or documentation.

The main projects are placed in a `src` subdirectory.

The test projects are placed in a `test` subdirectory.

Test projects match their main project equivalent and have a “`.Tests`” suffix.

Other subdirectories contain samples, tools, or documents, for example.

The solution file is in the root directory.

```
> build
> sample
< src
    > NetEscapades.Configuration.KubeSecrets
    > NetEscapades.Configuration.Remote
    > NetEscapades.Configuration.Yaml
    > VaultSharp
< test
    > NetEscapades.Configuration.KubeSecrets.Tests
    > NetEscapades.Configuration.Remote.Tests
    > NetEscapades.Configuration.Yaml.Tests
< tools
    & .gitattributes
    & .gitignore
    & appveyor.yml
< NetEscapades.Configuration.sln
```

Conventions around project structures have emerged in the ASP.NET Core framework libraries and open source projects on GitHub. You don’t have to follow them for your own project, but it’s worth being aware of them.

Whether or not you choose to follow these conventions is entirely up to you, but it’s good to be aware of them at least, so you can easily navigate other projects on GitHub.

Your test project is now referencing your web project, so you can write tests for classes in the web project. You’re going to be testing a simple class used for converting between currencies, as shown in the following listing.

#### Listing 23.3 Example CurrencyConverter class to convert currencies to GBP

```
public class CurrencyConverter
{
    public decimal ConvertToGbp(
        decimal value, decimal exchangeRate, int decimalPlaces)
    {
        if (exchangeRate <= 0)
        {
            throw new ArgumentException(
                "Exchange rate must be greater than zero",
                nameof(exchangeRate));
        }
    }
}
```

The `ConvertToGbp` method converts a value using the provided exchange rate and rounds it.

Guard clause, as only positive exchange rates are valid

```

    var valueInGbp = value / exchangeRate;
    return decimal.Round(valueInGbp, decimalPlaces);
}
}

```

Rounds the result and returns it

Converts the value

This class only has a single method, `ConvertToGbp()`, which converts a value from one currency into GBP, given the provided `exchangeRate`. It then rounds the value to the required number of decimal places and returns it.

**WARNING** This class is only a basic implementation. In practice, you'd need to handle arithmetic overflow/underflow for large or negative values, as well as considering other edge cases. This is only for demonstration purposes!

Imagine you want to convert 5.27 USD to GBP, and the exchange rate from GBP to USD is 1.31. If you want to round to four decimal places, you'd make this call:

```
converter.ConvertToGbp(value: 5.27, exchangeRate: 1.31, decimalPlaces: 4);
```

You have your sample application, a class to test, and a test project, so it's about time you wrote some tests.

#### 23.2.4 Adding Fact and Theory unit tests

When I write unit tests, I usually target one of three different paths through the method under test:

- *The happy path*—Where typical arguments with expected values are provided
- *The error path*—Where the arguments passed are invalid and tested for
- *Edge cases*—Where the provided arguments are right on the edge of expected values

I realize this is a broad classification, but it helps me think about the various scenarios I need to consider.<sup>2</sup> Let's start with the happy path, by writing a unit test that verifies that the `ConvertToGbp()` method is working as expected with typical input values.

##### Listing 23.4 Unit test for ConvertToGbp using expected arguments

```

[Fact]
public void ConvertToGbp_ConvertsCorrectly()
{
    var converter = new CurrencyConverter();
    decimal value = 3;
    decimal rate = 1.5m;
    int dp = 4;
    decimal expected = 2;
}

```

The result you expect

The [Fact] attribute marks the method as a test method.

You can call the test anything you like.

The class to test, commonly called the “system under test” (SUT)

<sup>2</sup> A whole other way to approach testing is property-based testing. This fascinating approach is common in functional programming communities, like F#. You can find a great introduction by Scott Wlaschin in his blog post, “An introduction to property-based testing”: <http://mng.bz/e5j9>. That post uses F#, but it is still highly accessible even if you're new to the language.

```

    var actual = converter.ConvertToGbp(value, rate, dp);
    Assert.Equal(expected, actual);
}

```

Verifies that the expected and actual values match. If they don't, this will throw an exception.

← Executes the method and captures the result

This is your first proper unit test, which has been configured using the Arrange, Act, Assert (AAA) style:

- *Arrange*—Define all the parameters and create an instance of the system (class) under test (SUT).
- *Act*—Execute the method being tested, and capture the result.
- *Assert*—Verify that the result of the Act stage had the expected value.

Most of the code in this test is standard C#, but if you're new to testing, the `Assert` call will be unfamiliar. This is a helper class provided by xUnit for making assertions about your code. If the parameters provided to `Assert.Equal()` aren't equal, the `Equal()` call will throw an exception and fail the test. If you change the `expected` variable in listing 23.4 to be `2.5` instead of `2`, for example, and run the test, you can see that Test Explorer shows a failure, as in figure 23.6.

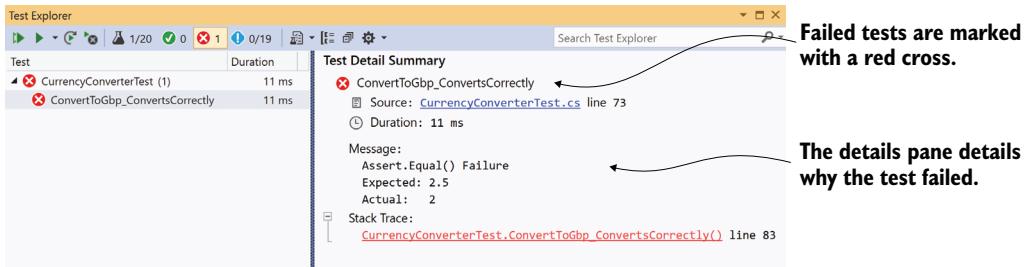


Figure 23.6 When a test fails, it's marked with a red cross in Test Explorer. Clicking the test in the left pane shows the reason for the failure in the right pane. In this case, the expected value was `2.5`, but the actual value was `2`.

**TIP** Alternative assertion libraries such as Fluent Assertions (<https://fluentassertions.com/>) and Shouldly (<https://github.com/shouldly/shouldly>) allow you to write your assertions in a more natural style, such as `actual.Should().Be(expected)`. These libraries are entirely optional, but I find they make tests more readable and error messages easier to understand.

In listing 23.4 you chose specific values for `value`, `exchangeRate`, and `decimalPlaces` to test the happy path. But this is only one set of values in an infinite number of possibilities, so you should probably test at least *a few* different combinations.

One way to achieve this would be to copy and paste the test multiple times, tweak the parameters, and change the test method name to make it unique. xUnit provides an alternative way to achieve the same thing without requiring so much duplication.

**NOTE** The names of your test class and method are used throughout the test framework to describe your test. You can customize how these are displayed in Visual Studio and in the CLI by configuring an `xunit.runner.json` file, as described here: <https://xunit.net/docs/configuration-files>.

Instead of creating a [Fact] test method, you can create a [Theory] test method. A theory provides a way of parameterizing your test methods, effectively taking your test method and running it multiple times with different arguments. Each set of arguments is considered a different test.

You could rewrite the [Fact] test in listing 23.4 to be a [Theory] test, as shown next. Instead of specifying the variables in the method body, pass them as parameters to the method, and then decorate the method with three [InlineData] attributes. Each instance of the attribute provides the parameters for a single run of the test.

#### Listing 23.5 Theory test for `ConvertToGbp` testing multiple sets of values

```
[Theory]           ← Marks the method as a parameterized test
[InlineData(0, 3, 0)]
[InlineData(3, 1.5, 2)]
[InlineData(3.75, 2.5, 1.5)]   ← Each [InlineData] attribute provides all the
                             parameters for a single run of the test method.

public void ConvertToGbp_ConvertsCorrectly (
    decimal value, decimal rate, decimal expected)
{
    var converter = new CurrencyConverter();
    int dps = 4;

    var actual = converter.ConvertToGbp(value, rate, dps);   ← The method takes
                                                               parameters, which
                                                               are provided by the
                                                               [InlineData] attributes.

    Assert.Equal(expected, actual);   ← Verifies the
}                                     result   ← Executes the
                                              system under
                                              test
```

If you run this [Theory] test using `dotnet test` or Visual Studio, it will show up as three separate tests, one for each set of [InlineData], as shown in figure 23.7.

[InlineData] isn't the only way to provide the parameters for your theory tests, but it's one of the most commonly used. You can also use a static property on your test class with the [MemberData] attribute, or a class itself using the [ClassData] attribute.<sup>3</sup>

You now have some tests for the happy path of the `ConvertToGbp()` method, and I even sneaked an edge case into listing 23.5 by testing the case where `value = 0`. The final concept I'll cover is testing error cases, where invalid values are passed to the method under test.

<sup>3</sup> I describe how you can use the [ClassData] and [MemberData] attributes in a blog post, "Creating parameterised tests in xUnit with [InlineData], [ClassData], and [MemberData]": <http://mng.bz/8ayP>.

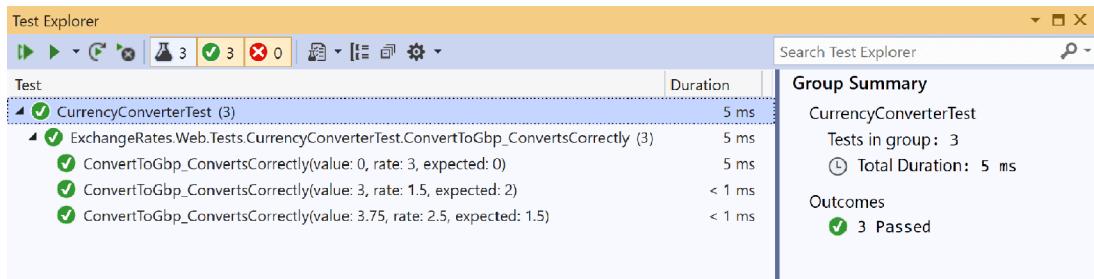


Figure 23.7 Each set of parameters in an `[InlineData]` attribute for a `[Theory]` test creates a separate test run. In this example, a single `[Theory]` has three `[InlineData]` attributes, so it creates three tests, named according to the method name and the provided parameters.

### 23.2.5 Testing failure conditions

A key part of unit testing is checking that the system under test handles edge cases and errors correctly. For the `CurrencyConverter`, that would mean checking how the class handles negative values, small or zero exchange rates, large values and rates, and so on.

Some of these edge cases might be rare but valid cases, whereas other cases might be technically invalid. Calling `ConvertToGbp` with a negative value is probably valid; the converted result should be negative too. A negative exchange rate doesn't make sense conceptually, so it should be considered an invalid value.

Depending on the design of the method, it's common to throw exceptions when invalid values are passed to a method. In listing 23.3 you saw that we throw an `ArgumentException` if the `exchangeRate` parameter is less than or equal to 0.

xUnit includes a variety of helpers on the `Assert` class for testing whether a method throws an exception of an expected type. You can then make further assertions on the exception; for example, to test whether the exception had an expected message.

**WARNING** Take care not to tie your test methods too closely to the internal implementation of a method. Doing so can make your tests brittle, where trivial changes to a class break the unit tests.

The following listing shows a `[Fact]` test to check the behavior of the `ConvertToGbp()` method when you pass it a 0 exchangeRate. The `Assert.Throws` method takes a lambda function that describes the action to execute, which should throw an exception when run.

#### Listing 23.6 Using `Assert.Throws<>` to test whether a method throws an exception

```
[Fact]
public void ThrowsExceptionIfRateIsZero()
{
    var converter = new CurrencyConverter();
    const decimal value = 1;
    const decimal rate = 0;           ← An invalid value
}
```

```

    const int dp = 2;
    var ex = Assert.Throws<ArgumentException>(
        () => converter.ConvertToGbp(value, rate, dp));
    // Further assertions on the exception thrown, ex
}

```

You expect an Argument-Exception to be thrown.

The method to execute, which should throw an exception

The `Assert.Throws` method executes the lambda and catches the exception. If the exception thrown matches the expected type, the test will pass. If no exception is thrown or the exception thrown isn't of the expected type, the `Assert.Throws` method will throw an exception and fail the test.

That brings us to the end of this introduction on unit testing with xUnit. The examples in this section described how to use the new .NET Test SDK, but we didn't cover anything specific to ASP.NET Core. In the rest of this chapter we'll focus on testing ASP.NET Core projects specifically. We'll start by unit testing middleware.

### 23.3 Unit testing custom middleware

In this section you'll learn how to test custom middleware in isolation. You'll see how to test whether your middleware handled a request or whether it called the next middleware in the pipeline. You'll also see how to read the response stream for your middleware.

In chapter 19 you saw how to create custom middleware and how you could encapsulate middleware as a class with an `Invoke` function. In this section you'll create unit tests for a simple health-check middleware component, similar to the one in chapter 19. This is a basic implementation, but it demonstrates the approach you can take for more complex middleware components.

The middleware you'll be testing is shown in listing 23.7. When invoked, this middleware checks that the path starts with `/ping` and, if it does, returns a plain text "pong" response. If the request doesn't match, it calls the next middleware in the pipeline (the provided `RequestDelegate`).

**Listing 23.7 StatusMiddleware to be tested, which returns a "pong" response**

```

public class StatusMiddleware
{
    private readonly RequestDelegate _next;
    public StatusMiddleware(RequestDelegate next)
    {
        _next = next;
    }
    public async Task Invoke(HttpContext context)
    {
        if(context.Request.Path.StartsWithSegments("/ping"))
        {
            context.Response.ContentType = "text/plain";
            await context.Response.WriteAsync("pong");
            return;
        }
    }
}

```

The `RequestDelegate` representing the rest of the middleware pipeline

Called when the middleware is executed

If the path starts with "/ping", a "pong" response is returned.

```

    await _next(context);
}
}

```

Otherwise, the next middleware in the pipeline is invoked.

In this section, you're only going to test two simple cases:

- When a request is made with a path of "/ping"
- When a request is made with a different path

**WARNING** Where possible, I recommend you *don't* directly inspect paths in your middleware like this. A better approach is to use endpoint routing instead, as I discussed in chapter 19. The middleware in this section is for demonstration purposes only.

Middleware is slightly complicated to unit test because the `HttpContext` object is conceptually a *big* class. It contains all the details for the request and the response, which can mean there's a lot of surface area for your middleware to interact with. For that reason, I find unit tests tend to be tightly coupled to the middleware implementation, which is generally undesirable.

For the first test, you'll look at the case where the incoming request Path *doesn't* start with /ping. In this case, `StatusMiddleware` should leave the `HttpContext` unchanged and should call the `RequestDelegate` provided in the constructor, which represents the next middleware in the pipeline.

You could test this behavior in several ways, but in listing 23.8 you test that the `RequestDelegate` (essentially a one-parameter function) is executed by setting a local variable to `true`. In the `Assert` at the end of the method, you verify that the variable was set and therefore that the delegate was invoked. To invoke `StatusMiddleware`, create and pass in a `DefaultHttpContext`,<sup>4</sup> which is an implementation of `HttpContext`.

#### Listing 23.8 Unit testing StatusMiddleware when a nonmatching path is provided

```

[Fact]
public async Task ForNonMatchingRequest_CallsNextDelegate()
{
    var context = new DefaultHttpContext();
    context.Request.Path = "/somethingelse";
    var wasExecuted = false;
    RequestDelegate next = (HttpContext ctx) =>
    {
        wasExecuted = true;
        return Task.CompletedTask;
    };
    var middleware = new StatusMiddleware(next);

```

Creates a `DefaultHttpContext` and sets the path for the request

The `RequestDelegate` representing the next middleware should be invoked in this example.

Creates an instance of the middleware, passing in the next `RequestDelegate`

Tracks whether the `RequestDelegate` was executed

<sup>4</sup> The `DefaultHttpContext` derives from `HttpContext` and is part of the base ASP.NET Core framework abstractions. If you're so inclined, you can explore the source code for it on GitHub at <http://mng.bz/q9qx>.

```
    await middleware.Invoke(context);           ← Invokes the middleware  
    Assert.IsTrue(wasExecuted);                 ← with the HttpContext; should  
}                                            invoke the RequestDelegate  
  
Verifies RequestDelegate  
was invoked
```

When the middleware is invoked, it checks the provided Path and finds that it doesn't match the required value of /ping. The middleware therefore calls the next Request-Delegate and returns.

The other obvious case to test is when the request Path is "/ping"; the middleware should generate an appropriate response. You could test several different characteristics of the response:

- The response should have a 200 OK status code.
  - The response should have a Content-Type of text/plain.
  - The response body should contain the "pong" string.

Each of these characteristics represents a different requirement, so you'd typically codify each as a separate unit test. This makes it easier to tell exactly which requirement hasn't been met when a test fails. For simplicity, in listing 23.9 I show all these assertions in the same test.

The positive case unit test is made more complex by the need to read the response body to confirm it contains "pong". DefaultHttpContext uses Stream.Null for the Response.Body object, which means anything written to Body is lost. To capture the response and read it out to verify the contents, you must replace the Body with a MemoryStream. After the middleware executes, you can use a StreamReader to read the contents of the MemoryStream into a string and verify it.

### **Listing 23.9 Unit testing StatusMiddleware when a matching Path is provided**

```
[Fact]
public async Task ReturnsPongBodyContent()
{
    var bodyStream = new MemoryStream();
    var context = new DefaultHttpContext();
    context.Response.Body = bodyStream;
    context.Request.Path = "/ping";
    RequestDelegate next = (ctx) => Task.CompletedTask;
    var middleware = new StatusMiddleware(next: next);

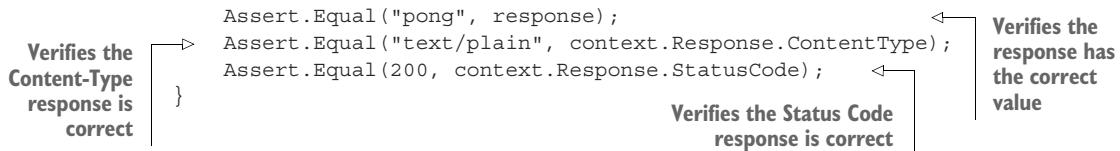
    await middleware.Invoke(context);

    string response;
    bodyStream.Seek(0, SeekOrigin.Begin);
    using (var stringReader = new StreamReader(bodyStream))
    {
        response = await stringReader.ReadToEndAsync();
    }
}
```

Creates a DefaultHttpContext and initializes the body with a MemoryStream to capture the response

Creates an instance of the middleware and passes in a simple RequestDelegate

Rewinds the MemoryStream and reads the response body into a string



As you can see, unit testing middleware requires a lot of setup to get it working. On the positive side, it allows you to test your middleware in isolation, but in some cases, especially for simple middleware without any dependencies on databases or other services, integration testing can (somewhat surprisingly) be easier. In section 23.5 you'll create integration tests for this middleware to see the difference.

Custom middleware is common in ASP.NET Core projects, but far more common are Razor Pages and API controllers. In the next section you'll see how you can unit test them in isolation from other components.

## 23.4 Unit testing API controllers

In this section you'll learn how to unit test API controllers. You'll learn about the benefits and difficulties of testing these components in isolation, and the situations when it can be useful.

Unit tests are all about isolating behavior; you want to test only the logic contained in the component itself, separate from the behavior of any dependencies. The Razor Pages and MVC/API *frameworks* use the filter pipeline, routing, and model-binding systems, but these are all *external* to the controller or PageModels. The PageModels and controllers themselves are responsible for only a limited number of things. Typically,

- For invalid requests (that have failed validation, for example), return an appropriate ActionResult (API controllers) or redisplay a form (Razor Pages).
- For valid requests, call the required business logic services and return an appropriate ActionResult (API controllers), or show or redirect to a success page (Razor Pages).
- Optionally, apply resource-based authorization as required.

Controllers and Razor Pages generally shouldn't contain business logic themselves; instead, they should call out to other services. Think of them more as orchestrators, serving as the intermediary between the HTTP interfaces your app exposes and your business logic services.

If you follow this separation, you'll find it easier to write unit tests for your business logic, and you'll benefit from greater flexibility when you want to change your controllers to meet your needs. With that in mind, there's often a drive to make your controllers and page handlers as thin as possible,<sup>5</sup> to the point where there's not much left to test!

---

<sup>5</sup> One of my first introductions to this idea was a series of posts by Jimmy Bogard. The following link points to the last post in the series, but it contains links to all the earlier posts too. Jimmy Bogard is also behind the MediatR library (<https://github.com/jbogard/MediatR>), which makes creating thin controllers even easier. See “Put your controllers on a diet: POSTs and commands”: <http://mng.bz/7VNQ>.

All that said, controllers and actions are classes and methods, so you *can* write unit tests for them. The difficulty is deciding what you want to test. As an example, we'll consider the simple API controller in the following listing, which converts a value using a provided exchange rate and returns a response.

### Listing 23.10 The API controller under test

```
[Route("api/[controller]")]
public class CurrencyController : ControllerBase
{
    private readonly CurrencyConverter _converter
        = new CurrencyConverter();

    [HttpPost]
    public ActionResult<decimal> Convert(InputModel model)
    {
        if (!ModelState.IsValid)
        {
            return BadRequest(ModelState);
        }

        decimal result = _convert.ConvertToGbp(model);

        return result;
    }
}
```

**The Convert method returns an ActionResult<T>.**

**The CurrencyConverter would normally be injected using DI. Created here for simplicity**

**If the input is invalid, returns a 400 Bad Request result, including the ModelState.**

**If the model is valid, calculate the result.**

**Return the result directly.**

Let's first consider the happy path, when the controller receives a valid request. The following listing shows that you can create an instance of the API controller, call an action method, and you'll receive an `ActionResult<T>` response.

### Listing 23.11 A simple API controller unit test

```
public class CurrencyControllerTest
{
    [Fact]
    public void Convert_ReturnsValue()
    {
        var controller = new CurrencyController();
        var model = new ConvertInputModel
        {
            Value = 1,
            ExchangeRate = 3,
            DecimalPlaces = 2,
        };

        ActionResult<decimal> result = controller.Convert(model);
        Assert.NotNull(result);
    }
}
```

**Creates an instance of the ConvertController to test and a model to send to the API**

**Asserts that the ActionResult is a ViewResult**

**Invokes the ConvertToGbp method and captures the value returned**

An important point to note here is that you're *only* testing the return value of the action, the `ActionResult<T>`, *not* the response that's sent back to the user. The process of serializing the result to the response is handled by the MVC formatter infrastructure, as you saw in chapter 9, not by the controller.

When you unit test controllers, you're testing them *separately* from the MVC infrastructure, such as formatting, model-binding, routing, and authentication. This is obviously by design, but as with testing middleware in section 23.3, it can make testing some aspects of your controller somewhat complex.

Consider model validation. As you saw in chapter 6, one of the key responsibilities of action methods and Razor Page handlers is to check the `ModelState.IsValid` property and act accordingly if a binding model is invalid. Testing that your controllers and `PageModels` correctly handle validation failures seems like a good candidate for a unit test.

Unfortunately, things aren't simple here either. The Razor Page/MVC framework automatically sets the `ModelState` property as part of the model-binding process. In practice, when your action method or page handler is invoked in your running app, you know that the `ModelState` will match the binding model values. But in a unit test, there's no model-binding, so you must set the `ModelState` yourself manually.

Imagine you're interested in testing the error path for the controller in listing 23.10, where the model is invalid and the controller should return `BadRequestObjectResult`. In a unit test, you can't rely on the `ModelState` property being correct for the binding model. Instead, you must *manually* add a model-binding error to the controller's `ModelState` before calling the action, as shown here.

#### Listing 23.12 Testing handling of validation errors in MVC controllers

```
[Fact]
public void Convert_ReturnsBadRequestWhenInvalid()
{
    var controller = new CurrencyController();
    var model = new ConvertInputModel
    {
        Value = 1,
        ExchangeRate = -2,
        DecimalPlaces = 2,
    };

    controller.ModelState.AddModelError(
        nameof(model.ExchangeRate),
        "Exchange rate must be greater than zero"
    );

    ActionResult<decimal> result = controller.Convert(model);
    Assert.IsType<BadRequestObjectResult>(result.Result);
}
```

Creates an instance of the Controller to test

Creates an invalid binding model by using a negative ExchangeRate

Manually adds a model error to the Controller's ModelState. This sets ModelState.IsValid to false.

Invokes the action method, passing in the binding models

Verifies the action method returned a BadRequestObjectResult

**NOTE** In listing 23.12, I passed in an invalid model, but I could just as easily have passed in a *valid* model, or even `null`; the controller doesn’t use the binding model if the `ModelState` isn’t valid, so the test would still pass. But if you’re writing unit tests like this one, I recommend trying to keep your model consistent with your `ModelState`; otherwise your unit tests aren’t testing a situation that occurs in practice.

Personally, I tend to shy away from unit testing API controllers directly in this way.<sup>6</sup> As you’ve seen with model binding, the controllers are somewhat dependent on earlier stages of the MVC framework, which you often need to emulate. Similarly, if your controllers access the `HttpContext` (available on the `ControllerBase` base classes), you may need to perform additional setup.

**NOTE** I haven’t discussed Razor Pages much in this section, as they suffer from many of the same problems, in that they are dependent on the supporting infrastructure of the framework. Nevertheless, if you do wish to test your Razor Page `PageModel`, you can read about it in Microsoft’s “Razor Pages unit tests in ASP.NET Core” documentation: <http://mng.bz/GxmM>.

Instead of using unit testing, I try to keep my controllers and Razor Pages as “thin” as possible. I push as much of the behavior in these classes into business logic services that can be easily unit tested, or into middleware and filters, which can be more easily tested independently.

**NOTE** This is a personal preference. Some people like to get as close to 100% test coverage for their code base as possible, but I find testing “orchestration” classes is often more hassle than it’s worth.

Although I often forgo *unit* testing controllers and Razor Pages, I often write *integration* tests that test them in the context of a complete application. In the next section, we’ll look at ways to write integration tests for your app, so you can test its various components in the context of the ASP.NET Core framework as a whole.

## 23.5 Integration testing: Testing your whole app in-memory

In this section you’ll learn how to create integration tests that test component interactions. You’ll learn to create a `TestServer` that sends HTTP requests in-memory to test custom middleware components more easily. You’ll then learn how to run integration tests for a real application, using your real app’s configuration, services, and middleware pipeline. Finally, you’ll learn how to use `WebApplicationFactory` to replace services in your app with test versions, to avoid depending on third-party APIs in your tests.

If you search the internet for the different types of testing, you’ll find a host of different types to choose from. The differences between them are sometimes subtle, and

---

<sup>6</sup> You can read more about why I generally don’t unit test my controllers in my blog article, “Should you unit-test API/MVC controllers in ASP.NET Core?”: <http://mng.bz/YqMo>.

people don't universally agree upon the definitions. I chose not to dwell on it in this book—I consider unit tests to be isolated tests of a component and integration tests to be tests that exercise multiple components at once.

In this section I'm going to show how you can write integration tests for the StatusMiddleware from section 23.3 and the API controller from section 23.4. Instead of isolating the components from the surrounding framework and invoking them directly, you'll specifically test them in a context similar to how you use them in practice.

Integration tests are an important part of confirming that your components function correctly, but they don't remove the need for unit tests. Unit tests are excellent for testing small pieces of logic contained in your components and are typically quick to execute. Integration tests are normally significantly slower, as they require much more configuration and may rely on external infrastructure, such as a database.

Consequently, it's normal to have far more unit tests for an app than integration tests. As you saw in section 23.2, unit tests typically verify the behavior of a component, using valid inputs, edge cases, and invalid inputs to ensure that the component behaves correctly in all cases. Once you have an extensive suite of unit tests, you'll likely only need a few integration tests to be confident your application is working correctly.

You could write many different types of integration tests for an application. You could test that a service can write to a database correctly, that it can integrate with a third-party service (for sending emails, for example), or that it can handle HTTP requests made to it.

In this section we're going to focus on the last point, verifying that your app can handle requests made to it, just as it would if you were accessing the app from a browser. For this, we're going to use a useful library provided by the ASP.NET Core team called Microsoft.AspNetCore.TestHost.

### 23.5.1 Creating a TestServer using the Test Host package

Imagine you want to write some integration tests for the StatusMiddleware from section 23.3. You've already written unit tests for it, but you want to have at least one integration test that tests the middleware in the context of the ASP.NET Core infrastructure.

You could go about this in many ways. Perhaps the most complete approach would be to create a separate project and configure StatusMiddleware as the only middleware in the pipeline. You'd then need to run this project, wait for it to start up, send requests to it, and inspect the responses.

This would possibly make for a good test, but it would also require a lot of configuration, and it would be fragile and error prone. What if the test app can't start because it tries to use an already-taken port? What if the test app doesn't shut down correctly? How long should the integration test wait for the app to start?

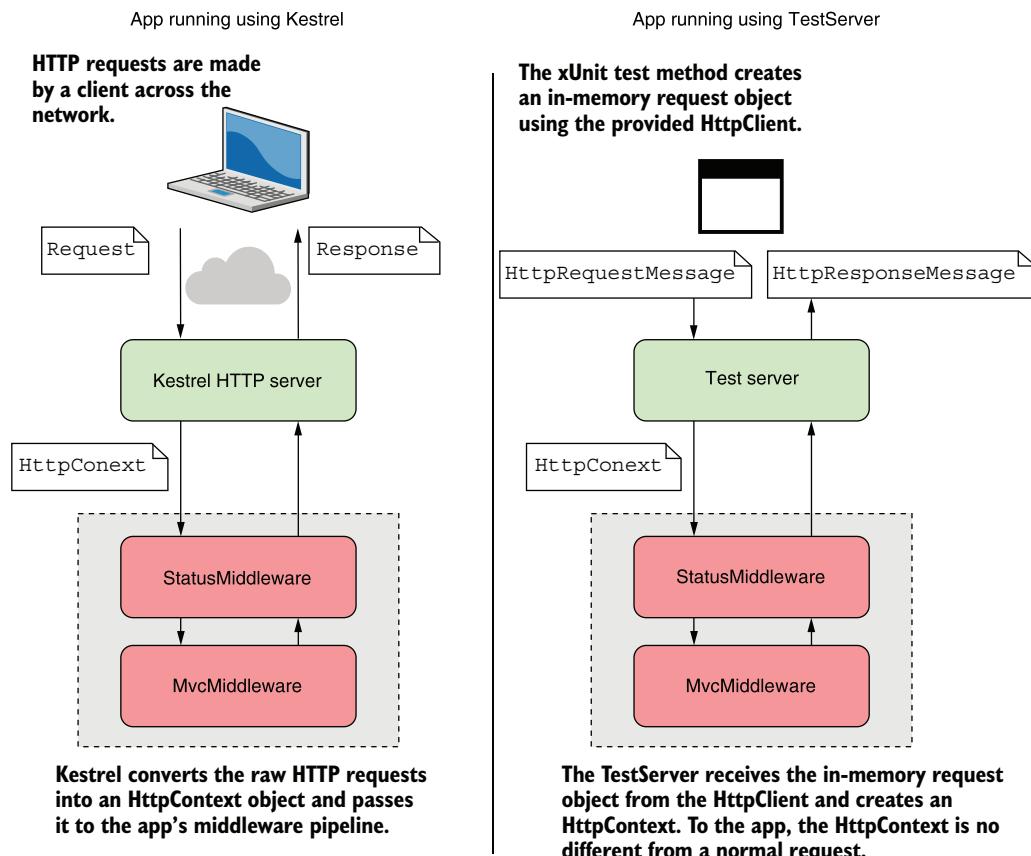
The ASP.NET Core Test Host package lets you get close to this setup without having the added complexity of spinning up a separate app. You add the Test Host to your test project by adding the Microsoft.AspNetCore.TestHost NuGet package, either

using the Visual Studio NuGet GUI, Package Manager Console, or .NET CLI. Alternatively, add the `<PackageReference>` element directly to your test project's `.csproj` file:

```
<PackageReference Include="Microsoft.AspNetCore.TestHost" Version="5.0.0" />
```

In a typical ASP.NET Core app, you create a `HostBuilder` in your `Program` class, configure a web server (Kestrel), and define your application's configuration, services, and middleware pipeline (using a `Startup` file). Finally, you call `Build()` on the `HostBuilder` to create an instance of an `IHost` that can be run and that will listen for requests on a given URL and port.

The Test Host package uses the same `HostBuilder` to define your test application, but instead of listening for requests at the network level, it creates an `IHost` that uses in-memory request objects instead, as shown in figure 23.8. It even exposes an



**Figure 23.8** When your app runs normally, it uses the Kestrel server. This listens for HTTP requests and converts the requests into an `HttpContext`, which is passed to the middleware pipeline. The `TestServer` doesn't listen for requests on the network. Instead, you use an `HttpClient` to make in-memory requests. From the point of view of the middleware, there's no difference.

`HttpClient` that you can use to send requests to the test app. You can interact with the `HttpClient` as though it were sending requests over the network, but in reality the requests are kept entirely in memory.

Listing 23.13 shows how to use the `Test Host` package to create a simple integration test for the `StatusMiddleware`. First, create a `HostBuilder` and call `ConfigureWebHost()` to define your application by adding middleware in the `Configure` method. This is equivalent to the `Startup.Configure()` method you would typically use to configure your application.

Call the `UseTestServer()` extension method in `ConfigureWebHost()`, which replaces the default Kestrel server with the `TestServer` from the `Test Host` package. The `TestServer` is the main component in the `Test Host` package, which makes all the magic possible. After configuring the `HostBuilder`, call `StartAsync()` to build and start the test application. You can then create an `HttpClient` using the extension method `GetTestClient()`. This returns an `HttpClient` configured to make in-memory requests to the `TestServer`.

### Listing 23.13 Creating an integration test with `TestServer`

```
public class StatusMiddlewareTests
{
    [Fact]
    public async Task StatusMiddlewareReturnsPong()
    {
        var hostBuilder = new HostBuilder()
            .ConfigureWebHost(webHost =>
        {
            webHost.Configure(app =>
                app.UseMiddleware<StatusMiddleware>();
            webHost.UseTestServer();
        });
    }

    Add the Status-Middleware as the only middleware in the pipeline. | Configures a Host-Builder to define the in-memory test app
    Build and start the host. | Configure the host to use the TestServer instead of Kestrel.
    Makes an in-memory request, which is handled by the app as normal | Creates an HttpClient, or you can interact directly with the server object
    | Verifies the response was a success (2xx) status code
    | Reads the body content and verifies that it contained "pong"
    }
}
```

This test ensures that the test application defined by `HostBuilder` returns the expected value when it receives a request to the `/ping` path. The request is entirely in-memory, but from the point of view of `StatusMiddleware`, it's the same as if the request came from the network.

The `HostBuilder` configuration in this example is simple. Even though I've called this an integration test, you're specifically testing the `StatusMiddleware` on its own,

rather than in the context of a “real” application. In many ways, I think this setup is preferable for testing custom middleware compared to the “proper” unit tests I showed in section 23.3.

Regardless of what you call it, this test relies on very simple configuration for the test app. You may also want to test the middleware in the context of your *real* application, so that the result is representative of your app’s *real* configuration.

If you want to run integration tests based on an existing app, then you won’t want to have to configure the test HostBuilder manually like you did in listing 23.13. Instead, you can use another helper package, Microsoft.AspNetCore.Mvc.Testing.

### 23.5.2 Testing your application with WebApplicationFactory

Building up a HostBuilder and using the Test Host package, as you did in section 23.5.1, can be useful when you want to test isolated “infrastructure” components, such as middleware. It’s also very common to want to test your “real” app, with the full middleware pipeline configured and all the required services added to DI. This gives you the most confidence that your application is going to work in production.

The TestServer that provides the in-memory server can be used for testing your real app, but, in principle, there’s a lot more configuration required. Your real app likely loads configuration files or static files, and it may use Razor Pages and views. Prior to .NET Core 2.1, configuring all of these was cumbersome. Thankfully, the introduction of the Microsoft.AspNetCore.Mvc.Testing package and WebApplicationFactory largely solves these configuration issues for you.

You can use the WebApplicationFactory class (provided by the Microsoft.AspNetCore.Mvc.Testing NuGet package) to run an in-memory version of your real application. It uses the TestServer behind the scenes, but it uses your app’s real configuration, DI service registration, and middleware pipeline. For example, the following listing shows an example that tests that when your application receives a “/ping” request, it responds with “pong”.

**Listing 23.14 Creating an integration test with WebApplicationFactory**

```
public class IntegrationTests : IClassFixture<WebApplicationFactory<Startup>>
{
    private readonly WebApplicationFactory<Startup> _fixture;
    public IntegrationTests(
        WebApplicationFactory<Startup> fixture)
    {
        _fixture = fixture;
    }

    [Fact]
    public async Task PingRequest_ReturnsPong()
    {
        HttpClient client = _fixture.CreateClient();
```

The code is annotated with three callout boxes:

- A box on the right side of the first two lines of code contains the text: "Your test must implement the interface, though there are no methods to implement." A vertical line connects this box to the start of the code block.
- A box on the right side of the constructor line contains the text: "Inject an instance of WebApplicationFactory<T>, where T is a class in your app." A vertical line connects this box to the "fixture" parameter in the constructor.
- A box on the right side of the final line contains the text: "Create an HttpClient that sends requests to the in-memory TestServer." A vertical line connects this box to the "client" variable in the final line.

```

        var response = await client.GetAsync("/ping");

        response.EnsureSuccessStatusCode();
        var content = await response.Content.ReadAsStringAsync();
        Assert.Equal("pong", content);
    }
}

```

**Make requests and verify the response as before.**

One of the advantages of using `WebApplicationFactory` as shown in listing 23.14 is that it requires *less* manual configuration than using the `TestServer` directly, as shown in listing 23.13, despite performing *more* configuration behind the scenes. The `WebApplicationFactory` tests your app using the configuration defined in your `Program.cs` and `Startup.cs` files.

Listings 23.14 and 23.13 are *conceptually* quite different too. Listing 23.13 tests that the `StatusMiddleware` behaves as expected in the context of a dummy ASP.NET Core app; listing 23.14 tests that *your app behaves as expected for a given input*. It doesn't say anything specific about *how* that happens. Your app doesn't have to use the `StatusMiddleware` for the test in listing 23.14 to pass; it just has to respond correctly to the given request. That means the test knows less about the internal implementation details of your app and is only concerned with its *behavior*.

**DEFINITION** Tests that fail whenever you change your app slightly are called *brittle* or *fragile*. Try to avoid brittle tests by ensuring they aren't dependent on the implementation details of your app.

To create tests that use `WebApplicationFactory`:

- 1 Install the `Microsoft.AspNetCore.Mvc.Testing` NuGet package in your project by running `dotnet add package Microsoft.AspNetCore.Mvc.Testing`, by using the NuGet explorer in Visual Studio, or by adding a `<PackageReference>` element to your project file as follows:

```
<PackageReference Include="Microsoft.AspNetCore.Mvc.Testing"
Version="5.0.0" />
```

- 2 Update the `<Project>` element in your test project's `.csproj` file to the following:

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

This is required by `WebApplicationFactory` so that it can find your configuration files and static files.

- 3 Implement `IClassFixture<WebApplicationFactory<T>>` in your xUnit test class, where `T` is a class in your real application's project. By convention, you typically use your application's `Startup` class for `T`.
  - `WebApplicationFactory` uses the `T` reference to find your application's `Program.CreateHostBuilder()` method to build an appropriate `TestServer` for tests.

- The `IClassFixture<TestFixture>` is an xUnit marker interface that tells xUnit to build an instance of `TestFixture` before building the test class and to inject the instance into the test class's constructor. You can read more about fixtures at <https://xunit.net/docs/shared-context>.
- 4 Accept an instance of `WebApplicationFactory<T>` in your test class's constructor. You can use this fixture to create an `HttpClient` for sending in-memory requests to the `TestServer`. Those requests emulate your application's production behavior, as your application's real configuration, services, and middleware are all used.

The big advantage of `WebApplicationFactory` is that you can easily test your real app's behavior. That power comes with responsibility—your app will behave just as it would in real life, so it will write to a database and send to third-party APIs! Depending on what you're testing, you may want to replace some of your dependencies to avoid this, as well as to make testing easier.

### 23.5.3 Replacing dependencies in `WebApplicationFactory`

When you use `WebApplicationFactory` to run integration tests on your app, your app will be running in-memory, but other than that, it's as though you're running your application using `dotnet run`. That means any connection strings, secrets, or API keys that can be loaded locally will also be used to run your application.

**TIP** By default, `WebApplicationFactory` uses the "Development" hosting environment, the same as when you run locally.

On the plus side, that means you have a genuine test that your application can start correctly. For example, if you've forgotten to register a required DI dependency that is detected on application startup, any tests that use `WebApplicationFactory` will fail.

On the downside, that means all your tests will be using the same database connection and services as when you run your application locally. It's common to want to replace those with alternative "test" versions of your services.

As a simple example, let's imagine the `CurrencyConverter` that you've been testing in this app uses `IHttpClientFactory` to call a third-party API to retrieve the latest exchange rates. You don't want to hit that API repeatedly in your integration tests, so you want to replace the `CurrencyConverter` with your own `StubCurrencyConverter`.

The first step is to ensure the service `CurrencyConverter` implements an interface, `ICurrencyConverter` for example, and that your app uses this interface throughout, not the implementation. For our simple example, the interface would probably look like the following:

```
public interface ICurrencyConverter
{
    decimal ConvertToGbp(decimal value, decimal rate, int dps);
}
```

You would register the service in `Startup.ConfigureServices()` using

```
services.AddScoped<ICurrencyConverter, CurrencyConverter>() ;
```

Now that your application only indirectly depends on `CurrencyConverter`, you can provide an alternative implementation in your tests.

**TIP** Using an interface decouples your application services from a specific implementation, allowing you to substitute alternative implementations. This is a key practice for making classes testable.

We'll create a simple alternative implementation of `ICurrencyConverter` for our tests that always returns the same value, 3. It's obviously not very useful as an actual converter, but that's not the point: you have complete control! Create the following class in your test project:

```
public class StubCurrencyConverter : ICurrencyConverter
{
    public decimal ConvertToGbp(decimal value, decimal rate, int dps)
    {
        return 3;
    }
}
```

You now have all the pieces you need to replace the implementation in your tests. To achieve that, we'll use a feature of `WebApplicationFactory` that lets you customize the DI container before starting the test server.

**TIP** It's important to remember you *only* want to replace the implementation when running in the test project. I've seen some people try to configure their *real* apps to replace live services for fake services when a specific value is set, for example. That is generally unnecessary, bloats your apps with test services, and generally adds confusion!

`WebApplicationFactory` exposes a method, `WithWebHostBuilder`, that allows you to customize your application before the in-memory `TestServer` starts. The following listing shows an integration test that uses this builder to replace the "default" `ICurrencyConverter` implementation with our test stub.

#### Listing 23.15 Replacing a dependency in a test using `WithWebHostBuilder`

```
public class IntegrationTests :
    IClassFixture<WebApplicationFactory<Startup>>
{
    private readonly WebApplicationFactory<Startup> _fixture;
    public IntegrationTests(WebApplicationFactory<Startup> fixture)
    {
        _fixture = fixture;
    }
}
```



Implement the required interface, and inject it into the constructor.

```
[Fact]
public async Task ConvertReturnsExpectedValue()
{
    var customFactory = _fixture.WithWebHostBuilder(
        (IWebHostBuilder hostBuilder) =>
    {
        hostBuilder.ConfigureTestServices(services =>
        {
            services.RemoveAll<ICurrencyConverter>();
            services.AddSingleton<ICurrencyConverter, StubCurrencyConverter>();
        });
    });

    HttpClient client = customFactory.CreateClient();

    var response = await client.GetAsync("/api/currency");
    response.EnsureSuccessStatusCode();
    var content = await response.Content.ReadAsStringAsync();

    Assert.Equal("3", content);
}
}
```

**Configure-TestServices executes after all other DI services are configured in your real app.**

**Removes all implementations of ICurrencyConverter from the DI container**

**Invoke the currency converter endpoint.**

**Create a custom factory with the additional configuration.**

**Adds the test service as a replacement**

**Calling CreateClient bootstraps the application and starts the TestServer.**

**As the test converter always returns 3, so does the API endpoint.**

There are a couple of important points to note in this example:

- `WithWebHostBuilder()` returns a *new* `WebApplicationFactory` instance. The new instance has your custom configuration, while the original injected `_fixture` instance remains unchanged.
- `ConfigureTestServices()` is called after your real app's `ConfigureServices()` method. That means you can replace services that have been previously registered. You can also use this to override configuration values, as you'll see in section 23.6.

`WithWebHostBuilder()` is handy when you want to replace a service for a single test. But what if you wanted to replace the `ICurrencyConverter` in every test. All that boilerplate would quickly become cumbersome. Instead, you can create a custom `WebApplicationFactory`.

#### 23.5.4 Reducing duplication by creating a custom `WebApplicationFactory`

If you find yourself writing `WithWebHostBuilder()` a lot in your integration tests, it might be worth creating a custom `WebApplicationFactory` instead. The following listing shows how to centralize the test service we used in listing 23.15 into a custom `WebApplicationFactory`.

**Listing 23.16 Creating a custom WebApplicationFactory to reduce duplication**

```
public class CustomWebApplicationFactory
    : WebApplicationFactory<Startup>
{
    protected override void ConfigureWebHost(
        IWebHostBuilder builder)
    {
        builder.ConfigureTestServices(services =>
        {
            services.RemoveAll<ICurrencyConverter>();
            services.AddSingleton
                <ICurrencyConverter, StubCurrencyConverter>();
        });
    }
}
```

**Derive from  
WebApplicationFactory.**

**Add custom  
configuration  
for your  
application.**

There are many functions available to override. This is equivalent to calling WithWebHostBuilder.

In this example, we override ConfigureWebHost and configure the test services for the factory.<sup>7</sup> You can use your custom factory in any test by injecting it as an IClassFixture, as you have before. For example, the following listing shows how you would update listing 23.15 to use the custom factory defined in listing 23.16.

**Listing 23.17 Using a custom WebApplicationFactory in an integration test**

```
public class IntegrationTests:
    IClassFixture<CustomWebApplicationFactory>
{
    private readonly CustomWebApplicationFactory _fixture;
    public IntegrationTests(CustomWebApplicationFactory fixture)
    {
        _fixture = fixture;
    }

    [Fact]
    public async Task ConvertReturnsExpectedValue()
    {
        HttpClient client = _fixture.CreateClient(); ←

        var response = await client.GetAsync("/api/currency");

        response.EnsureSuccessStatusCode();
        var content = await response.Content.ReadAsStringAsync();

        Assert.Equal("3", content); ←
    }
}
```

**Implement the IClassFixture  
interface for the custom factory.**

**The client already  
contains the test  
service configuration.**

**The result confirms the  
test service was used.**

**Inject an  
instance of the  
factory in the  
constructor.**

<sup>7</sup> WebApplicationFactory has many other methods you could override for other scenarios. For details, see <http://mng.bz/mgq8>.

You can also combine your custom `WebApplicationFactory`, which substitutes services that you always want to replace, with the `WithWebHostBuilder()` method to override additional services on a per-test basis. That combination gives you the best of both worlds: reduced duplication with the custom factory, and control with the per-test configuration.

Running integration tests using your real app's configuration provides about the closest you'll get to a guarantee that your app is working correctly. The sticking point in that guarantee is nearly always external dependencies, such as third-party APIs and databases.

In the final section of this chapter, we'll look at how to use the SQLite provider for EF Core with an in-memory database. You can use this approach to write tests for services that use an EF Core database context, without needing access to a real database.

## 23.6 Isolating the database with an in-memory EF Core provider

In this section you'll learn how to write unit tests for code that relies on an EF Core `DbContext`. You'll learn how to create an in-memory database, and the difference between the EF in-memory provider and the SQLite in-memory provider. Finally, you'll see how to use the in-memory SQLite provider to create fast, isolated tests for code that relies on a `DbContext`.

As you saw in chapter 12, EF Core is an ORM that is used primarily with relational databases. In this section I'm going to discuss one way to test services that depend on an EF Core `DbContext` without having to configure or interact with a real database.

**NOTE** To learn more about testing your EF Core code, see *Entity Framework Core in Action*, 2nd ed., by Jon P. Smith (Manning, 2021), <http://mng.bz/5j87>.

The following listing shows a highly stripped-down version of the `RecipeService` you created in chapter 12 for the recipe app. It shows a single method to fetch the details of a recipe using an injected EF Core `DbContext`.

### Listing 23.18 RecipeService to test, which uses EF Core to store and load entities

```
public class RecipeService
{
    readonly AppDbContext _context;
    public RecipeService(AppDbContext context)
    {
        _context = context;
    }
    public RecipeViewModel GetRecipe(int id)
    {
        return _context.Recipes
            .Where(x => x.RecipeId == id)
            .Select(x => new RecipeViewModel
            {
                // ...
            });
    }
}
```

An EF Core `DbContext` is injected in the constructor.

Uses the `DbSet<Recipes>` property to load recipes and creates a `RecipeViewModel`

```
{  
    Id = x.RecipeId,  
    Name = x.Name  
})  
.SingleOrDefault();  
}  
}
```

Writing unit tests for this class is a bit of a problem. Unit tests should be fast, repeatable, and isolated from other dependencies, but you have a dependency on your app's `DbContext`. You probably don't want to be writing to a real database in unit tests, as it would make the tests slow, potentially unrepeatable, and highly dependent on the configuration of the database: a fail on all three requirements!

**NOTE** Depending on your development environment, you *may* want to use a real database for your integration tests, despite these drawbacks. Using a database like the one you'll use in production increases the likelihood you'll detect any problems in your tests. You can find an example of using Docker to achieve this in Microsoft's "Testing ASP.NET Core services and web apps" documentation: <http://mng.bz/zxDw>.

Luckily, Microsoft ships two *in-memory* database providers for this scenario. Recall from chapter 12 that when you configure your app's `DbContext` in `Startup.ConfigureServices()`, you configure a specific database provider, such as SQL Server:

```
services.AddDbContext<AppDbContext>(options =>  
    options.UseSqlServer(connectionString);
```

The in-memory database providers are alternative providers designed *only for testing*. Microsoft includes two in-memory providers in ASP.NET Core:

- *Microsoft.EntityFrameworkCore.InMemory*—This provider doesn't simulate a database. Instead, it stores objects directly in memory. It isn't a relational database as such, so it doesn't have all the features of a normal database. You can't execute SQL against it directly, and it won't enforce constraints, but it's fast.
- *Microsoft.EntityFrameworkCore.Sqlite*—SQLite is a relational database. It's very limited in features compared to a database like SQL Server, but it's a true relational database, unlike the in-memory database provider. Normally a SQLite database is written to a file, but the provider includes an in-memory mode, in which the database stays in memory. This makes it much faster and easier to create and use for testing.

Instead of storing data in a database on disk, both of these providers store data in memory, as shown in figure 23.9. This makes them fast and easy to create and tear down, which allows you to create a new database for every test to ensure your tests stay isolated from one another.

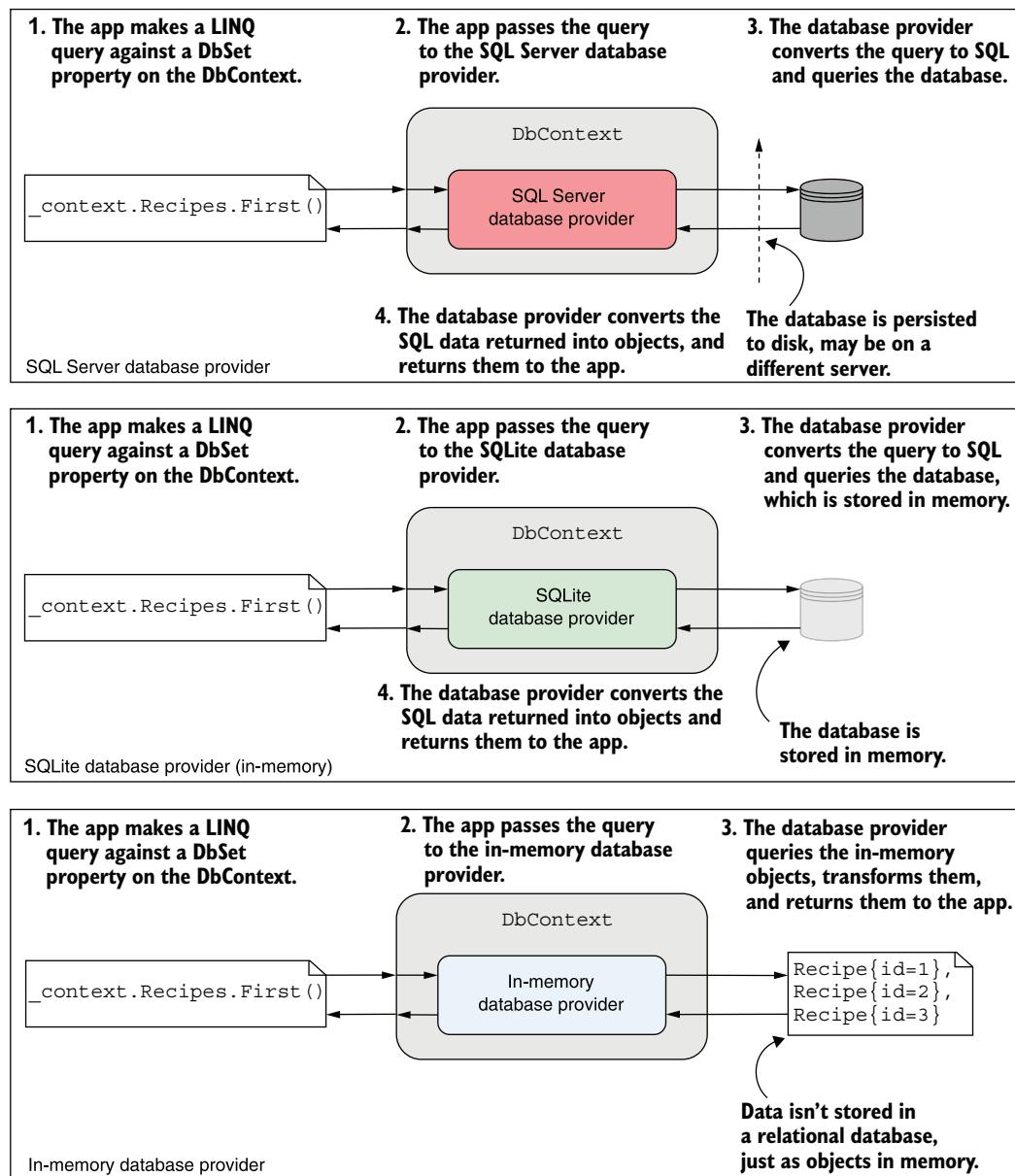


Figure 23.9 The in-memory database provider and SQLite provider (in-memory mode) compared to the SQL Server database provider. The in-memory database provider doesn't simulate a database as such. Instead, it stores objects in memory and executes LINQ queries against them directly.

**NOTE** In this section, I describe how to use the SQLite provider as an in-memory database, as it's more full-featured than the in-memory provider. For details on using the in-memory provider, see Microsoft's "EF Core In-Memory Database Provider" documentation: <http://mng.bz/hdIq>.

To use the SQLite provider in memory, add the `Microsoft.EntityFrameworkCore.Sqlite` package to your test project's `.csproj` file. This adds the `UseSqlite()` extension method, which you'll use to configure the database provider for your unit tests.

Listing 23.19 shows how you could use the in-memory SQLite provider to test the `GetRecipe()` method of `RecipeService`. Start by creating a `SqliteConnection` object and using the "`DataSource=:memory:`" connection string. This tells the provider to store the database in memory and then open the connection.

**WARNING** The in-memory database is destroyed when the connection is closed. If you don't open the connection yourself, EF Core will close the connection to the in-memory database when you dispose of the `DbContext`. If you want to share an in-memory database between `DbContexts`, you must explicitly open the connection yourself.

Next, pass the `SqliteConnection` instance into the `DbContextOptionsBuilder<>` and call `UseSqlite()`. This configures the resulting `DbContextOptions<>` object with the necessary services for the SQLite provider and provides the connection to the in-memory database. By passing this options object into an instance of `AppDbContext`, all calls to the `DbContext` result in calls to the in-memory database provider.

#### Listing 23.19 Using the in-memory database provider to test an EF Core `DbContext`

```

    Opens the connection so EF Core
    won't close it automatically
    [Fact]
    public void GetRecipeDetails_CanLoadFromContext()
    {
        var connection = new SqliteConnection("DataSource=:memory:");
        connection.Open();

        var options = new DbContextOptionsBuilder<AppDbContext>()
            .UseSqlite(connection)
            .Options;

        using (var context = new AppDbContext(options))
        {
            context.Database.EnsureCreated();
            context.Recipes.AddRange(
                new Recipe { RecipeId = 1, Name = "Recipe1" },
                new Recipe { RecipeId = 2, Name = "Recipe2" },
                new Recipe { RecipeId = 3, Name = "Recipe3" });
            context.SaveChanges();
        }
    }

```

The diagram uses callouts to explain various parts of the code:

- Creates a `DbContext` and passes in the options**: Points to the `using (var context = new AppDbContext(options))` block.
- Ensures the in-memory database matches EF Core's model (similar to running migrations)**: Points to the `context.Database.EnsureCreated()` call.
- Creates an instance of `DbContextOptions<>` and configures it to use the SQLite connection**: Points to the `var options = new DbContextOptionsBuilder<AppDbContext>()` and `.UseSqlite(connection)` lines.
- Configures an in-memory SQLite connection using the special "in-memory" connection string**: Points to the `connection.Open()` line.
- Adds some recipes to the `DbContext`**: Points to the `context.Recipes.AddRange(...)` call.
- Saves the changes to the in-memory database**: Points to the `context.SaveChanges()` call.

```

using (var context = new AppDbContext(options))
{
    var service = new RecipeService(context);
    var recipe = service.GetRecipe(id: 2);
    Assert.NotNull(recipe);
    Assert.Equal(2, recipe.Id);
    Assert.Equal("Recipe2", recipe.Name);
}

```

**Verifies that you correctly retrieved the recipe from the in-memory database**

**Creates a fresh DbContext to test that you can retrieve data from the DbContext**

**Creates the Recipe-Service to test and pass in the fresh DbContext**

**Executes the GetRecipe function. This executes the query against the in-memory database.**

This example follows the standard format for any time you need to test a class that depends on an EF Core DbContext:

- 1 Create a `SqliteConnection` with the `"DataSource=:memory;"` connection string and open the connection.
- 2 Create a `DbContextOptionsBuilder<>` and call `UseSqlite()`, passing in the open connection.
- 3 Retrieve the `DbContextOptions` object from the `Options` property.
- 4 Pass the options to an instance of your `DbContext` and ensure the database matches EF Core's model by calling `context.Database.EnsureCreated()`. This is similar to running migrations on your database, but it should *only* be used on test databases. Create and add any required test data to the in-memory database and call `SaveChanges()` to persist the data.
- 5 Create a new instance of your `DbContext` and inject it into your test class. All queries will be executed against the in-memory database.

By using two separate `DbContexts`, you can avoid bugs in your tests due to EF Core caching data without writing it to the database. With this approach, you can be sure that any data read in the second `DbContext` was persisted to the underlying in-memory database provider.

This was a very brief introduction to using the SQLite provider as an in-memory database provider, and EF Core testing in general, but if you follow the setup shown in listing 23.19, it should take you a long way. The source code for this chapter shows how you can combine this code with a custom `WebApplicationFactory` to use an in-memory database for your integration tests. For more details on testing EF Core, including additional options and strategies, see *Entity Framework Core in Action*, 2nd ed., by Jon P. Smith (Manning, 2021).

## Summary

- Unit test apps are console apps that have a dependency on the .NET Test SDK, a test framework such as xUnit, MSTest, or NUnit, and a test runner adapter. You can run the tests in a test project by calling `dotnet test` from the command line in your test project or by using the Test Explorer in Visual Studio.

- Many testing frameworks are compatible with the .NET Test SDK, but xUnit has emerged as an almost *de facto* standard for ASP.NET Core projects. The ASP.NET Core team themselves use it to test the framework.
- To create an xUnit test project, choose xUnit Test Project (.NET Core) in Visual Studio or use the `dotnet new xunit` CLI command. This creates a test project containing the `Microsoft.NET.Test.Sdk`, `xunit`, and `xunit.runner.visualstudio` NuGet packages.
- xUnit includes two different attributes to identify test methods. `[Fact]` methods should be public and parameterless. `[Theory]` methods can contain parameters, so they can be used to run a similar test repeatedly with different parameters. You can provide the data for each `[Theory]` run using the `[InlineData]` attribute.
- Use assertions in your test methods to verify that the system under test (SUT) returned an expected value. Assertions exist for most common scenarios, including verifying that a method call raised an exception of a specific type. If your code raises an unhandled exception, the test will fail.
- Use the `DefaultHttpContext` class to unit test your custom middleware components. If you need access to the response body, you must replace the default `Stream.Null` with a `MemoryStream` instance and manually read the stream after invoking the middleware.
- API controllers and Razor Page models can be unit tested just like other classes, but they should generally contain little business logic, so it may not be worth the effort. For example, the API controller is tested independently of routing, model validation, and filters, so you can't easily test logic that depends on any of these aspects.
- Integration tests allow you to test multiple components of your app at once, typically within the context of the ASP.NET Core framework itself. The `Microsoft.AspNetCore.TestHost` package provides a `TestServer` object that you can use to create a simple web host for testing. This creates an in-memory server that you can make requests to and receive responses from. You can use the `TestServer` directly when you wish to create integration tests for custom components like middleware.
- For more extensive integration tests of a real application, you should use the `WebApplicationFactory` class in the `Microsoft.AspNetCore.Mvc.Testing` package. Implement `IClassFixture<WebApplicationFactory<Startup>>` on your test class and inject an instance of `WebApplicationFactory<Startup>` into the constructor. This creates an in-memory version of your whole app, using the same configuration, DI services, and middleware pipeline. You can send in-memory requests to your app to get the best idea of how your application will behave in production.
- To customize the `WebApplicationFactory`, call `WithWebHostBuilder()` and call `ConfigureTestServices()`. This method is invoked after your app's standard

DI configuration. This enables you to add or remove the default services for your app, such as to replace a class that contacts a third-party API with a stub implementation.

- If you find you need to customize the services for every test, you can create a custom `WebApplicationFactory` by deriving from it and overriding the `ConfigureWebHost` method. You can place all your configuration in the custom factory and implement `IClassFixture<CustomWebApplicationFactory>` in your test classes, instead of calling `WithWebHostBuilder()` in every test method.
- You can use the EF Core SQLite provider as an in-memory database to test code that depends on an EF Core database context. You configure the in-memory provider by creating a `SqliteConnection` with a "DataSource=:memory:" connection string. Create a `DbContextOptionsBuilder<>` object and call `UseSqlite()`, passing in the connection. Finally, pass `DbContextOptions<>` into an instance of your app's `DbContext` and call `context.Database.EnsureCreated()` to prepare the in-memory database for use with EF Core.
- The SQLite in-memory database is maintained as long as there's an open `SqliteConnection`. By opening the connection manually, the database can be used with multiple `DbContexts`. If you don't call `Open()` on the connection, EF Core will close the connection (and delete the in-memory database) when the `DbContext` is disposed of.

# *appendix A*

## *Preparing your development environment*

---

### **This appendix covers**

- Installing the .NET SDK
- Choosing an editor or IDE

For .NET developers in a Windows-centric world, Visual Studio was pretty much a developer requirement in the past. But with .NET and ASP.NET Core going cross-platform, that's no longer the case.

All of ASP.NET Core (creating new projects, building, testing, and publishing) can be run from the command line for any supported operating system. All you need is the .NET SDK, which provides the .NET Command Line Interface (CLI). Alternatively, if you're on Windows and not comfortable with the command line, you can still use File > New Project in Visual Studio to dive straight in. With ASP.NET Core, it's all about choice!

In a similar vein, you can now get a great editing experience outside of Visual Studio thanks to the OmniSharp project.<sup>1</sup> This is an open source set of libraries and editor plugins that provide code suggestions and autocomplete (IntelliSense)

---

<sup>1</sup> Information about the OmniSharp project can be found at [www.omnisharp.net](http://www.omnisharp.net). Source code can be found at <https://github.com/omnisharp>.

across a wide range of editors and operating systems. How you set up your environment will likely depend on which operating system you’re using and what you’re used to.

Remember that for .NET Core and .NET 5.0, the operating system you choose for development has no bearing on the final systems you can run on—whether you choose Windows, macOS, or Linux for development, you can deploy to any supported system.

In this appendix I’ll show you how to install the .NET SDK so you can build, run, and publish .NET apps. I’ll also discuss some of the integrated development environment (IDE) and editor options available for you to build applications.

**NOTE** In this book, I use Visual Studio for most of the examples, but you’ll be able to follow along using any of the tools I discuss here. The book assumes you’ve successfully installed .NET 5.0 and an editor on your computer.

## A.1 *Installing the .NET SDK*

The most important thing you need for .NET Core and .NET 5.0 development is the .NET SDK. In this section I’ll describe how to install the .NET SDK and how to check which version you have installed.

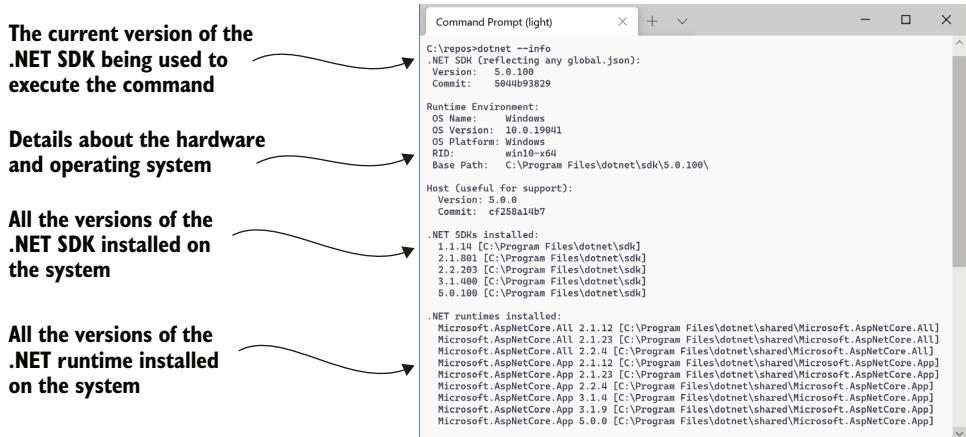
To start programming with .NET, you need to install the .NET SDK (previously called the .NET Core SDK). This contains the base libraries, tooling, and compiler you need to create .NET applications.

You can install the .NET SDK from <https://dotnet.microsoft.com/download>. This contains links to download the latest version of .NET for your operating system. If you’re using Windows or macOS, this page contains installer download links; if you’re using Linux, there are instructions for installing .NET using your distribution’s package manager, as a Snap package, or as a manual download.

**WARNING** Make sure you download the .NET SDK not the .NET *Runtime*. The .NET runtime is used to execute .NET applications, but it can’t be used to build them. The .NET SDK includes a copy of the runtime, so it can run your applications, but it can also build, test, and publish them.

After installing the .NET SDK, you can run commands with the .NET CLI using the `dotnet` command. Run `dotnet --info` to see information about the version of the .NET SDK currently in use, as well as the .NET SDKs and .NET runtimes you have installed, as shown in figure A.1.

As you can see in figure A.1, I have multiple versions of the .NET SDK (previously, the .NET Core SDK) installed. This is perfectly fine, but not necessary. Newer versions of the .NET SDK can build applications that target older versions of .NET Core. For example, the .NET 5.0 SDK can build .NET 5.0, .NET Core 3.x, .NET Core 2.x, and .NET Core 1.x applications. In contrast, the .NET Core 3.1 SDK *can’t* build .NET 5.0 applications.



**Figure A.1** Use `dotnet --info` to check which version of the .NET SDK is currently used and which versions are available. This screenshot shows I am currently using the release version of the .NET 5 SDK, version 5.0.100.

**TIP** Some IDEs, such as Visual Studio, can automatically install .NET 5.0 as part of their installation process. There is no problem installing multiple versions of .NET Core and .NET 5.0 side by side, so you can always install the .NET SDK manually, whether your IDE installs a different version or not.

By default, when you run `dotnet` commands from the command line, you'll be using the latest version of the .NET SDK you have installed. You can control that and use an older version of the SDK by adding a `global.json` file to the folder. For an introduction to this file, how to use it, and understanding .NET's versioning system, see my blog entry, "Exploring the new `rollForward` and `allowPrerelease` settings in `global.json`": <http://mng.bz/KMzP>.

Once you have the .NET SDK installed, it's time to choose an IDE or editor. The choices available will depend on which operating system you're using and will largely be driven by personal preference.

## A.2 Choosing an IDE or editor

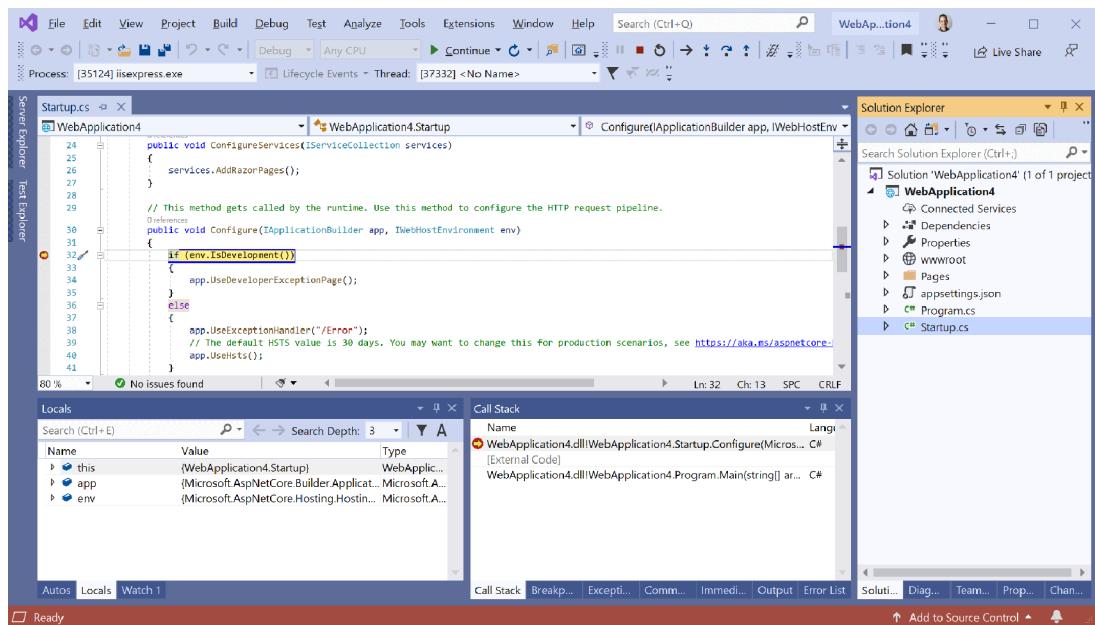
In this section I'll describe a few of the most popular IDEs and editors for .NET development and how to install them. Choosing an IDE is a very personal choice, so this section only describes a few of the options. If your favorite IDE isn't listed here, check the documentation to see if .NET is supported.

### A.2.1 Visual Studio (Windows)

For a long time, Windows has been the best system for building .NET applications, and with the availability of Visual Studio that's arguably still the case.

Visual Studio (figure A.2) is a full-featured IDE that provides one of the best all-around experiences for developing ASP.NET Core applications. Luckily, the Visual

Studio Community edition is now free for open source, students, and small teams of developers.



**Figure A.2** Visual Studio provides one of the most complete ASP.NET Core development environments for Windows users.

Visual Studio comes loaded with a whole host of templates for building new projects, best-in-class debugging, and publishing, without you ever needing to touch a command prompt. It's especially suited if you're publishing to Azure, as it has many direct hooks into Azure features to make development and deployment easier.

You can install Visual Studio by visiting <https://visualstudio.microsoft.com/vs/> and clicking Download Visual Studio. Choose the Community Edition (unless you have a license for the Professional or Enterprise version) and follow the prompts to install Visual Studio.

The Visual Studio *installer* is an application in and of itself, and it will ask you to select *workloads* to install. You can select as many as you like, but for ASP.NET Core development, ensure you select these at a minimum:

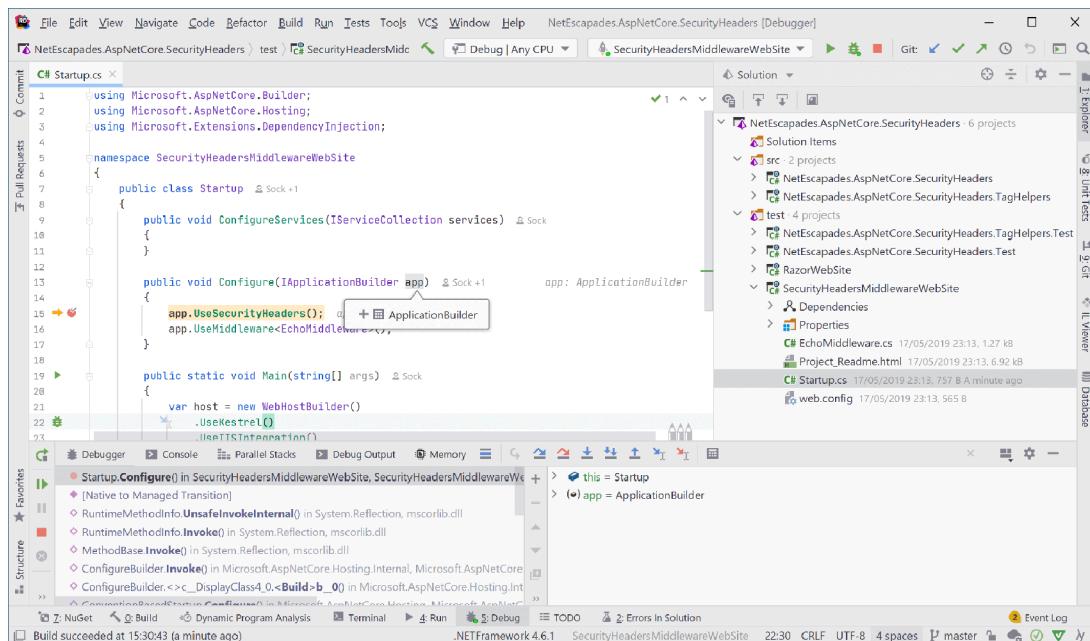
- ASP.NET and web development
- .NET Core cross-platform development

After selecting these workloads, click Download, and fetch a beverage of your choice. Despite having been on a diet recently, Visual Studio still requires many GB to be

downloaded and installed. Once it's finished, you'll be ready to start building ASP.NET Core applications.

## A.2.2 JetBrains Rider (Windows, Linux, macOS)

Rider (figure A.3), from the company JetBrains, is a cross-platform IDE alternative to Visual Studio. Released in 2017, Rider is another full-featured IDE, based on the venerable ReSharper plugin. If you're used to using Visual Studio with the ReSharper plugin, and the multitude of refactorings that this plugin provides, then I strongly suggest investigating Rider. Similarly, if you're familiar with JetBrains' IntelliJ products, you will feel at home in Rider.



**Figure A.3** Rider is a cross-platform .NET IDE from JetBrains. It is based on the ReSharper plugin for Visual Studio, so it includes many of the same refactoring features, as well as a debugger, test runner, and all the other integration features you would expect from a full-featured IDE.

To install Rider visit <https://www.jetbrains.com/rider/> and click Download. Rider comes with a 30-day free trial, after which you will need to purchase a license. If you already have a ReSharper license, you may already have a license for Rider. They also offer discounts or free licenses for various users, such as students and startups, so it's worth looking into.

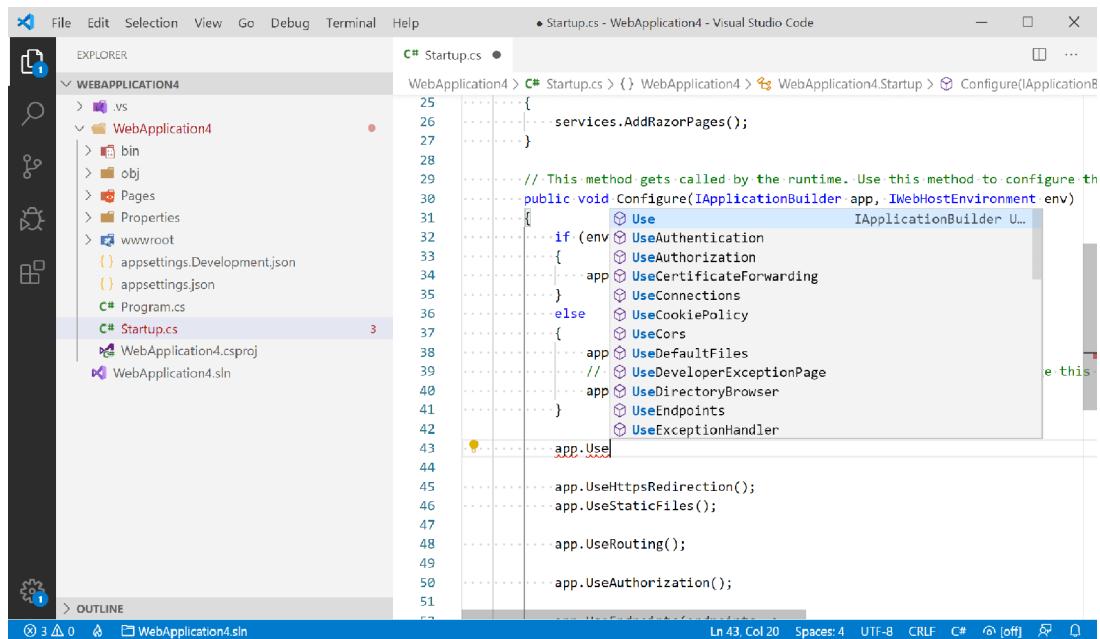
### A.2.3 Visual Studio for Mac (macOS)

Despite the branding, Visual Studio for Mac is a completely different product from Visual Studio. Rebranded and extended from its Xamarin Studio precursor, you can now use Visual Studio for Mac to build ASP.NET Core applications on macOS. Visual Studio for Mac generally has fewer features than Visual Studio or Rider, but it offers a native IDE and is under active development.

To install Visual Studio for Mac, visit <https://visualstudio.microsoft.com/vs/mac/>, click Download Visual Studio for Mac, and download and run the installer.

### A.2.4 Visual Studio Code (Windows, Linux, macOS)

Sometimes you don't want a full-fledged IDE. Maybe you want to quickly view or edit a file, or you don't like the sometimes-unpredictable performance of Visual Studio. In those cases, a simple editor may be all you want or need, and Visual Studio Code is a great choice. Visual Studio Code (figure A.4) is an open source, lightweight editor that provides editing, IntelliSense, and debugging for a wide range of languages, including C# and ASP.NET Core.



**Figure A.4** Visual Studio Code provides cross-platform IntelliSense and debugging.

To install Visual Studio Code, visit <https://code.visualstudio.com/>, click Download, and run the downloaded installer. The first time you open a folder containing a C#

project or solution file with Visual Studio Code, you'll be prompted to install a C# extension. This provides the IntelliSense and integration between Visual Studio Code and the .NET SDK.

The extension model of VS Code is one of its biggest assets, as you can add a huge amount of additional functionality. Whether you're working with Azure, AWS, or any other technology, be sure to check the extension marketplace at <https://marketplace.visualstudio.com/vscode> to see what's available. If you search for ".NET Core", you'll also find a huge array of extensions that can bring VS Code closer to that full-blown IDE experience, if you wish.

# *appendix B*

## *Understanding the .NET ecosystem*

---

### **This appendix covers**

- The history of .NET, leading to the development of .NET Core
- The position of .NET 5.0 in the .NET ecosystem
- Sharing code between projects with .NET Standard
- The future of .NET Standard with .NET 5.0

The .NET ecosystem has changed a lot since .NET was first introduced, but the development of .NET Core and .NET 5.0 has resulted in a particularly large degree of churn and the introduction of many new concepts.

This churn isn't surprising given Microsoft's newfound transparency regarding the development process and building in the open on GitHub. Unfortunately, it can be confusing for developers new to .NET, and even to seasoned veterans! In this appendix, I'll try to straighten out some of the terms that developers new to .NET often find confusing, as well as provide some context for the changes.

In this appendix I'll discuss the history of the .NET ecosystem, how it has evolved, and the issues Microsoft was attempting to solve. As part of this, I'll discuss the similarities and differences between .NET 5.0, .NET Core, and .NET Framework.

.NET Core wasn't developed in isolation, and one of its primary design goals was to improve the ability to share code between multiple frameworks. In section B.2 I'll describe how this was achieved in pre-.NET Core/.NET 5.0 days, using Portable Class Libraries (PCLs) and the successor approach using .NET Standard. Finally, in section B.3, I'll discuss what .NET 5.0 means for .NET Standard.

## B.1 **The evolution of .NET into .NET 5.0**

In this section I'll discuss the history of .NET 5.0 and .NET Core and why they were created. You'll learn about the various .NET platforms that were around in the early 2010s and why their sharding prompted the development of .NET Core as a new cross-platform runtime. Finally, you'll learn how .NET 5.0 has grown out of .NET Core and about the future of .NET.

### B.1.1 **Exploring the .NET platforms that prompted .NET Core**

If you're a .NET developer, chances are you're already familiar with the .NET Framework. The .NET Framework, version 4.8 at the time of writing, is a Windows-only development platform that you can use for both desktop and web development. It's installed by default on Windows and was historically used for most desktop and server .NET development.

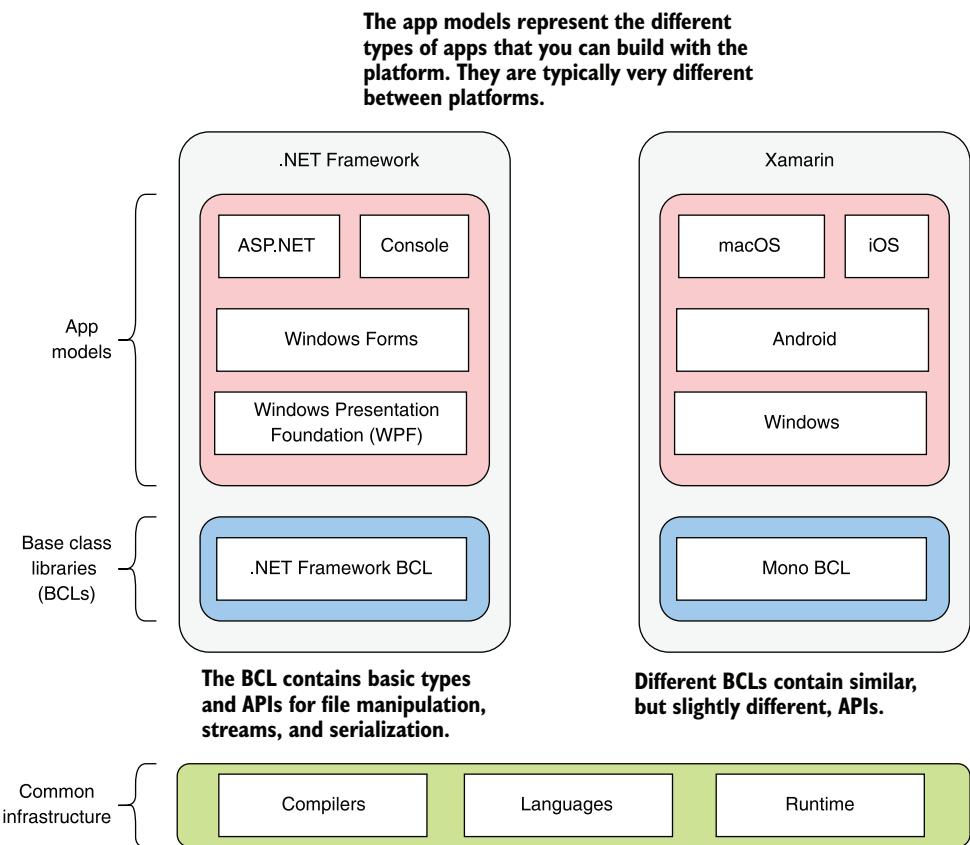
If you're a mobile developer, you might also be familiar with the Xamarin framework, which uses the cross-platform Mono implementation of the .NET Framework. This is an alternative *platform* to the .NET Framework that you can use to build mobile applications on Windows, Android, and iOS.

Historically, these two platforms were completely separate, but they consisted of many similar components and implemented similar APIs. Each platform contained libraries and app models specific to their platform, but they used similar fundamental libraries and types, as shown in figure B.1.

At the bottom of each stack is the tooling that allows you to compile and run .NET applications, such as the compiler and the common language runtime (CLR). At the top of each stack, you have the app-specific libraries that you use to build applications for your platform. For example, you could build a Windows Forms app on the .NET Framework, but not using the Xamarin platform, and vice versa for an iOS app.

In the middle of each stack you have the Base Class Libraries (BCLs). These are the fundamental .NET types you use in your apps on a daily basis: the `int` and `string` types, the file-reading APIs, the Task APIs, and so on. Although both .NET platforms have many similar types, they're fundamentally different, so you can't guarantee a type will exist on both platforms, or that it will expose the same methods and properties.

I've only discussed two platforms so far, the .NET Framework and Xamarin, but .NET has *many* different implementations, of which these are only two. Windows also has the Windows 8/8.1 platform and the Universal Windows Platform (UWP). On phones, in addition to Xamarin, there's the Windows Phone 8.1 and Silverlight Phone platforms. The list goes on and on (Unity, .NET Compact Framework (CF), .NET Micro . . .)!



**Figure B.1** The layers that make up the .NET Framework. Each builds on the capabilities of the layer below, with the highest layer providing the app models that you'll use to build your applications.

Each of these platforms uses a slightly different set of APIs (classes and methods) in their BCLs. Platforms have a certain number of similar APIs between them in their BCLs, but the intersection is patchy. On top of that, the libraries that make up the BCLs of a platform are fundamentally not interoperable. Any source code written for a given set of APIs must be specifically recompiled for each target platform.

Several years ago, Microsoft realized this sharding of .NET was a problem. Developers had to know a *slightly* different set of APIs for each platform, and sharing code so that it could be used on more than one platform was a pain.

On top of that, the primary web development platform of the .NET Framework was showing its age. The software industry was moving toward small, lightweight, cloud-native frameworks that you could deploy in cross-platform environments. The centrally installed Windows-only .NET Framework was not designed for these scenarios. Microsoft set about developing a new framework, called “Project K” during development, which ultimately became .NET Core.

### B.1.2 Introducing .NET Core

The .NET Core platform was Microsoft's solution to the centrally installed, Windows-only .NET Framework. .NET Core is highly modular, can be deployed side-by-side with other .NET Core installations (or alternatively, distributed with the app), and is cross-platform. The term .NET Core is somewhat overloaded, in that it was used through development as a general umbrella term to describe a variety of changes. The .NET Core platform consists of

- *A cross-platform BCL*—The BCL libraries of the .NET Core platform, historically called CoreFX. These contain all the primitive types and libraries for building .NET Core applications.
- *A new cross-platform runtime*—The runtime for .NET Core, called CoreCLR, which executes .NET Core applications.
- *The .NET CLI tooling*—The dotnet tool used for building and publishing apps.
- *The ASP.NET Core and EF Core libraries*—The app-layer libraries, used to build ASP.NET Core applications.

These components make up the .NET Core platform and find their analogs to the various components that make up the .NET Framework and Xamarin platforms you saw in figure B.1. By creating a new platform, Microsoft was able to maintain backward compatibility for apps that used the .NET Framework, while enabling new apps to be developed using .NET Core to take advantage of its cross-platform base and isolated deployment story.

**NOTE** You might be thinking, “Wait, they had too many .NET platforms, so they created *another* one?” If so, you’re on the ball. But luckily, with .NET Core came .NET Standard.

On its own, .NET Core would have meant *yet another* BCL of APIs for .NET developers to learn. But as part of the development of .NET Core, Microsoft introduced .NET Standard. .NET Standard, as the name suggests, ensured a *standard set* of APIs were available on every .NET platform. You no longer had to learn the specific set of APIs available for the flavor of .NET you were using; if you could use the .NET Standard APIs, you knew you’d be fine on multiple platforms. I’ll talk more about .NET Standard in section B.2.

.NET Standard was a good stop-gap solution for writing code that could work on multiple platforms, but it didn’t address one fundamental issue: there were still multiple platforms. Every platform had its *own separate code* that must be maintained by Microsoft, despite being *almost* identical in many places.

Microsoft was innovating quickly in .NET Core, introducing new C# features such as `Async` `Enumerables` and `Span<T>`, as well as providing many performance improvements.<sup>1</sup>

---

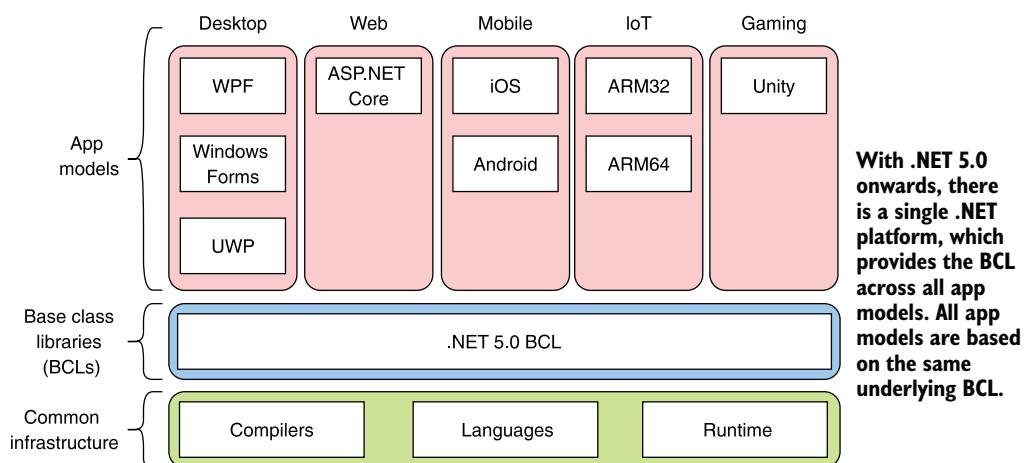
<sup>1</sup> There is a post on the .NET blog detailing the vast low-level improvements made to .NET Core. These are fascinating if you are into that sort of thing! You can find the “Performance Improvements in .NET 5” blog post here: <http://mng.bz/9MEx>.

Unfortunately, none of the other platforms could take advantage of these without significant work. Microsoft's vision for tackling this head-on was One.NET.

### B.1.3 .NET 5.0: The first step in the One .NET vision

In May 2019, Microsoft announced that the next major version of .NET Core after 3.0 would be .NET 5.0.<sup>2</sup> This was the first step in their attempt to unify the .NET platform.

Previously, as I discussed in section B.1.1, you had to use .NET Framework to build Windows desktop apps, Xamarin to build iOS or Android apps, and .NET Core to build cross-platform web apps. Each app model was tied to the underlying platform and used a distinct BCL. The One .NET vision, which started with .NET 5.0, is to have a *single* .NET platform, with a *single* BCL, that can be used with every app model: Windows desktop apps, iOS or Android apps, as well as cross-platform web apps, as shown in figure B.2.



**Figure B.2** .NET 5.0 started the vision to provide a single platform for running multiple app models. Instead of each app model requiring a separate .NET platform, with a separate BCL, all app models will be able to use the same underlying .NET platform and BCL.

Practically speaking, .NET 5.0 really is “just” the next version of .NET Core. There are very few breaking changes in moving an ASP.NET Core 3.1 application to .NET 5.0,<sup>3</sup> and for the most part the upgrade is very easy. .NET 5.0 adds additional features (such as gRPC and Blazor), but fundamentally not much has changed for most ASP.NET Core applications.

<sup>2</sup> You can find the announcement blog post here: <https://devblogs.microsoft.com/dotnet/introducing-net-5/>. This contains a lot of detail of future plans, so I strongly suggest reading it.

<sup>3</sup> You can see the list of breaking changes in Microsoft’s “Breaking changes in .NET 5.0” documentation at <http://mng.bz/j4Dz>.

**NOTE** A common point of confusion is the name: .NET 5.0. The “Core” moniker was dropped to try and signify that “there is only one version of .NET now.” Also, version 4 was skipped, to avoid confusion between the new version and .NET Framework version 4. Hopefully this naming decision will pay off in the long run, even if it’s confusing now.

Microsoft intended to perform the unification of both the BCL and the infrastructure in .NET 5.0, but delays have meant that, so far, only the BCL has been unified; Xamarin apps still use the Mono-based build infrastructure and runtime in .NET 5.0. Microsoft plans to complete the unification efforts, merging the Mono and .NET 5.0 build infrastructure and runtimes in the next release, .NET 6.0.

.NET 5.0 represents the first step on the road to One .NET. The hope is that basing all future development efforts on one platform will reduce duplication of effort and provide both greater stability and progress for the platform. With that in mind, Microsoft has committed to a regular release cadence, so you can easily plan how to keep your apps up to date as new versions of .NET are released.

#### B.1.4 The future: .NET 6.0 and beyond

As with many open source projects, developing in the open is often associated with a faster release cycle than with the traditional .NET Framework. This was certainly the case with .NET Core, with new releases (major and minor) coming regularly for the first few years of development.

While many developers like this faster cadence and the new features it brings, it can lead to some uncertainty. Is it worth spending time upgrading to the latest version now if a new version is going to be released next week?

To counteract the potential churn and give users confidence in continued support, each .NET Core (and .NET 5.0+) release falls into one of two *support tracks*:

- *Long Term Support (LTS)*—These releases are supported for three years from their first release.
- *Current*—These releases are supported until three months after the *next* LTS or current release.

Having two supported tracks leaves you with a simple choice: if you want more features, and are happy to commit to updating your app more frequently, choose Current releases; if you want fewer updates but also fewer features, choose LTS.<sup>4</sup>

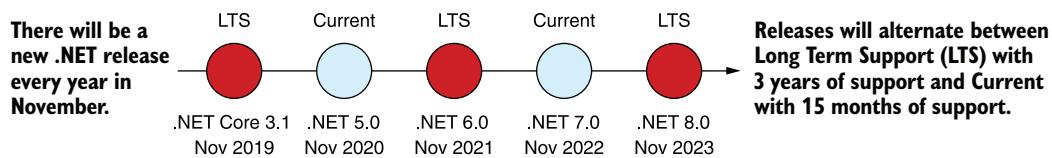
The two-track approach went some way to alleviating uncertainty, but it still left users unsure *exactly* when a new release would occur, and hence how long the current version would be supported.

With .NET 5.0, Microsoft committed to a well-defined release cycle consisting of shipping a new major version of .NET every year, alternating between LTS releases

---

<sup>4</sup> For more details on .NET support policies, see Microsoft’s “.NET Core and .NET 5 Support Policy”: <https://dotnet.microsoft.com/platform/support/policy/dotnet-core>.

and Current releases, as shown in figure B.3. Minor updates are not intended to be common but will occur in interstitial months if required.



**Figure B.3** The timeline for releases of new .NET versions. A new .NET version will be released every year in November. Releases will alternate between Long Term Support (LTS) versions and Current release versions.

With this timeline, you know how long a version of .NET will be supported. If you use a Current track release (such as .NET 5.0) you know you will be supported until three months after the release of .NET 6.0 in November 2021. As an LTS release, .NET 6.0 will be supported until November 2024.

The unification of multiple .NET platforms starting in .NET 5.0 means that there will be less need in future to share code between multiple platforms: that's one of the big selling points of One .NET. Nevertheless, you will no doubt need to share code with existing legacy applications for many years, so code sharing is still a concern.

As I described in section B.1.2, .NET Standard was introduced with .NET Core as a way of sharing code between .NET Core applications and existing legacy applications. Before I dig into the details of .NET Standard, I'll briefly discuss its predecessor, Portable Class Libraries, and why they're now obsolete thanks to .NET Standard.

## B.2 *Sharing code between projects*

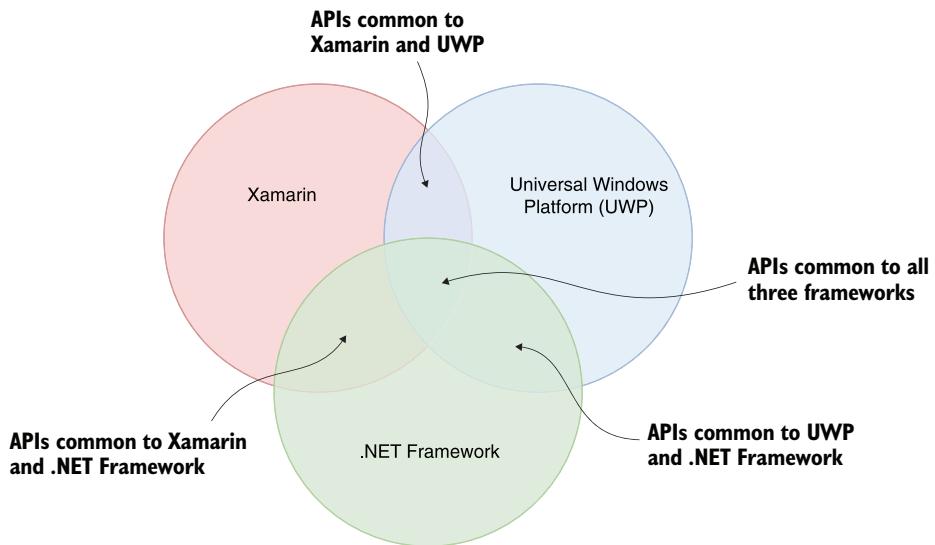
In this section I'll discuss the history of sharing code between .NET platforms using Portable Class Libraries. I'll then introduce .NET Standard as an alternative solution that was introduced with .NET Core.

With so many different .NET implementations, the .NET ecosystem needed a way to share code between libraries, long before .NET Core was envisaged. What if you wanted to use the same classes in both your ASP.NET .NET Framework project and your Silverlight project? You'd have to create a separate project for each platform, copy and paste files between them, and recompile your code for each platform. The result was two different libraries from the same code. Portable Class Libraries (PCLs) were the initial solution to this problem.

### B.2.1 *Finding a common intersection with Portable Class Libraries*

PCLs were introduced to make the process of compiling and sharing code between multiple platforms simpler. When creating a library, developers could specify the platforms they wanted to support, and the project would only have access to the set of APIs common among all of them. Each additional platform supported would reduce

the API surface to only those APIs available in *all* the selected platforms, as shown in figure B.4.



**Figure B.4** Each additional framework that must be supported by a PCL reduces the APIs available to your application. If you support multiple frameworks, you have vastly fewer APIs available to you.

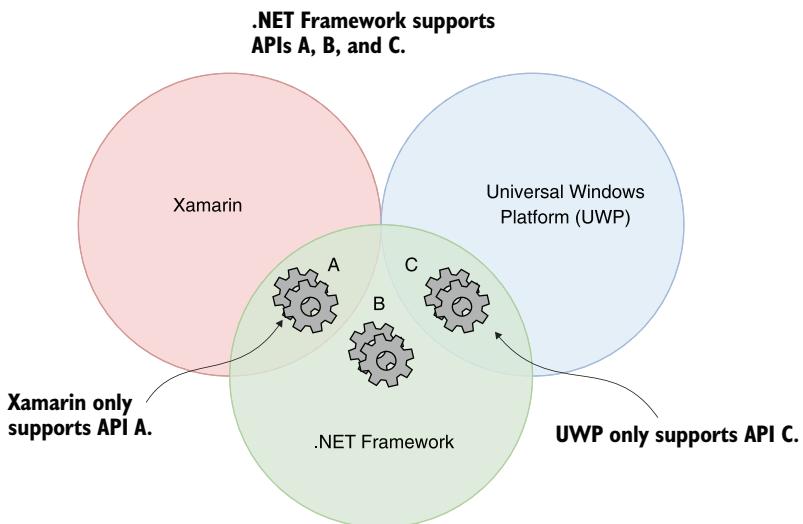
To create a PCL library, you'd create a library that targeted a specific PCL “profile.” This profile contained a precomputed list of APIs known to be available on the associated platforms. That way, you could create one library that you could share across your selected platforms. You could have a single project and a single resulting package—no copy and paste or duplicate projects required.

This approach was a definite improvement over the previous option, but creating PCLs was often tricky. There were inherent tooling complexities to contend with, and understanding the APIs available for each different combination of platforms that made up a PCL profile was difficult.<sup>5</sup>

On top of these issues, every additional platform you targeted would reduce the BCL API surface available for you to use in your library. For example, the .NET Framework might contain APIs A, B, and C. But if Xamarin only has API A and Universal Windows Platform (UWP) only has API C, your library can't use any of them, as shown in figure B.5.

---

<sup>5</sup> See here for the full horrifying list: <https://portablelibraryprofiles.stephencleary.com/>.



**Figure B.5** Each platform exposes slightly different APIs. When creating PCLs, only those APIs that are available in all the targeted platforms are available. In this case, none of the APIs, A, B, or C, is available in all targeted platforms, so none of them can be used in the PCL.

One additional issue with PCL libraries was that they were inherently tied to the underlying platforms they targeted. In order to work with a new target platform, you'd have to recompile the PCL, even if no source code changes were required.

Say you're using a PCL library that supports UWP 10.0 and .NET Framework 4.5. If Microsoft were to release a new platform, let's say .NET Fridge, which exposes the same API as UWP 10.0, you wouldn't be able to use the existing library with your new .NET Fridge application. Instead, you'd have to wait for the library author to recompile their PCL to support the new platform, and who knows when that would be!

PCLs had their day, and they solved a definite problem, but for modern development .NET Standard provides a much cleaner approach.

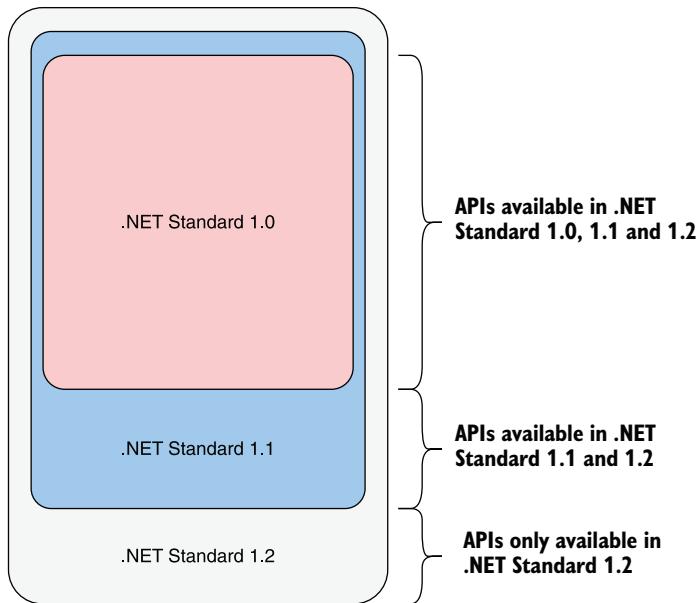
## B.2.2 .NET Standard: A common interface for .NET

As part of the development of .NET Core, Microsoft announced .NET Standard as the successor to PCL libraries. .NET Standard takes the PCL relationship between platform support and APIs available, and flips it on its head:

- **PCLs**—A PCL profile targets a specific set of *platforms*. The APIs available to a PCL library are the common APIs shared by all the platforms in the profile.
- **.NET Standard**—A .NET Standard version defines a specific set of *APIs*. These APIs are always available in a .NET Standard library. Any platform that implements all these APIs supports that version of .NET Standard.

.NET Standard isn't something you download. Instead, it's a list of APIs that a .NET Standard-compatible platform must implement.<sup>6</sup> You can create libraries that target .NET Standard, and you can use that library in any app that targets a .NET Standard-compatible platform.

.NET Standard has multiple *versions*, each of which is a superset of the previous versions. For example, .NET Standard 1.2 includes all the APIs from .NET Standard 1.1, which in turn includes all the APIs from .NET Standard 1.0, as shown in figure B.6.

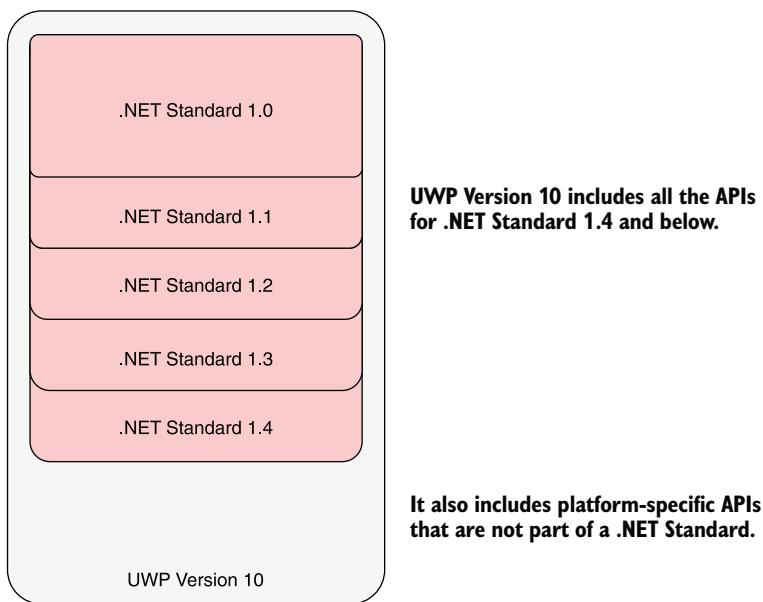


**Figure B.6** Each version of .NET Standard includes all the APIs from previous versions. The smaller the version of .NET Standard, the smaller the number of APIs.

When you create a .NET Standard library, you target a specific version of .NET Standard and can reference any library that targets that version or earlier. If you're writing a library that targets .NET Standard 1.2, you can reference packages that target .NET Standard 1.2, 1.1, or 1.0. Your package can, in turn, be referenced by any library that targets .NET Standard 1.2 or later, or any library that targets a *platform that implements* .NET Standard 1.2 or later.

<sup>6</sup> .NET Standard is, literally, a list of APIs. You can view the APIs included in each version of .NET Standard on GitHub here: <http://mng.bz/yYqe>. For example, you can see the APIs included in .NET Standard 1.0 here: <http://mng.bz/MXY8>.

A platform implements a specific version of .NET Standard if it contains all the APIs required by that version of .NET Standard. By extension, a platform that supports a specific version of .NET Standard also supports all previous versions of .NET Standard. For example, UWP version 10 supports .NET Standard 1.4, which means it also supports .NET Standard versions 1.0–1.3, as shown in figure B.7.



**Figure B.7** The UWP Platform version 10 supports .NET Standard 1.4. That means it contains all the APIs required by the .NET Standard specification version 1.4, which means it also contains all the APIs in earlier versions of .NET Standard. It also contains additional platform-specific APIs that aren't part of any version of .NET Standard.

Each version of a platform supports a different version of .NET Standard. .NET Framework 4.5 supports .NET Standard 1.1, but .NET Framework 4.7.1 supports .NET Standard 2.0. Table B.1 shows some of the versions supported by various .NET platforms. For a more complete list, see Microsoft's “.NET Standard” overview: <https://docs.microsoft.com/dotnet/standard/net-standard>.

A version of this table is often used to explain .NET Standard, but for me the relationship between .NET Standard and a .NET platform all made sense when I saw an example that explained .NET Standard in terms of C# constructs.<sup>7</sup>

<sup>7</sup> The example was originally provided by David Fowler from the ASP.NET team. You can view an updated version of this metaphor here: <http://mng.bz/aoqX>.

**Table B.1 Highest supported .NET Standard version for various .NET platform versions. A blank cell means that version of .NET Standard isn't supported on the platform.**

.NET Standard version	1.0	1.1	1.2	1.3	1.4	1.5	1.6	2.0	2.1
.NET Core and .NET 5.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	2.0	3.0
.NET Framework	4.5	4.5	4.5.1	4.6	4.6.1	4.6.2		4.7.1	
Mono	4.6	4.6	4.6	4.6	4.6	4.6	4.6	5.4	6.4
UWP	10.0	10.0	10.0	10.0	10.0	10.0.16299	10.0.16299	10.0.16299	
Windows	8.0	8.0	8.1						
Windows Phone	8.1	8.1	8.1						

You can think of each version of .NET Standard as a series of inherited interfaces, and the .NET platforms as implementations of one of these interfaces. In the following listing, I use the last two rows of table B.1 to illustrate this, considering .NET Standard 1.0–1.2 and looking at the Windows 8.0 platform and Windows Phone 8.1.

#### Listing B.1 An interpretation of .NET Standard in C#

```
interface NETStandard1_0 { void SomeMethod(); } ← Defines the APIs available in .NET Standard 1.0
interface NETStandard1_1 : NETStandard1_0 { void OtherMethod(); } ← .NET Standard 1.1 inherits all the APIs from .NET Standard 1.0.
interface NETStandard1_2 : NETStandard1_1 { void YetAnotherMethod(); } ← .NET Standard 1.2 inherits all the APIs from .NET Standard 1.1.
class Windows8 : NETStandard1_1 { void SomeMethod () { /* Method implementation */ }
void OtherMethod() { /* Method implementation */ }

void ADifferentMethod() { /* Method implementation */ } } ← APIs available in .NET Standard 1.2 but not in 1.1 or 1.0
Windows 8.0 implements .NET Standard 1.1. ← Implementations of the APIs required by .NET Standard 1.1 and 1.0
Additional APIs that aren't part of .NET Standard, but exist on the Windows 8.0 platform
```

```
→ class WindowsPhone81 : NETStandard1_2
{
    void SomeMethod () { /* Method implementation */ }
    void OtherMethod() { /* Method implementation */ }
    void YetAnotherMethod () { /* Method implementation */ }

    void ExtraMethod1() { /* Method implementation */ }
    void ExtraMethod2() { /* Method implementation */ }
}
```

**Windows Phone 8.1 implements .NET Standard 1.2.**

Implementations of the APIs required by .NET Standard 1.2, 1.1, and 1.0

Additional APIs that aren't part of .NET Standard, but exist on the Windows Phone 8.1 platform

In the same way that you write programs to use interfaces rather than specific implementations, you can target your *libraries* against a .NET Standard interface without worrying about the individual implementation details of the platform. You can then use your library with any platform that implements the required interface version.

One of the advantages you gain by targeting .NET Standard is the ability to target new platforms without having to recompile any of your libraries or wait for dependent library authors to recompile theirs. It also makes reasoning about the exact APIs available far simpler—the higher the .NET Standard version you target, the more APIs will be available to you.

**WARNING** Even if a platform implements a given version of .NET Standard, the method implementation might throw a `PlatformNotSupportedException`. For example, some reflection APIs might not be available on all platforms. .NET 5.0 includes Roslyn Analyzer support to detect this situation and will warn you of the issue at build time. See “Automatically find latent bugs in your code with .NET 5” on the .NET Blog (<http://mng.bz/WdX4>) for more details about analyzers introduced in .NET 5.0.

Unfortunately, things are never as simple as you want them to be. Although .NET Standard 2.0 is a strict superset of .NET Standard 1.6, apps targeting .NET Framework 4.6.1 can reference .NET Standard 2.0 libraries, *even though it technically only implements .NET Standard 1.4, as shown in table B.1*.<sup>8</sup>

**WARNING** Even though .NET Framework 4.6.1 technically only implements .NET Standard 1.4, it can reference .NET Standard 2.0 libraries. This is a special case and applies only to versions 4.6.1–4.7.0. .NET Framework 4.7.1 implements .NET Standard 2.0, so it can reference .NET Standard 2.0 libraries natively.

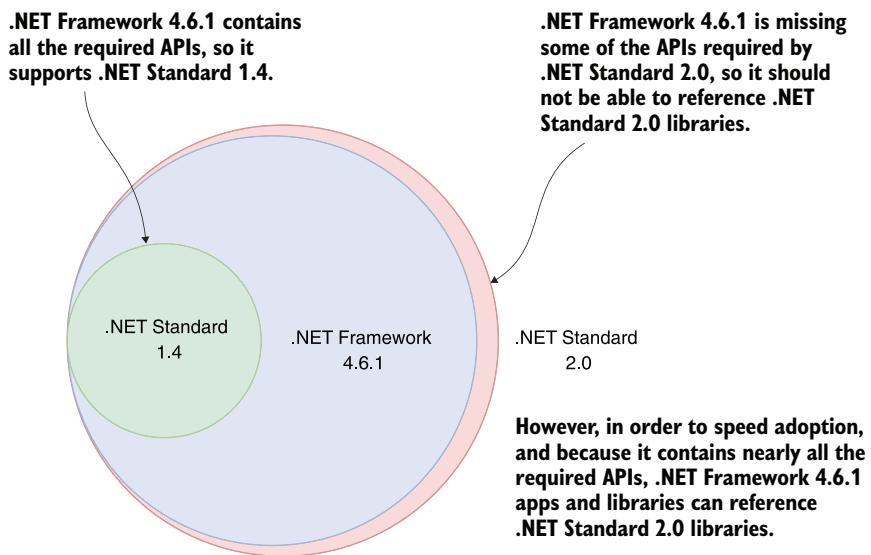
The reasoning behind this move was to counteract a chicken-and-egg problem. One of the early complaints about .NET Core 1.x was how few APIs were available, which made porting projects to .NET Core tricky. Consequently, in .NET Core 2.0, Microsoft

<sup>8</sup> The reasoning behind this move was laid out in the “Introducing .NET Standard” post on the .NET Blog, which I highly recommend reading: <https://devblogs.microsoft.com/dotnet/introducing-net-standard/>.

added thousands of APIs that were available in .NET Framework 4.6.1, the most widely installed .NET Framework version, and added these APIs to .NET Standard 2.0. The intention was for .NET Standard 2.0 to provide the same APIs as .NET Framework 4.6.1.

Unfortunately, .NET Framework 4.6.1 *doesn't* contain the APIs in .NET Standard 1.5 or 1.6. Given that .NET Standard 2.0 is a strict superset of .NET Standard 1.6, .NET Framework 4.6.1 can't support .NET Standard 2.0 directly.

This left Microsoft with a problem. If the most popular version of the .NET Framework didn't support .NET Standard 2.0, no one would write .NET Standard 2.0 libraries, which would hamstring .NET Core 2.0 as well. Consequently, Microsoft took the decision to *allow .NET Framework 4.6.1 to reference .NET Standard 2.0 libraries*, as shown in figure B.8.



**Figure B.8** .NET Framework 4.6.1 doesn't contain the APIs required for .NET Standard 1.5, 1.6, or 2.0. But it contains *nearly* all the APIs required for .NET Standard 2.0. In order to speed up adoption and to make it easier to start using .NET Standard 2.0 libraries, you can reference .NET Standard 2.0 libraries for a .NET Framework 4.6.1 app.

All this leads to some fundamental technical difficulties. .NET Framework 4.6.1 can reference .NET Standard 2.0 libraries, even though it *technically* doesn't support them, but you must have the .NET Core 2.0 SDK installed to ensure everything works correctly.

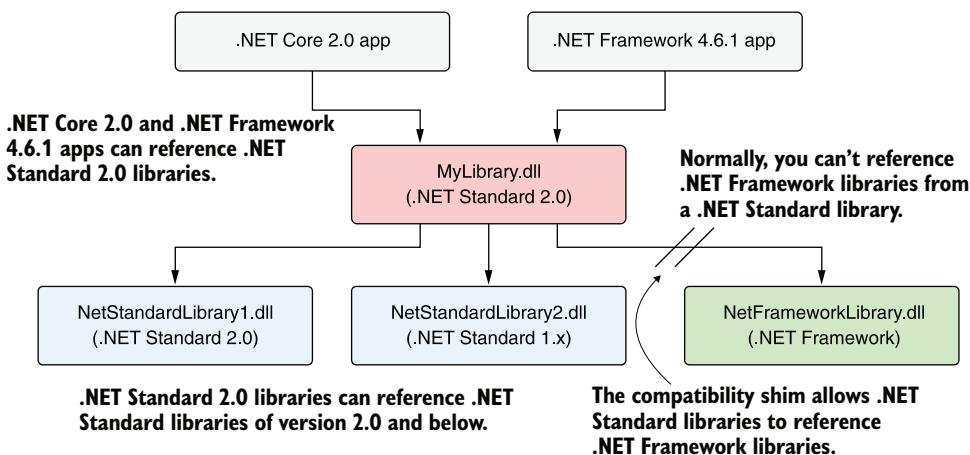
To blur things even further, libraries compiled against .NET Framework 4.6.1 can be referenced by .NET Standard libraries through the use of a compatibility shim, as I'll describe in the next section.

### B.2.3 Fudging .NET Standard 2.0 support with the compatibility shim

Microsoft's plan for .NET Standard 2.0 was to make it easier to build .NET Core apps. If users built libraries targeting .NET Standard 2.0, they could still use them in their .NET Framework 4.6.1 apps, but they could also use the libraries in their .NET Core apps.

The problem was that when .NET Standard 2.0 was first released, no libraries (NuGet packages) would implement it yet. Given that .NET Standard libraries can only reference other .NET Standard libraries, you'd have to wait for all your dependencies to update to .NET Standard, which would have to wait for *their* dependencies first, and so on.

To speed things up, Microsoft created a compatibility shim. This shim allows a .NET Standard 2.0 library to reference *.NET Framework 4.6.1 libraries*. Ordinarily, this sort of reference wouldn't be possible; .NET Standard libraries can only reference .NET Standard libraries of an equal or lower version, as shown in figure B.9.<sup>9</sup>



**Figure B.9** By default, .NET Standard libraries can only reference other .NET Standard libraries, targeting the same .NET Standard version or lower. With the compatibility shim, .NET Standard libraries can also reference libraries compiled against .NET Framework 4.6.1.

By enabling this shim, suddenly .NET Core 2.0 apps could use any of the many .NET Framework 4.6.1 (or lower) NuGet libraries available. As long as the referenced library stuck to APIs that are part of .NET Standard 2.0, you'd be able to reference .NET Framework libraries in your .NET Core 2+ apps or .NET Standard 2.0 libraries, even if your app runs cross-platform on Linux or macOS.

<sup>9</sup> The process by which this magic is achieved is complicated. This “.NET Standard 2” article on GitHub describes the process of assembly unification in detail: <http://mng.bz/8NEZ>.

**WARNING** If the library uses .NET Framework-specific APIs, you'll get an exception at runtime. There's no easy way of knowing whether a library is safe to use, short of examining the source code, so the .NET tooling will raise a warning every time you build. Be sure to thoroughly test your app if you rely on this shim.

If your head is spinning at this point, I don't blame you. This was a particularly confusing point in the evolution of .NET Standard, in which rules were being bent to fit the current environment. This inevitably led to various caveats and hand-waving, followed by bugs and fixes.<sup>10</sup> Luckily, if your development is focused on .NET 5.0, .NET Standard is not something you will generally have to worry about.

### B.3 .NET 5.0 and the future of .NET Standard

In this section I'll discuss what .NET 5.0 means for the future of .NET Standard and the approach you should take for new applications targeting .NET 5.0.

**NOTE** The advice in this section is based on the official guidance from Microsoft regarding the future of .NET Standard here: <http://mng.bz/goqG>.

.NET Standard was necessary when .NET Core was a young framework, to ensure you still had access to the existing NuGet package ecosystem. .NET 5.0 is an evolution of .NET Core, so you can take advantage of that same ecosystem in .NET 5.0.

.NET 5.0 implements .NET Standard 2.1, the latest version of the standard, which is also implemented by .NET Core 3.0. That means .NET 5.0 applications can reference

- Any NuGet package or library that implements .NET Standard 1.0–2.1
- Any NuGet package or library that implements .NET Core 1.x–3.x or .NET 5.0

.NET Standard was designed to handle code-sharing between multiple .NET platforms. But the release of .NET 5.0, and the One .NET vision, specifically aims to have only a *single* platform. Is .NET Standard still useful?

Yes and no. From .NET 5.0 onwards, no more versions of .NET Standard are planned, as subsequent versions of .NET (for example, .NET 7.0) will already be able to reference libraries targeting earlier versions of .NET (such as .NET 5.0 and .NET 6.0).

Unfortunately, the One .NET vision didn't quite come to fruition in .NET 5.0—Xamarin apps still use the Mono runtime and tooling (though they use the same BCL as .NET 5.0). That means if you need to share code between .NET 5.0 applications and Xamarin applications, you will need to use .NET Standard. In .NET 6.0, if all goes to plan, .NET Standard will not be required for that scenario; you'll be able to target .NET 6.0 and share your code across ASP.NET Core applications and Xamarin apps.

---

<sup>10</sup> You can find an example of one such issue here, but there were, unfortunately, many similar cases: <https://github.com/dotnet/runtime/issues/29314>.

.NET Standard will also remain useful when you need to share code between .NET 5.0+ applications and legacy (.NET Core, .NET Framework, Xamarin) applications. .NET Standard remains the mechanism for this cross-.NET platform code-sharing.

## **Summary**

- .NET has many different implementations, including the .NET Framework, Mono, and Unity. Each of these is a separate platform with separate Base Class Libraries (BCLs) and app models. .NET Core is another separate platform.
- Each platform has a BCL that provides fundamental .NET types and classes, such as strings, file manipulation, and streams. Each platform has a slightly different BCL.
- .NET 5.0 is the first step in unifying these platforms, most notably Mono (and hence Xamarin) and .NET Core, under the One .NET vision. App models currently associated with other platforms will be made available on the new .NET 5+ platform.
- .NET 5.0 unifies the code bases for the Mono and .NET Core BCLs. In .NET 6.0, the runtime and tooling will also be unified, completing the One .NET vision. This will allow you to build web, desktop, and mobile apps and more with a single .NET platform, .NET 6.0.
- .NET will see a new major release every year. These will alternate between Long Term Support releases, which receive 3 years of support, and Current releases, which receive 15 months of support.
- Portable Class Libraries (PCLs) attempted to solve the problem of sharing code between .NET platforms by allowing you to write code to the logical intersection of each platform's BCL. Each additional platform you targeted meant fewer BCL APIs in common.
- .NET Standard defines a standard set of APIs that are available across all platforms that support it. You can write libraries that target a specific version of .NET Standard and they'll be compatible with any platform that supports that version of .NET Standard.
- Each version of .NET Standard is a superset of the previous one. For example, .NET Standard 1.2 includes all the APIs from .NET Standard 1.1, which in turn includes all the APIs from .NET Standard 1.0.
- Each version of a platform supports a specific version of .NET Standard. For example, .NET Framework 4.5.1 supports .NET Standard 1.2 (and hence also .NET Standard 1.1 and 1.0).
- .NET Framework 4.6.1 technically only supports .NET Standard 1.4. Thanks to a compatibility shim, you can reference .NET Standard 2.0 libraries from a .NET Framework 4.6.1 app. Similarly, you can reference a .NET Framework library from a .NET Standard 2.0 library, which wouldn't be possible without the shim.

- If you rely on the compatibility shim to reference a .NET Framework 4.6.1 library from a .NET Standard 2.0 library, and the referenced library uses .NET Framework-specific APIs, you'll get an exception at runtime.
- An app must target a .NET platform implementation, such as .NET 5.0 or .NET Core 3.1. It can't target .NET Standard.
- .NET 5.0 supports .NET Standard 2.1. It can reference any .NET Standard library and any .NET Core library.

# *appendix C*

## *Useful references*

---

In this appendix I'll provide a number of links and references that I've found useful for learning about .NET Core/.NET 5.0, .NET Standard, and ASP.NET Core.

### **C.1 Relevant books**

In this book, we touched on several topics and aspects of the .NET ecosystem that are somewhat peripheral to building ASP.NET Core applications. For a deeper understanding of those topics, I recommend the books in this section. They cover areas that you'll inevitably encounter when building ASP.NET Core applications:

- Vladimir Khorikov, *Unit Testing Principles, Patterns, and Practices* (Manning, 2020), <http://mng.bz/E2go>. Learn to refine your unit tests using modern best practices in this excellent book that contains examples in C#.
- Dustin Metzgar, *.NET Core in Action* (Manning, 2018), <http://mng.bz/NY9N>. ASP.NET Core apps are built using .NET Core and .NET 5.0. This book provides everything you need to know about running on the platform.
- Roy Osherove, *The Art of Unit Testing*, second edition (Manning, 2013), <http://mng.bz/DRan>. In *ASP.NET Core in Action*, I discuss the mechanics of unit testing ASP.NET Core applications. For a deeper discussion of how to create your tests, I recommend *The Art of Unit Testing*.
- Chris Sainty, *Blazor in Action* (Manning, 2021), <http://mng.bz/l1P6>. Blazor is an exciting new framework that uses the power of industry standard WebAssembly to run .NET in the browser. With Blazor you can build single-page applications, just as you would with a JavaScript framework like Angular or React, but using the C# language and tooling that you already know.
- Jon P. Smith, *Entity Framework Core in Action*, second edition (Manning, 2021), <http://mng.bz/BRj0>. If you're using EF Core in your apps, I highly

recommend *Entity Framework Core in Action*. It covers all the features and pitfalls of EF Core, as well as how to tune your app for performance.

- Steven Van Deursen and Mark Seemann, *Dependency Injection Principles, Practices, and Patterns* (Manning, 2019), <http://mng.bz/d4lN>. Dependency injection is a core aspect of ASP.NET Core, so *Dependency Injection Principles, Practices, and Patterns* is especially relevant now. It introduces the patterns and antipatterns of dependency injection in the context of .NET and the C# language.

## C.2 Announcement blog posts

When Microsoft releases a new version of ASP.NET Core or .NET Core, they typically write an announcement blog post. These posts provide a high-level overview of the topic, with many examples of new features. They're a great place to start if you want to quickly get acquainted with a topic:

- Richard Lander, “Announcing .NET 5.0,” *.NET Blog* (Microsoft, November 10, 2020), <https://devblogs.microsoft.com/dotnet/announcing-net-5-0/>. Announcement blog post for .NET 5.0, describing some of the highlights introduced in .NET 5.0.
- Richard Lander, “Introducing .NET 5.0,” *.NET Blog* (Microsoft, May 6, 2019), <https://devblogs.microsoft.com/dotnet/introducing-net-5/>. The original announcement blog post for .NET 5.0, describing the vision for the platform.
- Immo Landwerth, “The future of .NET Standard,” *.NET Blog* (Microsoft, September 15, 2020), <https://devblogs.microsoft.com/dotnet/the-future-of-net-standard/>. A discussion of what .NET 5.0 means for the future of .NET Standard, including guidance for library authors.
- Immo Landwerth, “.NET Standard—Demystifying .NET Core and .NET Standard,” *Microsoft Developer Network* (Microsoft, September 2017), <https://msdn.microsoft.com/en-us/magazine/mt842506.aspx>. A long post introducing .NET Core and explaining where .NET Standard fits in the .NET ecosystem.
- Microsoft Docs, “.NET Core and .NET 5 Support Policy,” <https://dotnet.microsoft.com/platform/support/policy/dotnet-core>. Microsoft’s official support policy for .NET Core and .NET 5.0.
- Daniel Roth, “Announcing ASP.NET Core in .NET 5,” *ASP.NET Blog* (Microsoft, November 10, 2020), <https://devblogs.microsoft.com/aspnet/announcing-asp-net-core-in-net-5/>. Announcement blog post for ASP.NET Core 5.0. Describes how to upgrade a project from 3.1 to 5.0 and provides links to some of the features specific to ASP.NET Core 5.0.
- Mads Torgersen, “C# 9.0 on the record,” *.NET Blog* (Microsoft, November 10, 2020), <https://devblogs.microsoft.com/dotnet/c-9-0-on-the-record/>. Announcement blog post for C# 9.0, released alongside .NET 5.0.

### C.3 Microsoft documentation

Historically, Microsoft documentation has been poor, but with ASP.NET Core there has been a massive push to ensure the docs are useful and current. You can find walkthroughs, targeted documentation for specific features, documentation for supported APIs, and even an in-browser C# compiler:

- Microsoft Docs, “.NET API Browser,” <https://docs.microsoft.com/dotnet/api/>. This is an API browser that can be used to work out which .NET APIs are available on which .NET platforms.
- Microsoft Docs, “ASP.NET documentation,” <https://docs.microsoft.com/aspnet/core/>. This is the official documentation for ASP.NET Core.
- Microsoft Docs, “Cross-platform targeting”, <https://docs.microsoft.com/dotnet/standard/library-guidance/cross-platform-targeting>. The official guidance on choosing a target framework for your libraries.
- Rowan Miller, Brice Lambson, Maria Wenzel, Diego Vega, and Martin Milan, “Entity Framework Core” (Microsoft, September 20, 2020), <https://docs.microsoft.com/ef/core/>. This is the official documentation for EF Core.

### C.4 Security-related links

Security is an important aspect of modern web development. This section contains some of the references I refer to regularly, which describe some best practices for web development, as well as practices to avoid:

- Brock Allen and Dominick Baier, “IdentityServer4 1.0.0 documentation,” <https://identityserver4.readthedocs.io/>. Documentation for IdentityServer, the OpenID Connect and OAuth 2.0 framework for ASP.NET Core.
- Dominick Baier, *Dominick Baier on Identity & Access Control* (blog), <https://leastprivilege.com/>. The personal blog of Dominick Baier, co-author of IdentityServer. A great resource when working with authentication and authorization in ASP.NET Core.
- Scott Helme, *Scott Helme* (blog), <https://scotthelme.co.uk/>. Scott Helme’s blog, with advice on security standards, especially security headers you can add to your application.
- Scott Helme, “SecurityHeaders.io—Analyse your HTTP response headers,” <https://securityheaders.com/>. Test your website’s security headers and get advice on why and how you should add them to your app.
- Troy Hunt, *Troy Hunt* (blog), <https://www.troyhunt.com>. Personal blog of Troy Hunt with security-related advice for web developers, particularly .NET developers.
- Microsoft Docs, “Overview of ASP.NET Core Security” (Microsoft, October 24, 2018), <https://docs.microsoft.com/aspnet/core/security/>. The home page of the official ASP.NET Core documentation for all things security related.

## C.5 ASP.NET Core GitHub repositories

ASP.NET Core is entirely open source and developed on GitHub. One of the best ways I've found to learn about the framework is to browse the source code itself. This section contains the main repositories for ASP.NET Core, .NET 5.0, and EF Core:

- .NET Foundation, “ASP.NET Core,” <https://github.com/dotnet/aspnetcore>. The framework libraries that make up ASP.NET Core.
- .NET Foundation, “Entity Framework Core,” <https://github.com/dotnet/efcore>. The EF Core library.
- .NET Foundation, “.NET Runtime,” <https://github.com/dotnet/runtime>. The .NET CoreCLR runtime and BCL libraries, as well as extension libraries.
- .NET Foundation, “.NET SDK and CLI,” <https://github.com/dotnet/sdk>. The .NET command-line interface (CLI), assets for building the .NET SDK, and project templates.

## C.6 Tooling and services

This section contains links to tools and services you can use to build ASP.NET Core projects:

- .NET SDK—<https://dotnet.microsoft.com/download>
- Cloudflare, a global content delivery network you can use to add caching and HTTPS to your applications for free—<https://www.cloudflare.com/>
- JetBrains Rider—<https://www.jetbrains.com/rider/>
- Let’s Encrypt, a free, automated, and open Certificate Authority. You can use it to obtain free SSL certificates to secure your application—<https://letsencrypt.org/>
- Muhammed Rehan Saeed’s .NET Boxed, a comprehensive collection of templates to get started with ASP.NET Core, preconfigured with many best practices—<https://github.com/Dotnet-Boxed/Templates>
- Visual Studio, Visual Studio for Mac, and Visual Studio Code—<https://www.visualstudio.com/>

## C.7 ASP.NET Core blogs

This section contains blogs that focus on ASP.NET Core. Whether you’re trying to get an overview of a general topic, or trying to solve a specific problem, it can be useful to have multiple viewpoints on a topic:

- Khalid Abuhakmeh, *Abuhakmeh*, <https://khalidabuhakmeh.com/>. A wide variety of posts from Khalid, focused on .NET and software development in general.
- Chris Alcock, *The Morning Brew*, <http://blog.cwa.me.uk/>. A collection of .NET-related blog posts, curated daily.
- Damien Boden, *Software Engineering*, <https://damienbod.com/>. An excellent blog by Microsoft MVP Damien Boden on ASP.NET Core with lots of posts about ASP.NET Core with Angular.

- Mike Brind, *Mikesdotnetting*, <https://www.mikesdotnetting.com/>. Mike Brind has many posts on ASP.NET Core, especially focused on ASP.NET Core Razor Pages.
- Steve Gordon, *Steve Gordon—Code with Steve*, <https://www.stevejgordon.co.uk/>. Personal blog of Steve Gordon, focused on .NET. Often focused on writing high-performance code with .NET.
- Scott Hanselman, *Scott Hanselman*, <https://www.hanselman.com/blog/>. Renowned speaker Scott Hanselman's personal blog. A highly diverse blog focused predominantly on .NET.
- Andrew Lock, *.NET Escapades*, <https://andrewlock.net>. My personal blog focused on ASP.NET Core.
- Microsoft .NET Team, *.NET Blog*, <https://blogs.msdn.microsoft.com/dotnet>. The .NET team's blog, with lots of great links.
- David Pine, *IEvangelist*, <http://davidpine.net/>. Personal blog of Microsoft MVP David Pine, with lots of posts on ASP.NET Core.
- Muhammed Rehan Saeed, *Muhammed Rehan Saeed*, <https://rehansaeed.com>. Personal blog of Muhammad Rehan Saeed, Microsoft MVP and author of the .NET Boxed project.
- Rick Strahl, *Rick Strahl's Weblog*, <https://weblog.west-wind.com/>. Excellent blog by Rick Strahl covering a wide variety of ASP.NET Core topics.
- Filip W., *StrathWeb*, <https://www.strathweb.com>. Lots of posts on ASP.NET Core and ASP.NET from Filip, a Microsoft MVP and prolific open source contributor.

## C.8 Video links

If you prefer video for learning a subject, I recommend checking out the links in this section. In particular, the ASP.NET Core community standup provides great insight into the changes you'll see in future ASP.NET Core versions, straight from the team building the framework.

- Microsoft, “.NET Conf 2020,” YouTube video playlist (November 13, 2020), <http://mng.bz/ryRB>. All the sessions from the .NET Conf 2020 online conference announcing .NET 5.0.
- .NET Foundation, “.NET Community Standup,” <https://dotnet.microsoft.com/platform/community/standup>. Weekly videos with the ASP.NET Core team discussing development of the framework. Also includes standups with the .NET team, the Xamarin team, and the EF Core team.
- Immo Landwerth, “.NET Standard—Introduction,” YouTube video (November 28, 2016), <http://mng.bz/Vd0P>. The first video in an excellent series on .NET standard.
- Microsoft, “Channel 9: Videos for developers from the people building Microsoft products and services,” <https://channel9.msdn.com/>. Microsoft’s official

video channel. Contains a huge number of videos on .NET and ASP.NET Core, among many others.

- Shawn Wildermuth, “Building a Web App with ASP.NET Core, MVC, Entity Framework Core, Bootstrap, and Angular,” Pluralsight course, 9:52 hours (October 7, 2019), <http://mng.bz/xmRW>. Shawn Wildermuth’s course on building an ASP.NET Core application.
- Steve Gordon, “Integration Testing ASP.NET Core Applications: Best Practices,” Pluralsight course, 3:25 hours (July 15, 2020), <http://mng.bz/A09z>. One of several courses from Steve Gordon providing guidance and advice on building ASP.NET Core applications.



# *index*

---

## Numerics

---

2FA (two-factor authentication) 445

## A

---

<a> element 248–249  
AbstractValidator<> class 660  
Accept header 281, 284, 425  
Action() method 145–147  
[action] token 274  
action attribute 231, 233–234  
ActionFilterAttribute 420–421  
action filters, customizing model binding and action results with 419–423  
action methods 97, 108  
  custom exception handling for 423–425  
  injecting services directly into 315–317  
  linking to URLs 270–276  
    combining route attributes 272–274  
    handling HTTP verbs 274–276  
    using token replacement to reduce duplication 274  
  short-circuiting 416–418  
ActionResults  
  generating URLs from route parameters with 146–147  
  returning responses with 117–120  
    creating ActionResult classes 119–120  
    NotFoundResult 119  
    PageResult 118–119  
    RedirectToPageResult 118–119  
    StatusCodeResult 119  
action results  
  customizing before they execute 425–427  
  customizing with action filters 419–423

Add() method 386, 716  
Add\*File() method 346  
AddAppConfiguration() method 339–340, 346  
AddAuthorization() method 482, 495, 637  
AddConfiguration 561  
AddConsole() method 554  
AddControllers() method 262, 279, 283, 303, 310, 410, 637  
AddControllersWithViews() method 303  
AddCors() 596, 600  
AddDbContext<T> method 377  
AddDefaultIdentity() method 452, 456  
AddEnvironmentVariables() method 343  
AddFile() method 553–554  
AddHostedService() method 693  
AddHttpMessageHandler<T>() 684  
AddJob<T>() method 707  
AddJsonFile() method 339  
AddMvc() method 303  
AddMvcOptions() method 410  
AddPolicy() method 482, 599  
AddRazorPages() method 45, 107, 152, 262, 327  
AddRazorPagesOptions() method 152  
AddScoped() method 306, 322  
AddSeq() method 566  
AddSingleton<T>() method 308  
@addTagHelper directive 214, 645  
AddTransient() method 321  
AddUserSecrets() method 345  
AJAX (Asynchronous JavaScript and XML) 573  
AllowAnonymous() method 628  
[AllowAnonymous] attribute 477–478  
ambient values 146  
Anchor Tag Helper, generating links with 248–250  
[ApiController] attribute 256, 263, 265, 276–280, 419, 424

- APIs
- authentication for 442–446
  - calling HTTP APIs 668–674
  - unit testing API controllers 731–734
  - Web APIs 255–287
- DbContext class 376–378, 381, 385–386, 451, 453, 459, 464, 695, 706, 747
- Append Version Tag Helper, cache-busting with 250–251
- application model 100–101, 159–160
- application pool 512
- ApplicationUser entity 466, 496
- ApplicationUser type 460–461
- <appsettings> element 331
- {area} route parameter 451
- areas 450
- asp-action attribute 234
- asp-append-version attribute 250
- asp-area attribute 249, 451
- asp- attribute 252
- asp-controller attribute 234
- asp-for attribute 235, 237, 240, 242–243
- asp-format attribute 239
- asp-host attribute 249
- asp-items attribute 242–243
- ASP.NET Core 3–24
- authentication in 439–442
  - authenticating users for subsequent requests 441–442
  - signing in to apps 440–441
  - authorization in 474–481
    - handling unauthorized requests 478–481
    - preventing anonymous users from accessing apps 476–478
  - blogs 779–780
  - dependency injection in 300–303
  - GitHub repositories 779
  - hosting model 504–511
    - choosing deployment method 510–511
    - running vs. publishing 506–510
  - HTTPS development certificates 576–578
  - inner workings of 19–23
    - HTTP web requests 19–21
    - request processing 21–23
  - logging abstractions 543–544
  - overview 4–5, 7–9
  - routing in 126–134
  - testing apps in 716–717
  - web frameworks 5
  - when to choose 9–19
    - converting existing apps to ASP.NET Core 18–19
  - .NET Framework developers creating new apps 14–18
- new to .NET development 12–14
- type of apps that can be built 9–12
- ASP.NET Core app configuration 330–363
- configuration model 331–333
- configuration objects 335–347
- adding configuration provider in Program.cs 338–340
  - reloading configuration values 346–347
  - storing configuration secrets safely 342–346
  - using multiple providers to override configuration values 340–342
- CreateDefaultBuilder 333–335
- for multiple environments 354–361
- identifying hosting environment 355–356
  - loading environment-specific configuration files 356–358
  - setting hosting environment 358–361
- strongly typed settings 347–354
- binding without IOptions interface 353–354
  - designing options classes for automatic binding 351–353
  - IOptions interface 349–350
  - reloading strongly typed options 350–351
- ASPNETCORE\_ENVIRONMENT variable 358, 522
- ASPNETCORE\_HTTPS\_PORT variable 583
- ASPNETCORE\_URLS variable 522
- ASP.NET Core web apps 25–57
- building web host 39–42
  - configuring 42–50
    - adding and configuring services 44–45
    - defining how requests are handled with middleware 45–50
  - creating 29–33
    - building apps 32–33
    - using templates 29–32
  - defining dependencies 37–39
  - generating responses 50–55
    - generating HTML 51–53
    - handling request logic 53–55
  - overview 26–29
  - project layout 35–37
    - running 34–35
- asp-page attribute 233, 248–249
- asp-page-handler attribute 234, 248–249
- asp-protocol attribute 249
- asp-route-\* attribute 234, 248–249
- asp-route attribute 234
- asp-route-id value 234
- asp-validation-for attribute 245
- asp-validation-summary attribute 246
- Assert class 727
- Assert.Throws() method 728
- async() method 719

Asynchronous JavaScript and XML (AJAX) 573  
attribute routing 130, 270–276  
    combining route attributes 272–274  
    convention-based routing vs. 130–132  
    handling HTTP verbs with 274–276  
    using token replacement to reduce duplication in 274  
authentication 436–469  
    for APIs and distributed apps 442–446  
    Identity  
        adding custom data to users 466–468  
        adding to existing projects 458–462  
        creating projects that use 448–458  
        customizing page in default UI 463–465  
        defined 446–448  
    in ASP.NET Core 439–442  
    authenticating users for subsequent requests 441–442  
    signing in to ASP.NET Core apps 440–441  
users and claims 438–439  
AuthenticationMiddleware 441, 452, 459–460, 480  
authorization 470–502  
    applying to endpoints 627–629  
    creating custom policies 484–491  
        custom requirements and handlers 486–491  
        requirements and handlers 484–486  
    hiding elements in Razor templates from unauthorized users 498–501  
    in ASP.NET Core 474–481  
    handling unauthorized requests 478–481  
    preventing anonymous users from accessing app 476–478  
    overview 471–474  
    policies for claims-based 481–484  
    resource-based 492–498  
        creating resource-based  
            AuthorizationHandler 495–498  
        manually authorizing requests 493–495  
    authorization filters, protecting APIs with 415–416  
AuthorizationHandlerContext 488  
AuthorizationHandlers, creating resource-based 495–498  
AuthorizationHandler<T> class 496  
AuthorizationMiddleware 50, 130, 474, 477–478, 621–622  
AuthorizationOptions object 482  
AuthorizationPolicyBuilder 482–483, 487  
AuthorizationResult object 494  
[Authorize] attribute 416, 474, 476–478, 481–482, 485, 487–488, 491–493, 498, 588, 621–622, 627–628  
AuthorizeAsync 494, 496, 499–500  
AuthorizeFilter 416

**B**

BackgroundService 692, 703, 707, 709  
background tasks and services 689–713  
    coordinating using Quartz.NET 703–711  
    configuring job to run on schedule 706–708  
    installing Quartz.NET 704–705  
    using clustering to add redundancy to background tasks 708–711  
creating headless worker services 696–703  
    creating from templates 698–700  
    running in production 700–703  
    running 690–696  
    on timer 691–694  
    using scoped services 694–696  
BaseController class 274  
BCLs (Base Class Libraries) 759  
binding model 156–187  
    binding complex types 168–172  
    binding collections and dictionaries 169–170  
    binding file uploads 170–172  
    simplifying method parameters by binding to complex objects 168–169  
    binding simple types 164–167  
    choosing binding source 172–174  
    generating 99–100  
    handling user input with model validation 174–184  
        need for validation 174–176  
        using DataAnnotations attributes 176–178  
        validating on client for user experience 182–184  
        validating on server for safety 178–182  
    models in Razor Pages and MVC 157–160  
    organizing binding models in Razor Pages 184–186  
binding sources 162, 165, 172  
[BindNever] attribute 173  
[BindProperty] attribute 116, 159, 161, 163, 168, 171, 185, 195, 236, 264, 428  
[BindRequired] attribute 173  
blogs 779–780  
builder.ClearProviders() method 554  
BundlerMinifier 525–534  
    adding to apps 529–532  
    serving common files from CDNs 533–534  
    speeding up apps using bundling and minification 527–529  
    using minified files in production with Environment Tag Helper 532–533

**C**

C#, using in Razor templates 200–201  
CA (certificate authority) 577, 579

captured dependencies 324–327  
 CD (continuous delivery/deployment) 510  
 CDNs (content delivery networks), serving common files from 533–534  
 certificate authority (CA) 577, 579  
 CF (Compact Framework) 759  
 CI (continuous integration) 510  
 claims-based authorization  
     creating custom policies 484–491  
     policies for 481–484  
 ClaimsPrincipal 438, 441, 455, 466–467, 475, 480, 488, 496  
 class attribute 226, 232  
 [ClassData] attribute 726  
 CLI (Command Line Interface) 33, 751  
 CLR (common language runtime) 759  
 clustering, using to add redundancy to background tasks 708–711  
 code-behind (.cshtml.cs file) 194–195  
 collections, binding 169–170  
 Compact Framework (CF) 759  
 compatibility shim 772–773  
 conditionals, adding to Razor templates 201–204  
 ConfigurationBuilder 335, 339–340, 346, 356–357  
 ConfigurationManager 332  
 configuration objects 335–347  
     adding configuration provider in Program.cs 338–340  
     reloading configuration values when they change 346–347  
     storing configuration secrets safely 342–346  
         in environment variables in production 343–344  
         with User Secrets Manager in development 344–346  
     using multiple providers to override configuration values 340–342  
 configuration provider 332  
 configuration secrets, storing safely 342–346  
     in environment variables in production 343–344  
     with User Secrets Manager in development 344–346  
 Configure() method 43, 45–46, 65, 67, 72, 75, 80, 105, 127, 132, 340, 353, 452, 460, 597, 618, 737  
 ConfigureApiBehaviorOptions() method 279  
 ConfigureAppConfiguration() method 335, 337–339, 345, 356  
 ConfigureContainer() method 636–637  
 ConfigureHostingConfiguration() method 334  
 ConfigureLogging() method 334, 554, 561, 566  
 ConfigureServices() method 43, 45, 67, 104, 107, 262, 302, 306, 310, 339–340, 377, 482, 487, 658, 663, 676, 693, 698, 742  
 ConfigureWebHost() method 737  
 ConfigureWebHostDefaults() method 41  
 conneg (content negotiation) 281  
 ConsumesAttribute 417  
 Content() method 653  
 content delivery networks (CDNs), serving common files from 533–534  
 content negotiation 284–285  
 ContentResult 117  
 ContentRootPath property 47  
 Content.SetHtmlContent() 645  
 Content-Type header 281, 612  
 context.ActionArgument 422  
 context.Database.Migrate() 381  
 context.Fail() method 490  
 context.Result property 418, 429  
 context.Succeed() 489  
 continuous delivery/deployment (CD) 510  
 continuous integration (CI) 510  
 [controller] token 274  
 ControllerBase class 263–264, 268, 422, 602, 652, 734  
 Controller base class 145, 217, 263  
 Controller component 95, 157  
 convention-based routing 130–132  
 CORS (cross-origin resource sharing) 573, 593–601  
     adding global policy to whole app 596–598  
     adding to specific Web API actions with EnableCorsAttribute 598–599  
     configuring policies 599–601  
         overview 594–596  
 CorsMiddleware 597  
 CorsPolicyBuilder 600  
 coupling 298  
 CreateApplicationBuilder() 623  
 Create command 391  
 CreateDefaultBuilder() method 40–41, 333–335, 553–554, 561, 566  
 CreateHostBuilder 338, 557  
 CreateInstance() 433  
 CreateLogger() 543, 550  
 Critical log level 548  
 cross-origin resource sharing. *See* CORS  
 cross-site scripting (XSS) attacks 584–588  
 CRUD (create, read, update, and delete) 365  
 .cshtml.cs file (code-behind) 194–195  
 .csproj project file 37–39  
 CSRF (cross-site request forgery) attacks 234, 588–593

- custom components 609–639  
complex configuration requirements 629–635  
partially building configuration to configure additional providers 630–632  
using services to configure `IOptions` 632–635  
endpoints 621–629  
    applying authorization 627–629  
    creating component 622–625  
    creating simple endpoints 625–627  
middleware pipeline 610–621  
    adding to pipeline 616–618  
    branching middleware pipelines 612–615  
    building custom component 619–621  
    creating simple endpoints 611–612  
third-party DI container 635–638
- 
- D**
- `data-*` attributes 231  
`DataAnnotations` attributes 228–229, 237, 376, 395, 659  
    `FluentValidation` vs. 660–663  
    model validation 176–178  
database context 371  
database providers 369, 380  
databases  
    building data model 375–377  
    choosing provider 373–375  
    mapping to app code 370–372  
    migrations 378–384  
        adding 382–384  
        creating 379–382  
    querying data from and saving data to 384–394  
        creating records 385–387  
        loading list of records 387–389  
        loading single record 389–390  
        updating model with changes 390–394  
    registering data context 377–378  
`DataContext` 319–324  
`data-val-*` attribute 238, 240, 245  
`DbContext` 371–372, 376, 451, 453, 695, 706  
`DbContextOptionsBuilder` 377  
`DbContextOptions` object 748  
`DbContext.Remove(entity)` command 393  
Debug configuration 508  
Debug log level 548  
decimal type 167  
`DelegatingHandler` class 684  
`delete()` method 394  
`DELETE` request 150, 276, 591  
`DELETE` verb 150, 275  
deploying apps 503–536  
    ASP.NET Core hosting model 504–511  
    choosing deployment method 510–511  
    running vs. publishing 506–510  
configuring URLs 522–524  
hosting on Linux 517–522  
    preparing apps for deployment 520–522  
    running apps behind reverse proxy on Linux 517–519  
optimizing client-side assets 525–534  
    adding `BundlerMinifier` to apps 529–532  
    serving common files from CDNs 533–534  
speeding up apps using bundling and minification 527–529  
using minified files in production with Environment Tag Helper 532–533  
publishing to IIS 511–517  
    configuring IIS for ASP.NET Core 512–514  
    preparing and publishing 514–517  
design patterns 44  
`DeveloperExceptionPage` 78–80  
`DeveloperExceptionPageMiddleware` 47, 59, 78, 84  
development environment 751–757  
IDEs and editors 753–757  
    JetBrains Rider 755  
    Visual Studio 753–755  
    Visual Studio Code 756–757  
    Visual Studio for Mac 756  
installing .NET SDK 752–753  
DI (dependency injection) 292–329  
    benefits of 293–298  
    creating loosely coupled code 298–300  
    dependency injection containers 44, 296, 302–318  
        adding ASP.NET Core framework services to 302–303  
    injecting services into action methods, page handlers, and views 315–318  
    registering service multiple times 311–315  
    registering services using objects and lambdas 307–311  
    registering services with 304–307  
third-party 635–638  
in ASP.NET Core 300–302  
lifetimes 318–327  
    captured dependencies 324–327  
    scoped 322–323  
    singleton 323–324  
    transient 321–322  
overview 292–302  
    using with filter attributes 431–433  
dictionaries, binding 169–170  
`Dictionary<string, object>` 422  
directives 194  
    `[DisableCors]` attribute 599  
`DisableFormValueModelBindingAttribute` 417  
    `[Display]` attribute 235  
distributed apps, authentication for 442–446

domain model 100  
dotnet add package command 636  
dotnet build command 33, 507  
dotnet command 508, 576, 752  
dotnet <dll> syntax 509  
dotnet ef command 454  
dotnet ef database update command 382  
dotnet ef migrations add command 383  
dotnet new command 507  
dotnet new web command 104, 134  
dotnet new worker command 698  
dotnet publish command 507, 517  
dotnet restore command 32–33, 374, 449, 459,  
  507  
dotnet run command 33–34, 107, 504, 507, 700,  
  705, 740  
dotnet test command 716  
dotnet tool 761  
dynamic web pages 5, 199–206  
  adding loops and conditionals to Razor  
    templates 201–204  
  rendering HTML with Raw 204–206  
  using C# in Razor templates 200–201

**E**

EF Core (Entity Framework Core) 364–397  
  adding to apps 372–378  
    building data model 375–377  
    choosing database provider and  
      installing 373–375  
    registering data context 377–378  
  avoiding SQL injection attacks 603–604  
  defined 366–367  
  mapping database to app code 370–372  
  migrations 378–384  
    adding 382–384  
    creating 379–382  
  object-relational mappers 367–368  
  overview 366, 372  
  querying data from and saving data to  
    database 384–394  
    creating records 385–387  
    loading list of records 387–389  
    loading single record 389–390  
    updating model with changes 390–394  
  updating data model to support Identity  
  460–461  
  using in production apps 394–396  
  when to use 369–370  
[EmailAddress] attribute 237, 642, 655  
[EnableCors] attribute 596, 598  
EnableCorsAttribute 598–599  
EndpointMiddleware 60, 91, 104, 128, 133,  
  149–150, 262, 268, 405, 478, 482, 622

endpoint routing 126–130, 621–629  
  applying authorization 627–629  
  creating component 622–625  
  creating custom endpoints 625–627  
EndpointRoutingMiddleware 127  
entities 371  
Entity Framework Core. *See* EF Core  
EnvironmentName property 47, 355  
Environment Tag Helper  
  using conditional markup with 251–252  
  using minified files in production with  
  532–533  
EnvironmentTagHelper 252  
environment variables 343–344  
error handling 77–89  
  handling exceptions in production 80–84  
  status codes 84–88  
  viewing exceptions in development 78–80  
  Web APIs and 89  
Error log level 547–548, 560  
ETW (Event Tracing for Windows) 552  
event category 546  
EventId 547  
EventLevel 565  
exception filters 402, 423–425  
ExceptionHandlerMiddleware 48, 59, 80–84, 86,  
  357  
exceptions 547  
extension methods 67

**F**

[Fact] attribute 719  
Fact unit tests 724–726  
Fail() method 490  
feature toggle 417  
FileResult 117  
filter pipelines 398–435  
  adding filters 408–410  
  changing log verbosity with filtering 559–564  
  creating custom filters 413–429  
    action filters 419–423  
    authorization filters 415–416  
    exception filters 423–425  
    page filters 427–429  
    resource filters 416–418  
    result filters 425–427  
  creating simple filters 406–408  
dependency injection with filter attributes  
  431–433  
middleware vs. filters 404–406  
MVC filter pipeline 401–402  
order of filter execution 411–413  
  default scope execution order 411–412  
  overriding default order 412–413

**filter pipelines (*continued*)**

- Razor Pages filter pipeline 403–404
  - short-circuiting 429–431
- float type 167
- FluentValidation 659–665
  - adding to apps 663–665
  - DataAnnotations attributes vs. 660–663
- for attribute 235–236, 238
- foreach loop 192, 202, 311
- foreign key 375
- for loop 195, 201
- <form> element 230, 233, 590
- FormatFilterAttribute 425
- forms, creating using Tag Helpers 228–248
  - Form Tag Helper 233–235
  - Input Tag Helper 236–240
  - Label Tag Helper 235–236
  - Select Tag Helper 240–245
  - Textarea Tag Helper 236–240
  - Validation Message Tag Helper 245–248
  - Validation Summary Tag Helper 245–248
- Form Tag Helper 233–235
- FormTagHelper 233
- form values 162
- ForwardedHeadersMiddleware 520–521, 584
  - [FromBody] attribute 173, 277–278
  - [FromForm] attribute 173
  - [FromHeader] attribute 173
  - [FromQuery] attribute 173
  - [FromRoute] attribute 173
  - [FromServices] attribute 173
    - injecting services directly into action methods 315–317
    - injecting services directly into page handlers 315, 317
- FromSqlRaw() method 603

**G**

- GetCurrentDirectory 334
- GetLatestRates() method 679
- GetPathByAction() method 148
- GetPathByPage() method 148
- GET request 54, 115–116, 150, 161, 263, 589, 591, 598
- GetSection(section) method 340
- GetService() method 309
- GetService<T>() method 658
- GetTypedHeaders() method 426
- GetUriByPage() method 148
- GET verb 54, 189, 275
- GitHub repositories 779
- global query filters 394
- globbing patterns 531

**H**

- 
- {handler} route parameter 149
  - {handler} route value 404
  - HandleRequirementAsync() method 488, 496
  - handlers 484
    - handler value 149–150
  - headers 21
  - HeadersMiddleware class 620
  - HEAD verb 151
  - helper methods, creating ActionResult classes
    - using 119–120
  - hooks 400
  - HostBuilder 325, 333–334, 337–339, 345, 514, 520, 553–554, 561, 736–737
  - Host.CreateDefaultBuilder() method 333, 520
  - hosting apps
    - ASP.NET Core hosting model 504–511
      - choosing deployment method 510–511
      - running vs. publishing 506–510
    - on Linux 517–522
      - preparing apps for deployment 520–522
      - running app behind reverse proxy 517–519
  - HostingEnvironment 334
  - hostname 19
  - href attribute 248
  - HSTS (HTTP Strict Transport Security) 581
  - HstsMiddleware 48, 581–582
  - HTML
    - building, using view model 101
    - creating Markdown Tag Helper to convert content to 647–649
    - generating with Razor Pages 51–53
    - rendering using Razor views 188–222
      - creating dynamic web pages 199–206
      - creating Razor views 193–199
      - layouts, partial views, and \_ViewStart 206–215
      - views 189–193
    - rendering with Raw 204–206
  - HtmlEncoder 644
  - HtmlHelper() method 206
  - HTTP
    - calling HTTP APIs 668–674
    - handling HTTP verbs with attribute routing 274–276
    - handling transient HTTP errors with Polly 681–684
    - using typed clients to encapsulate HTTP calls 679–681
  - HttpClient 737
    - calling HTTP APIs 668–674
    - creating with IHttpClientFactory 674–681
      - configuring named clients at registration time 677–679

- HttpClient (*continued*)**
- managing HttpClientHandler lifetime 674–677
  - using typed clients to encapsulate HTTP calls 679–681
- HttpClientHandler** 674–677
- HttpContext** 22, 28, 50, 63, 128, 197, 405, 407, 418, 421, 438, 520, 612, 615, 622, 653, 729
- HttpContext.User** 438, 440–441, 455, 467, 475
- HttpMessageHandler** 684–686
- HttpRequestException** 683
- HttpRequestMessage** 684
- HTTPS** 573–584
  - ASP.NET Core HTTPS development
    - certificates 576–578
  - configuring Kestrel with production HTTPS certificate 579–580
  - enforcing for whole app 580–584
    - HTTPS redirection middleware 582–584
    - with HTTP Strict Transport Security headers 581–582
- HttpsRedirectionMiddleware** 48, 582–583
- HTTP Strict Transport Security headers** 581–582
- 
- I**
- IActionFilter** 420, 422
- IActionResult** 54, 116–117, 119, 259, 402, 405, 407, 418, 424–426, 429, 653
- IApApplicationBuilder** 46, 67, 72–73, 611, 614, 623
- IAsyncActionFilter** 422
- IAuthorizationRequirement** 486–487
- IAuthorizationService** 493–495
- IClassFixture** 743
- IConfiguration** 339–341, 346–351, 356, 561, 629–630, 632
- IConfigureOptions** 629, 632–635
- ICurrencyProvider** 635, 658, 662
- id** attribute 238
- IDE** (integrated development environment) 752
- Identity** (ASP.NET Core Identity)
  - adding custom data to users 466–468
  - adding to existing projects 458–462
    - configuring services and middleware 459–460
  - updating EF Core data model to support Identity 460–461
  - updating Razor views to link to Identity UI 461–462
- creating projects that use 448–458
  - data model 453–455
  - exploring templates in Solution Explorer 450–453
  - from templates 448–450
  - interacting with Identity 455–458
- customizing page in default UI 463–465
- defined 446–448
- IdentityDbContext** 453, 460
- identity provider** 443
- IdentityUser entity** 452–453, 455, 466
- IDEs and editors** 753–757
  - JetBrains Rider 755
  - Visual Studio 753–755
  - Visual Studio Code 756–757
  - Visual Studio for Mac 756
- IDisposable** 669
- IEndpointConventionBuilder** 627
- IEndpointRouteBuilder** 624
- IEnumerable** 242–243
- <if>** element 646
- if** attribute 646
- IFilterFactory** 432–433
- if loop** 201–202
- IFormFile**, binding file uploads with 170–172
- if statement** 55, 201–202, 251, 500
- IfTagHelper** 646
- IHost** 39–43, 696–703, 736
  - creating worker service from templates 698–700
  - running worker service in production 700–703
- IHostBuilder** 40–43, 45, 47, 335, 554, 556, 630, 636, 700, 702
- IHostBuilder.ConfigureLogging()** method 553
- IHostBuilder.ConfigureWebHostDefaults()** method 514
- IHostedService**, running background tasks
  - with 690–696
  - on timer 691–694
  - using scoped services 694–696
- IHostEnvironment** 355–358
- IHostEnvironment.EnvironmentName** property 355
- IHttpClientFactory** 667, 699–700, 740
  - calling HTTP APIs 668–674
  - creating custom HttpMessageHandler 684–686
  - creating HttpClient with 674–681
    - configuring named clients at registration time 677–679
    - using IHttpClientFactory to manage HttpClientHandler lifetime 674–677
    - using typed clients to encapsulate HTTP calls 679–681
  - handling transient HTTP errors with Polly 681–684
- IIS** (Internet Information Services), publishing apps to 511–517
  - configuring IIS for ASP.NET Core 512–514
  - preparing and publishing 514–517
- IJob** 704, 706–707
- IL** (Intermediate Language) 13

ILogger 543, 545, 547, 549–550, 554, 556, 558  
ILoggerFactory 543, 550, 553, 556–559  
ILoggerProvider 543, 545, 552, 556  
<img> element 250  
imperative authorization 493  
implementation type 306  
IMvcBuilder object 279, 283  
Index() method 218, 263, 271  
[index] 170  
IndexModel 107, 546  
Index Razor Page 249  
Information log level 547–548, 560  
information resources  
    announcement blog posts 777  
    ASP.NET Core blogs 779–780  
    ASP.NET Core GitHub repositories 779  
    books 776–777  
    Microsoft documentation 778  
    security-related links 778  
    tooling and services 779  
    video links 780–781  
@inject directive 197, 318  
[InlineData] attributes 726  
<input> element 226, 231–232, 236  
<input> type 239  
input element 237  
InputFormatter 173  
input formatters 268  
InputModel class 185  
Input property 116, 179, 229  
Input Tag Helper 236–240  
insecure direct object references 605  
INSERT statement 387  
integration testing 734–744  
    creating TestServer using Test Host  
        package 735–738  
    reducing duplication by creating custom  
        WebApplicationFactory 742–744  
    replacing dependencies in  
        WebApplicationFactory 740–742  
    testing apps with WebApplicationFactory  
        738–740  
int parameter 166  
int type 167, 759  
Invoke() method 619–620  
InvokeAsync() method 652–653  
IOptions interface  
    binding without 353–354  
    overview 349–350  
    using services to configure with  
        IConfigureOptions 632–635  
IOptionsSnapshot 350–351  
IOrderedFilter 412–413  
IOutputFormatter 282, 284  
IPageFilter 428

IQueryable 387  
IResourceFilter 406  
IResultFilter 420  
IsDeleted flag 394  
IServiceCollection 45, 705  
IServiceProvider 309, 695  
IServiceProvider.CreateScope() method 695  
IsValid() method 182, 656–657  
IUrlHelper 144–148  
IWebHostEnvironment 46–47, 358

## J

---

JetBrains Rider 755  
jobs, in Quartz.net 704  
JWT (JSON Web Tokens) 684

## K

---

Kestrel 579–580  
key files 592

## L

---

<label> element 235  
Label Tag Helper 235–236  
Language Integrated Query (LINQ) 13  
Last-Modified header 426  
layouts 53  
    overriding parent layouts using sections  
        209–211  
    using for shared markup 207–209  
<li> element 196  
lifetime of services 318–327  
    captured dependencies 324–327  
    scoped 322–323  
    singleton 323–324  
    transient 321–322  
<link> element 250  
LinkGenerator class 147–148  
LINQ (Language Integrated Query) 13  
Linux, hosting apps on 517–522  
    preparing apps for deployment 520–522  
    running app behind reverse proxy on  
        Linux 517–519  
List<string> type 169  
LocalRedirect() method 602  
Log() method 543  
logging 539–571  
    adding log messages to apps 545–551  
    formatting messages and capturing parameter  
        values 550–551  
log category 549–550  
log level 547–548  
    changing log verbosity with filtering 559–564

logging (*continued*)  
 logging providers 552–559  
   adding to apps 553–555  
   replacing default `ILoggerFactory` with  
     Serilog 556–559  
 structured logging 564–570  
   adding to apps 565–568  
   using scopes to add additional properties to  
     logs 568–570  
 using effectively in production apps 541–544  
   ASP.NET Core logging abstractions 543–544  
   custom log messages 549–543  
`LogInformation()` method 546, 548  
`log level` 546–547  
`LogWarning()` method 548  
 loops, adding to Razor templates 201–204  
 loosely coupled code, creating 298–300  
 LTS (Long Term Support) 12, 17, 763

**M**

`Main()` method 40  
`Main` entry point 39  
 managed app pool 512  
`Map()` method 613, 621, 624, 626  
`MapControllers()` 262  
 Map extension, branching middleware pipelines  
   with 612–615  
`MapGet()` method 127, 625–627  
 Map middleware 614  
`MapPost()` method 626  
`MapPut()` method 626  
`MapRazorPages` 50, 104, 107, 127, 132  
`<markdown>` element 647–648  
 Markdown Tag Helper 647–649  
`[MemberData]` attribute 726  
 metadata 176  
 method attribute 233  
 method parameters, simplifying by binding to  
   complex objects 168–169  
 Microsoft  
   announcement blog posts 777  
   documentation 778  
 middleware 58–90  
   combining in pipeline 64–76  
     holding pages 65–68  
     Razor Pages app 72–76  
     static files 68–71  
   custom pipeline components 610–621  
     adding to pipeline 616–618  
     branching middleware pipelines 612–615  
     building 619–621  
     creating simple endpoints 611–612  
   defined 60–64  
   defining how requests are handled with 45–50

error handling 77–89  
   handling exceptions in production 80–84  
   status codes 84–88  
   viewing exceptions in development 78–80  
   Web APIs and 89  
 filters vs. 404–406  
 HTTPS redirection 582–584  
   unit testing 728–731  
 migrations 378–384  
   adding 382–384  
   creating 379–382  
 minification  
   adding BundlerMinifier to apps 529–532  
   optimizing client-side assets using 525, 533–534  
   speeding up apps using 527–529  
   using minified files in production with Environment Tag Helper 532–533  
`[ModelBinder]` attribute 174  
 ModelBinders 172  
 model binding 94, 115, 161  
 Model component 95, 157  
`@model` directive 52, 194, 197–198, 219, 465  
 Model property 198, 219  
`ModelStateDictionary` object 178  
`ModelState.IsValid` property 733  
`ModelState` property 116, 159, 178–179, 268, 733  
 model validation, handling user input with  
   174–184  
     need for validation 174–176  
     using `DataAnnotations` attributes 176–178  
     validating on client for user experience 182–184  
     validating on server for safety 178–182  
 Model-View-View Model (MVVM) 97  
 multiple method parameter 166  
 MultiValues 242  
`Must()` method 661  
 MVC design pattern 94–97  
   applying to Razor Pages 97–104  
     building HTML using view model 101  
     complete request 102–104  
     directing requests to Razor Page and building  
       binding model 99–100  
     executing handler using app model 100–101  
     applying to Web API 266–270  
     compared to Razor Pages 107–113  
       MVC controllers 108–110, 112–113  
       Razor Pages 110–112  
     creating Web APIs using 255–287  
       applying MVC design pattern 266–270  
       attribute routing 270–276  
       creating projects 259–266  
       generating response from model 280–285  
       overview 256–259  
       using common conventions with  
         `[ApiController]` attribute 276–280

MVC design pattern (*continued*)  
generating URLs from route parameters for  
    MVC controller 145–146  
models in 157–160  
selecting views from MVC controllers 215–221  
MVC filter pipeline 401–402  
MVVM (Model-View-View Model) 97

## N

---

nameof operator 146, 349, 422  
.NET 5.0  
    as step in One .NET vision 762–763  
    evolution of .NET into 759–764  
.NET 6.0 763–764  
.NET Core  
    .NET platforms that prompted 759–760  
    overview 761–762  
.NET ecosystem 758–775  
    evolution of .NET into .NET 5.0 759–764  
    future of .NET Standard 773–774  
    sharing code between projects 764–773  
.NET Framework developers  
    experienced, creating new apps 14–18  
    new 12–14  
.NET SDK 752–753  
.NET Standard 766–771  
    future of 773–774  
    .NET Standard 2.0 support with compatibility  
        shim 772–773  
No Managed Code pool 512  
NotEmpty() method 660  
NotFound() method 119  
NotFoundResult 119, 266  
NullReferenceException 77

## O

---

ObjectInstance property 657  
object-relational mappers (ORMs) 367–368  
Ok() method 264  
OkObjectResult 426  
OkResult 266, 281  
OnActionExecuted() method 420, 423  
OnActionExecuting() method 421  
OnActionExecuting ControllerBase()  
    method 423  
OnConfiguring() method 378  
One .NET vision 762–763  
OnGet() handler 54–55, 94, 99, 107, 116, 140,  
    149–150, 165, 186, 275, 546  
OnHead() handler 151  
OnPageHandlerExecuting() method 428  
OnPost() handler 116, 162, 179, 186, 275,  
    466

OnPostAsync() handler 149–150  
OnPostCustomSearch() handler 149–150  
OnResourceExecuted() method 406  
OnResourceExecuting() method 406  
OnResultExecuted() method 426  
OpenReadStream() method 171  
open redirect attacks 601–603  
<optgroup> element 243–244  
<option> element 242  
Options object 280, 340  
Options pattern, strongly typed configuration  
    with 347–354  
binding without IOptions interface 353–354  
designing options classes for automatic  
    binding 351–353  
IOptions interface 349–350  
    reloading strongly typed options 350–351  
OPTIONS verb 275, 595  
Order property 412, 647  
origins 593  
ORMs (object-relational mappers) 367–368  
output formatters 270, 283

## P

---

PaaS (Platform as a Service) 506  
PackageReference element 33, 374  
Page() method 94, 119, 145–147  
page-based websites 93  
@page directive 51–53, 136–138, 142, 153, 166,  
    194, 198, 270  
page filters 427–429  
page handlers 189, 465  
    handling request logic with 53–55  
    injecting services directly into 315–317  
    Razor Pages and 113–120  
        accepting parameters to page handlers  
            115–117  
        returning responses with ActionResults  
            117–120  
        selecting to invoke 149–151  
PageHandlerSelected() method 412  
Page helper() method 179  
PageModel 52–55, 93, 96, 98–99, 101, 108, 116,  
    119, 133, 144, 156, 159–161, 168, 178, 185,  
    188, 191, 194, 196, 198, 201, 212, 227, 229,  
    234–236, 241–242, 268–269, 317–318, 348,  
    409, 465, 481, 499–500, 602, 642, 665, 731,  
    733  
page model 160  
PageResponseCacheFilter 427  
PageResult 114, 118–119, 179, 217  
@page statement 136  
parameterization 603–604  
<partial /> Tag Helper 212

## partial views

- adding logic to 649–654
- encapsulating markup with 211–213
- password protection 605–606
- path 19, 21
- PathBase 615
- Path property 615
- PCLs (Portable Class Libraries) 759, 764–766
- pipeline 12, 60
- Platform as a Service (PaaS) 506
- PMC (Package Manager Console) 379
- policies, authorization
  - custom 484–491
  - overview 481–484
  - requirements and handlers 484–491
    - creating authorization handlers to satisfy requirements 488–491
    - creating IAuthorizationRequirement to represent requirement 486–487
    - creating policy with multiple requirements 487–488
- policyBuilder 482
- Polly, handling transient HTTP errors with 681–684
- Portable Class Libraries (PCLs) 759, 764–766
- POST request 116, 150, 161–162, 184, 588, 591
- POST verb 54, 189, 233, 275
- primary key 375
- ProblemDetails object 256, 265, 277–278, 424
- Process() method 643–644, 646
- ProcessAsync() method 643
- ProducesAttribute 425
- production apps
  - using EF Core in 394–396
  - using logging effectively in 541–544
    - ASP.NET Core logging abstractions 543–544
      - custom log messages 542–543
- Program class 39–42, 719, 736
- Program.CreateHostBuilder() method 739
- Program.cs 338–340, 698
- publish command 516
- PUT request 161

## Q

- Quartz.NET, coordinating background tasks using 703–711
- configuring job to run on schedule 706–708
- installing Quartz.NET 704–705
- using clustering to add redundancy to background tasks 708–711
- {query} route parameter 160
- query string values 162

## R

- 
- Raw, rendering HTML with 204–206
- Razor Pages 50–121
  - adding filters to 408–410
  - adding to apps 104–107
  - customizing conventions with 151–153
  - customizing model binding with page filters 427–429
  - generating HTML 51–53
  - generating URLs from route parameters 144–145
  - handling request logic with PageModels and handlers 53–55
  - models in 157–160
  - MVC design pattern
    - applying to Razor Pages 97–104
    - compared to Razor Pages 107–113
    - overview 94, 97
  - organizing binding models in 184–186
  - overview 93–94
  - page handlers and 113–120
    - accepting parameters to page handlers 115–117
    - returning responses with ActionResults 117–120
  - route templates
    - adding additional constraints to parameters 139–142
    - adding segments to template 136–137
    - customizing 134–138
    - matching arbitrary URLs with catch-all parameter 142–143
    - optional and default values 138–139
    - replacing template completely 137–138
    - syntax 138–143
  - routing to 132–134- Razor Pages filter pipeline 403–404
- Razor templates 195–196
  - adding loops and conditionals to 201–204
  - hiding elements from unauthorized users 498–501
  - using C# in 200–201
- Razor views 188–222
  - creating 193–199
    - code-behind 194–195
    - passing data to views 197–199
    - Razor templates 195–196
  - creating dynamic web pages 199–206
    - adding loops and conditionals to Razor templates 201–204
    - rendering HTML with Raw 204–206
    - using C# in Razor templates 200–201
  - encapsulating markup with partial views 211–213

Razor views (*continued*)  
    importing common directives 213–214  
    layouts  
        overriding parent layouts using sections 209–211  
        using for shared markup 207–209  
    running code for every view 214–215  
    selecting views from MVC controllers 215–221  
    updating to link to Identity UI 461–462  
    views 189–193  
RDD (runtime-dependent deployments) 509  
ReadFromJsonAsync() method 627  
RedirectResult 117  
RedirectToAction 147  
RedirectToPage() method 116, 119, 146–147, 179  
RedirectToPageResult 118–119, 179  
reflection 43  
registration 300  
[RegularExpression] attribute 655  
Release configuration 508  
@RenderBody() method 208–209  
@RenderSection() method 210  
Replace() method 314  
Repository 319–322, 324  
RequestDelegate 619, 729–730  
Request.Scheme 520  
RequestServices.GetService() 421  
RequireAssertion() method 484  
RequireAuthorization() method 627–628  
RequireAuthorization(policy) 628  
RequireClaim() method 483  
RequireCors(policy) method 628  
[Required] attribute 179, 642, 655  
RequireHost(hosts) method 628  
resource-based authorization 492–498  
    creating resource-based  
        AuthorizationHandler 495–498  
    manually authorizing requests with  
        IAuthorizationService 493–495  
ResourceExecutedContext method 407  
resource filters 416–418  
Response.Body object 730  
ResponseCacheFilter 419, 427  
REST (representational state transfer) 10  
ResultFilterAttribute class 426  
result filters 425–427  
Result property 421  
return Page() 94  
return statement 94  
reverse proxy 23, 505, 520  
[Route] attributes 130, 271–272, 275  
RouteAttribute 271, 273  
RouteOptions object 151

route parameter 135–136  
route templates 128, 132  
route values 136, 143, 162  
routing 122–155  
    convention-based routing vs. attribute  
        routing 130–132  
    customizing conventions with Razor Pages 151–153  
    customizing Razor Page route templates 134–138  
        adding segments to template 136–137  
        replacing template completely 137–138  
        defined 123–126  
        endpoint routing 127–130  
        generating URLs from route parameters 143–149  
        for MVC controllers 145–146  
        for Razor Pages 144–145  
        from other parts of app 147–149  
        with ActionResults 146–147  
    in ASP.NET Core 126–134  
    selecting page handler to invoke 149–151  
    template syntax 138–143  
        adding additional constraints to  
            parameters 139–142  
        matching arbitrary URLs with catch-all  
            parameter 142–143  
        optional and default values 138–139  
    to Razor Pages 132–134  
RoutingMiddleware 127–128, 133, 262, 268, 477, 622  
RuleFor() method 660  
Run() method 42  
Run extension 611–612  
Run method 611–612, 621  
runtime-dependent deployments (RDD) 509

## S

---

same-origin policy 593  
SaveChanges() method 392  
SaveChangesAsync() method 386, 392  
scaffolding 463  
SCD (self-contained deployments) 509  
schema 378  
scoped services 322–323, 694–696  
scopes, using to add additional properties to  
    logs 568–570  
    <script> element 250, 585  
Scripts section 210  
Sdk attribute 38  
search parameter 604  
secrets 331  
@section 209–210

security 572–608  
 cross-origin resource sharing 593–601  
 adding global policy to whole app 596–598  
 adding to specific Web API actions with  
*EnableCorsAttribute* 598–599  
 configuring policies 599–601  
 overview 594–596  
 cross-site request forgery attacks 588–593  
 cross-site scripting attacks 584–588  
**HTTPS** 573–584  
 ASP.NET Core HTTPS development  
 certificates 576–578  
 configuring Kestrel with production HTTPS  
 certificate 579–580  
 enforcing for whole app 580–584  
 insecure direct object references 605  
 open redirect attacks 601–603  
 protecting user passwords and data 605–606  
 security-related links 778  
 SQL injection attacks 603–604  
 security headers 580  
 seed data 396  
`Select()` method 387  
`<select>` element 242, 244  
`SelectListGroup` 243  
`SelectListItem` object 243  
`Select` method 389  
 Select Tag Helper 240–245  
`SendAsync()` method 684  
 Serilog 556–559  
 service configuration 302–318  
   adding ASP.NET Core framework services to  
     container 302–303  
   injecting services directly into action  
     methods 315–317  
   injecting services directly into page  
     handlers 315–317  
   injecting services into view templates 317–318  
 registering service multiple times 311–315  
   conditionally registering services 314–315  
   injecting multiple implementations of  
     interface 311–313  
   injecting single implementation when multi-  
     ple services are registered 313–314  
 registering services using objects and  
 lambdas 307–311  
   registering services with container 304–307  
**ServiceFilterAttribute** 431, 433  
 service registration 42  
**ServiceRegistry** 637  
`services.Configure<T>()` method 632, 634  
 short-circuiting  
   action methods with resource filters  
     416–418  
   filter pipelines 429–431  
 Sidebar 209–210  
**SignInAsync** 467  
**SignInManager** service 440  
 single method parameter 166  
**SingleOrDefaultAsync()** clause 389  
 single-page applications (SPAs) 10, 29, 92–93,  
   256  
 singleton services 323–324  
 Solution Explorer 450–453  
`<span>` element 230–231, 245  
 SPAs (single-page applications) 10, 29, 92–93,  
   256  
**SQL** injection attacks with Entity Framework  
 Core (EF Core) and parameterization  
   603–604  
**SqlLiteConnection** 747  
**SRP** (single responsibility principle) 44, 294, 659  
**StartAsync()** method 691, 737  
 Startup class 42, 65, 67, 302, 339, 345, 349, 381,  
   637, 663, 739  
   adding and configuring services 44–45  
   defining how requests are handled with  
     middleware 45–50  
 Startup.Configure() method 620–621, 625, 737  
 Startup.ConfigureServices() method 596, 632,  
   741, 745  
 StaticFileMiddleware 49, 68, 70, 130, 627  
 static void main() method 39  
 static void Main function 504  
 static void main method 719  
 StatusCodePagesMiddleware 84–88  
 StatusCodeResult 119, 259, 266, 281, 424  
 StatusMiddleware 729, 735, 737, 739  
 StreamReader 730  
 string array 283  
 string columns 395  
 string keys 197, 340, 347, 353, 629  
 [StringLength(min, max)] attribute 655  
 string type 167, 657, 759  
 strongly typed configuration 347–354  
   binding without `IOptions` interface 353–354  
   designing options classes for automatic  
     binding 351–353  
`IOptions` interface 349–350  
   reloading strongly typed options with  
     `IOptionsSnapshot` 350–351  
 structured logging 564–570  
   adding to apps 565–568  
   using scopes to add additional properties to  
     logs 568–570  
 Succeed() method 489–490, 496  
 support tracks 763  
 SuppressOutput() method 647  
`<system-info>` element 642, 644  
 SystemInfoTagHelper 643

**T**

TagHelper class 643  
TagHelperOutput 645  
Tag Helpers 189, 223–254  
  cache-busting with Append Version Tag Helper 250–251  
  catering to editors with 225–228  
  creating forms using 228–248  
    Form Tag Helper 233–235  
    Input Tag Helper 236–240  
    Label Tag Helper 235–236  
    Select Tag Helper 240–245  
    Textarea Tag Helper 236–240  
  Validation Message Tag Helper 245–248  
  Validation Summary Tag Helper 245–248  
custom Razor Tag Helpers 641–649  
  creating to conditionally hide elements 645–647  
  creating to convert Markdown to HTML 647–649  
  printing environment information 642–645  
generating links with Anchor Tag Helper 248–250  
  using conditional markup with Environment Tag Helper 251–252  
  using minified files in production with Environment Tag Helper 532–533  
TargetFramework element 38  
Tasks 201–202, 692  
task variable 204  
Test Host package 735–738  
testing apps 714–750  
  in ASP.NET Core 716–717  
  integration testing 734–744  
    creating TestServer using Test Host package 735–738  
    reducing duplication by creating custom WebappFactory 742–744  
    replacing dependencies in WebappFactory 740–742  
  testing apps with WebappFactory 738–740  
isolating database with in-memory EF Core provider 744–748  
unit testing API controllers 731–734  
unit testing custom middleware 728–731  
unit testing with xUnit 717–728  
  adding Fact and Theory unit tests 724–726  
  creating test projects 718–719  
  referencing app from test project 721–724  
  running tests with dotnet test 720–721  
  testing failure conditions 727–728  
TestServer, creating using Test Host package 735–738  
Textarea Tag Helper 236–240

Theory unit tests 724–726  
Timestamp columns 395  
ToListAsync() command 388  
ToString() method 281, 569  
Trace log level 548  
transient services 321–322  
TryAdd, conditionally registering services using 314–315  
TryAddScoped 314  
try-catch block 424  
try-catch-finally block 557  
two-factor authentication (2FA) 445  
TypeFilterAttribute 431–433

**U**

unit testing  
  API controllers 731–734  
  custom middleware 728–731  
  with xUnit 717–728  
    adding Fact and Theory unit tests 724–726  
    creating test projects 718–719  
    referencing app from test project 721, 724  
    running tests with dotnet test 720–721  
    testing failure conditions 727–728  
UpdateExchangeRatesJob 707  
Url helper 144–145  
Url.IsLocalUrl() method 602  
Url.Page() method 146  
Url property 144, 146  
URLs  
  configuring 522–524  
  generating from route parameters 143–149  
    for MVC controllers 145–146  
    for Razor Pages 144–145  
    from other parts of app 147–149  
    with ActionResults 146–147  
  linking action methods to 270–276  
  combining route attributes 272–274  
  handling HTTP verbs 274–276  
  using token replacement to reduce duplication 274  
  matching arbitrary URLs with catch-all parameter 142–143  
–urls parameter 524  
Use\* method 378  
UseAuthentication() 453, 476, 597  
UseAuthorization() 452–453, 460, 476, 621  
UseContentRoot 334  
UseCors() 597, 599  
UseEndpoints() method 75, 127, 262  
UseEndpoints method 452, 476, 597, 623–625  
Use extension, adding to middleware pipeline with 67, 73, 616–618  
UseHsts() method 584

UseHttpsRedirection() method 584  
 UseIIS() method 514, 521  
 UseIISIntegration() method 514, 520, 584  
 Use method 616–617, 619  
 user input, handling with model validation 174–184  
   need for validation 174–176  
   using DataAnnotations attributes 176, 178  
   validating on client for user experience 182–184  
   validating on server for safety 178–182  
 UserManager.AddClaimAsync() method 466  
 UseRouting() 452, 460, 476, 597  
 User property 438, 489  
 User Secrets Manager 344–346  
 UseSerilog() method 556  
 UseSqlite() method 747  
 UseSqlServer method 377  
 UseStartup<>() method 40  
 UseStatusCodePages 86  
 UseSystemd() method 700, 702  
 UseTestServer() method 737  
 UseWelcomePage() method 67  
 UseWindowsService() method 701–702  
 @using directive 214  
 UWP (Universal Windows Platform) 759, 765

**V**

[ValidateAntiForgeryToken] attributes 591  
 ValidateScopes option 325  
 ValidationAttribute 655–656  
 validation attributes 654–659  
 ValidationContext 657–658  
 validation framework, replacing with  
   FluentValidation 659–665  
   adding to apps 663–665  
   DataAnnotations attributes vs. 660–663  
 Validation Message Tag Helper 245–248  
 ValidationProblemDetails object 277–278  
 ValidationSummary enum value 246  
 Validation Summary Tag Helper 245–248  
 Value attribute 244  
 verbs 574  
 video links 780–781  
 View() method 653  
 View component 95, 157  
 [ViewComponent] attribute 653  
 ViewComponent class 652  
 ViewData 53, 197–199, 208  
 \_ViewImports 213–214  
 View method 547  
 view model, building HTML using 101  
 view rendering 650  
 ViewResult 117, 216, 218–219, 259

views

  overview 189–193  
   passing data to 197–199  
   selecting views from MVC controllers 215–221  
 \_ViewStart 214–215  
 view templates, injecting services into 317–318  
 Visual Studio 753–755  
 Visual Studio Code 756–757  
 Visual Studio for Mac 756  
 void() method 94, 107

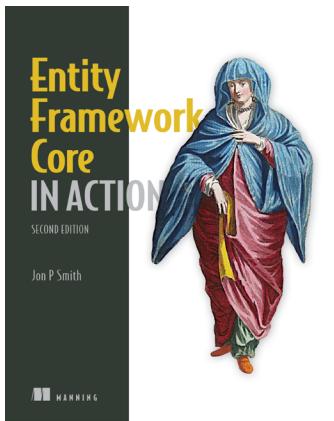
**W**


---

Warning log level 547–548, 560  
 WCF (Windows Communication Foundation) 18  
 Web APIs 255–287  
   applying MVC design pattern to 266–270  
   attribute routing 270–276  
   combining route attributes 272–274  
   handling HTTP verbs with 274–276  
   using token replacement to reduce duplication in 274  
   creating projects 259–266  
   error handling middleware and 89  
   generating response from model 280–285  
   adding XML support 282–284  
   choosing response format with content negotiation 284–285  
   overview 256–259  
   using common conventions with  
     [ApiController] attribute 276–280  
 WebappFactory  
   reducing duplication by creating custom 742–744  
   replacing dependencies in 740–742  
   testing apps with 738–740  
 WebApplicationFactory 738–741  
 web frameworks 5  
 WebHostBuilder object 41  
 web requests  
   complete Razor Page request 102–104  
   defining how requests are handled with middleware 45–50  
   directing to Razor Pages 99–100  
   generating responses 50–55  
     generating HTML 51–53  
     handling request logic 53–55  
   handling with middleware pipeline 58–90  
     combining in pipeline 64–76  
     defined 60–64  
     error handling 77–89  
   inner workings of 19–21  
   making useful through binding model 160–174  
   processing 21–23

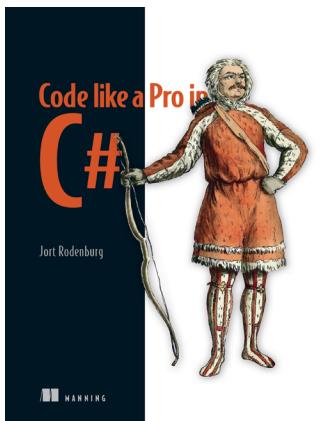
- web requests (*continued*)  
    returning responses with ActionResult 117–120  
        creating ActionResult classes using helper  
            methods 119–120  
    NotFoundResult 119  
    PageResult 118–119  
    RedirectToPageResult 118–119  
    StatusCodeResult 119  
WebRootPath property 47, 358  
WelcomePageMiddleware 65, 75  
Where() clause 387, 394  
Windows Communication Foundation (WCF) 18  
WithDisplayName(name) method 628  
WithMetadata(items) method 629  
WithOrigins() method 600  
WithWebHostBuilder 741  
WithWebHostBuilder() 742, 744  
Worker.cs 698
- worker services 696–703  
    creating from templates 698–700  
    running in production 700–703  
workloads 754  
WriteAsJsonAsync 627  
WriteJsonAsync, creating simple endpoints  
    with 625–627
- 
- X**
- XML support 282–284  
XSS (cross-site scripting) attacks 584–588  
xUnit 717–728  
    adding Fact and Theory unit tests 724–726  
    creating test projects 718–719  
    referencing app from test project 721, 724  
    running tests with dotnet test 720–721  
    testing failure conditions 727–728

## RELATED MANNING TITLES



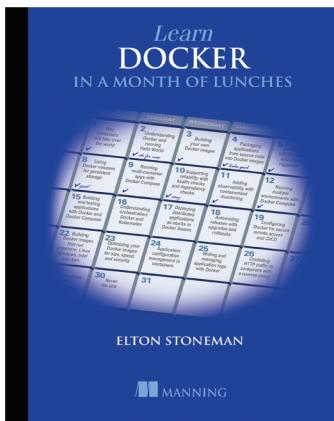
*Entity Framework Core in Action, 2nd Ed.*  
by Jon P Smith

ISBN 9781617298363  
575 pages (estimated), \$59.99  
Summer 2021 (estimated)



*Code like a Pro in C#*  
by Jort Rodenburg

ISBN 9781617298028  
425 pages (estimated), \$59.99  
Summer 2021 (estimated)



*Learn Docker in a Month of Lunches*  
by Elton Stoneman

ISBN 9781617297052  
464 pages, \$49.99  
June 2020

*For ordering information go to [www.manning.com](http://www.manning.com)*

**Useful .NET CLI (dotnet) Commands. Use `--help` to see all the optional arguments**

Command	Run from	Description
<code>dotnet restore</code>	Solution folder	Restores the NuGet packages for all the projects in the solution.
<code>dotnet build</code>	Solution folder	Builds all the projects in the solution. To build in Release mode, use the <code>-c</code> switch.
<code>dotnet run</code>	Project folder	Runs the project in the current folder. Use during development to run your app.
<code>dotnet publish -c Release -o &lt;Folder&gt;</code>	Project folder	Publishes the project to the provided folder. This copies all the required files to the provided output folder, so it can be deployed.
<code>dotnet test</code>	Solution folder	Restores packages, builds, and executes any unit tests found in the solution. Requires the .NET Test SDK and a testing framework adapter (see chapter 28).
<code>dotnet add package &lt;Name&gt;</code>	Project folder	Install the NuGet package with the provided name into the current project. Optionally specify a package version, for example <code>-v 2.1.0</code> .
<code>dotnet new --list</code>	Anywhere	View all the installed templates for creating ASP.NET Core apps, libraries, test projects, solution files, and many more.
<code>dotnet new --install &lt;PackageName&gt;</code>	Anywhere	Installs a new template from the provided NuGet package name. For example, <code>dotnet new --install "NetEscapades.Templates::*"</code> .
<code>dotnet new &lt;Template&gt; -o &lt;Folder&gt; -n &lt;NewName&gt;</code>	Anywhere, empty folder for new projects	Create a new item from the provided template, specifying the folder in which to place the item, and the name for the item.
<code>dotnet --info</code>	Anywhere	Displays information about the .NET Core SDKs and runtimes installed on your machine.
<code>dotnet ef add migrations &lt;Name&gt;</code>	Project folder	Adds a new EF Core migration to the project with the provided name. Requires the EF Core global tool (see Chapter 12).
<code>dotnet ef database update</code>	Project folder	Apply pending EF Core migrations to a database. Warning – this will modify your database!

# ASP.NET Core IN ACTION Second Edition

Andrew Lock

**B**uild full-stack web applications that run anywhere. Developers love ASP.NET Core for its libraries and pre-built components that maximize productivity. Version 5.0 offers new features for server-side apps, as well as background services for cross-platform development.

**ASP.NET Core in Action, Second Edition** is a comprehensive guide to creating web applications with ASP.NET Core 5.0. Go from basic HTTP concepts to advanced framework customization. Illustrations and annotated code make learning visual and easy. Master logins, dependency injection, security, and more. This updated edition covers the latest features, including Razor Pages and the new hosting paradigm.

## What's Inside

- Developing apps for Windows and non-Windows servers
- Configuring applications
- Building custom components
- Logging, testing, and security

For intermediate C# developers.

**Andrew Lock** is a Microsoft MVP who has worked with ASP.NET Core since before its first release.

Register this print book to get free access to all ebook formats.

Visit <https://www.manning.com/freebook>

“One of the greatest and most complete books about ASP.NET Core!”

—Delcoigne Vincent, Wavenet

“Fantastic book. The topics are explained clearly and thoroughly.”

—Luis Moux, EMO

“A comprehensive reference for ASP.NET Core.”

—Jean-François Morin  
Laval University

“The most comprehensive ASP.NET Core book on the market. It covers just about everything you need to learn to quickly become productive in the often confusing and fast-changing world of .NET Core.”

—Filip Wojcieszyn  
Sonova AG



See first page

ISBN: 978-1-61729-830-1



56999

9 781617 298301



MANNING

\$69.99 / Can \$92.99 [INCLUDING eBOOK]