



Pro ASP.NET Core 3

Develop Cloud-Ready Web Applications
Using MVC, Blazor, and Razor Pages

—
Eighth Edition

—
Adam Freeman

Apress®

Pro ASP.NET Core 3

Develop Cloud-Ready Web Applications Using MVC,
Blazor, and Razor Pages

Eighth Edition



Adam Freeman

Apress®

Pro ASP.NET Core 3: Develop Cloud-Ready Web Applications Using MVC, Blazor, and Razor Pages

Adam Freeman
London, UK

ISBN-13 (pbk): 978-1-4842-5439-4
<https://doi.org/10.1007/978-1-4842-5440-0>

ISBN-13 (electronic): 978-1-4842-5440-0

Copyright © 2020 by Adam Freeman

This work is subject to copyright. All rights are reserved by the Publisher, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, reuse of illustrations, recitation, broadcasting, reproduction on microfilms or in any other physical way, and transmission or information storage and retrieval, electronic adaptation, computer software, or by similar or dissimilar methodology now known or hereafter developed.

Trademarked names, logos, and images may appear in this book. Rather than use a trademark symbol with every occurrence of a trademarked name, logo, or image we use the names, logos, and images only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

The use in this publication of trade names, trademarks, service marks, and similar terms, even if they are not identified as such, is not to be taken as an expression of opinion as to whether or not they are subject to proprietary rights.

While the advice and information in this book are believed to be true and accurate at the date of publication, neither the authors nor the editors nor the publisher can accept any legal responsibility for any errors or omissions that may be made. The publisher makes no warranty, express or implied, with respect to the material contained herein.

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Joan Murray
Development Editor: Laura Berendson
Editorial Operations Manager: Mark Powers

Cover designed by eStudioCalamar

Cover image designed by Freepik (www.freepik.com)

Distributed to the book trade worldwide by Apress Media, LLC, 1 New York Plaza, New York, NY 10004, U.S.A. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail editorial@apress.com; for reprint, paperback, or audio rights, please email bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484254394. For more detailed information, please visit www.apress.com/source-code.

Printed on acid-free paper

*Dedicated to my lovely wife, Jacqui Griffyth.
(And also to Peanut.)*

Table of Contents

■ About the Author	xxvii
■ About the Technical Reviewer	xxix
■ Part I: Introducing ASP.NET Core	1
■ Chapter 1: Putting ASP.NET Core in Context	3
Understanding ASP.NET Core.....	3
Understanding the Application Frameworks.....	3
Understanding the Utility Frameworks	4
Understanding the ASP.NET Core Platform	5
Understanding This Book	5
What Software Do I Need to Follow the Examples?.....	5
What Platform Do I Need to Follow the Examples?	5
What If I Have Problems Following the Examples?.....	6
What If I Find an Error in the Book?.....	6
What Does This Book Cover?	6
What Doesn't This Book Cover?.....	7
How Do I Contact the Author?.....	7
What If I Really Enjoyed This Book?.....	7
What If This Book Has Made Me Angry and I Want to Complain?	7
Summary	7
■ Chapter 2: Getting Started	9
Choosing a Code Editor	9
Installing Visual Studio	10
Installing Visual Studio Code	12

Creating an ASP.NET Core Project	16
Opening the Project Using Visual Studio.....	17
Opening the Project with Visual Studio Code.....	19
Running the ASP.NET Core Application	20
Understanding Endpoints	21
Understanding Routes	22
Understanding HTML Rendering.....	23
Putting the Pieces Together	27
Summary	27
■ Chapter 3: Your First ASP.NET Core Application	29
Setting the Scene	29
Creating the Project.....	29
Adding a Data Model	31
Creating a Second Action and View	31
Linking Action Methods	32
Building the Form	33
Receiving Form Data	35
Adding the Thanks View	37
Displaying the Responses.....	38
Adding Validation	40
Styling the Content	45
Summary.....	50
■ Chapter 4: Using the Development Tools	51
Creating ASP.NET Core Projects.....	51
Creating a Project Using the Command Line	52
Creating a Project Using Visual Studio	54
Adding Code and Content to Projects.....	57
Understanding Item Scaffolding	58
Building and Running Projects	59
Building and Running Projects Using the Command Line	60
Building and Running Projects Using Visual Studio Code.....	61
Building and Running Projects Using Visual Studio	61

Managing Packages	62
Managing NuGet Packages.....	62
Managing Tool Packages	63
Managing Client-Side Packages.....	63
Managing Packages Using Visual Studio.....	65
Debugging Projects	66
Summary	67
■ Chapter 5: Essential C# Features.....	69
Preparing for This Chapter	69
Opening the Project.....	70
Enabling the MVC Framework	70
Creating the Application Components	71
Selecting the HTTP Port.....	72
Running the Example Application	72
Using the Null Conditional Operator	73
Chaining the Null Conditional Operator	74
Combining the Conditional and Coalescing Operators	75
Using Automatically Implemented Properties	77
Using Auto-implemented Property Initializers.....	78
Creating Read-Only Automatically Implemented Properties.....	78
Using String Interpolation.....	80
Using Object and Collection Initializers	81
Using an Index Initializer	82
Pattern Matching.....	83
Pattern Matching in switch Statements	84
Using Extension Methods	85
Applying Extension Methods to an Interface	86
Creating Filtering Extension Methods.....	88
Using Lambda Expressions	89
Defining Functions.....	90
Using Lambda Expression Methods and Properties	93
Using Type Inference and Anonymous Types.....	95
Using Anonymous Types	95
Using Default Implementations in Interfaces	97

Using Asynchronous Methods	99
Working with Tasks Directly	99
Applying the async and await Keywords	100
Using an Asynchronous Enumerable	102
Getting Names.....	105
Summary	106
■ Chapter 6: Testing ASP.NET Core Applications.....	107
Preparing for This Chapter	107
Opening the Project	108
Selecting the HTTP Port.....	108
Enabling the MVC Framework	109
Creating the Application Components	109
Running the Example Application	111
Creating a Unit Test Project	111
Removing the Default Test Class	112
Writing and Running Unit Tests	112
Running Tests with the Visual Studio Test Explorer	114
Running Tests with Visual Studio Code.....	114
Running Tests from the Command Line.....	115
Correcting the Unit Test	115
Isolating Components for Unit Testing.....	116
Using a Mocking Package	120
Creating a Mock Object	121
Summary	122
■ Chapter 7: SportsStore: A Real Application.....	123
Creating the Projects.....	123
Creating the Unit Test Project	124
Creating the Application Project Folders	124
Opening the Projects	125
Preparing the Application Services and the Request Pipeline.....	125
Configuring the Razor View Engine	127
Creating the Controller and View	128
Starting the Data Model.....	128
Checking and Running the Application.....	129

Adding Data to the Application	129
Installing the Entity Framework Core Packages	129
Defining the Connection String	130
Creating the Database Context Class	130
Configuring Entity Framework Core	131
Creating a Repository	132
Creating the Database Migration	134
Creating Seed Data	135
Displaying a List of Products	137
Preparing the Controller	138
Updating the View	139
Running the Application	140
Adding Pagination	140
Displaying Page Links	142
Improving the URLs	149
Styling the Content	151
Installing the Bootstrap Package	151
Applying Bootstrap Styles	152
Creating a Partial View	154
Summary	155
■ Chapter 8: SportsStore: Navigation and Cart	157
Adding Navigation Controls	157
Filtering the Product List	157
Refining the URL Scheme	161
Building a Category Navigation Menu	165
Correcting the Page Count	171
Building the Shopping Cart	174
Configuring Razor Pages	174
Creating a Razor Page	176
Creating the Add To Cart Buttons	177
Enabling Sessions	178
Implementing the Cart Feature	180
Summary	188

■ Chapter 9: SportsStore: Completing the Cart	189
Refining the Cart Model with a Service	189
Creating a Storage-Aware Cart Class	189
Registering the Service	191
Simplifying the Cart Razor Page	192
Completing the Cart Functionality	194
Removing Items from the Cart	194
Adding the Cart Summary Widget	196
Submitting Orders	199
Creating the Model Class	199
Adding the Checkout Process	200
Creating the Controller and View	201
Implementing Order Processing	204
Completing the Order Controller	206
Displaying Validation Errors	209
Displaying a Summary Page	211
Summary	212
■ Chapter 10: SportsStore: Administration	213
Preparing Blazor Server	213
Creating the Imports File	214
Creating the Startup Razor Page	215
Creating the Routing and Layout Components	215
Creating the Razor Components	216
Checking the Blazor Setup	217
Managing Orders	218
Enhancing the Model	218
Displaying Orders to the Administrator	219
Adding Catalog Management	222
Expanding the Repository	222
Applying Validation Attributes to the Data Model	223
Creating the List Component	224
Creating the Detail Component	226
Creating the Editor Component	227
Deleting Products	230
Summary	231

- **Chapter 11: SportsStore: Security and Deployment** 233
 - Securing the Administration Features 233
 - Creating the Identity Database 233
 - Adding a Conventional Administration Feature..... 238
 - Applying a Basic Authorization Policy..... 239
 - Creating the Account Controller and Views 241
 - Testing the Security Policy..... 244
 - Preparing ASP.NET Core for Deployment 244
 - Configuring Error Handling 244
 - Creating the Production Configuration Settings 246
 - Creating the Docker Image 246
 - Running the Containerized Application..... 248
 - Summary 250
- **Part II: The ASP.NET Core Platform** 251
 - **Chapter 12: Understanding the ASP.NET Core Platform**..... 253
 - Preparing for This Chapter 254
 - Running the Example Application 255
 - Understanding the ASP.NET Core Platform 255
 - Understanding Middleware and the Request Pipeline..... 255
 - Understanding Services..... 256
 - Understanding the ASP.NET Core Project 256
 - Understanding the Entry Point..... 258
 - Understanding the Startup Class..... 259
 - Understanding the Project File 260
 - Creating Custom Middleware 261
 - Defining Middleware Using a Class 265
 - Understanding the Return Pipeline Path..... 267
 - Short-Circuiting the Request Pipeline 268
 - Creating Pipeline Branches 270
 - Creating Terminal Middleware..... 272
 - Configuring Middleware 274
 - Using the Options Pattern with Class-Based Middleware 277
 - Summary 278

Chapter 13: Using URL Routing	279
Preparing for This Chapter	280
Understanding URL Routing.....	282
Adding the Routing Middleware and Defining an Endpoint	283
Understanding URL Patterns.....	286
Using Segment Variables in URL Patterns	287
Generating URLs from Routes.....	291
Managing URL Matching	294
Matching Multiple Values from a Single URL Segment.....	294
Using Default Values for Segment Variables.....	296
Using Optional Segments in a URL Pattern.....	296
Using a catchall Segment Variable	298
Constraining Segment Matching	299
Defining Fallback Routes.....	302
Advanced Routing Features	303
Creating Custom Constraints.....	303
Avoiding Ambiguous Route Exceptions.....	305
Accessing the Endpoint in a Middleware Component	307
Summary	308
Chapter 14: Using Dependency Injection	309
Preparing for This Chapter	310
Creating a Middleware Component and an Endpoint	311
Configuring the Request Pipeline	311
Understanding Service Location and Tight Coupling	313
Understanding the Service Location Problem	313
Understanding the Tightly Coupled Components Problem	315
Using Dependency Injection	317
Using a Service in a Middleware Class.....	318
Using a Service in an Endpoint.....	320
Using Service Lifecycles	324
Creating Transient Services	325
Avoiding the Transient Service Reuse Pitfall	325
Using Scoped Services.....	329

Other Dependency Injection Features	334
Creating Dependency Chains.....	334
Accessing Services in the ConfigureServices Method	336
Using Service Factory Functions	337
Creating Services with Multiple Implementations.....	338
Using Unbound Types in Services.....	341
Summary	342
■ Chapter 15: Using the Platform Features, Part 1	343
Preparing for This Chapter	344
Using the Configuration Service	345
Understanding the Environment-Specific Configuration File.....	346
Accessing Configuration Settings.....	347
Using the Configuration Data in Services	348
Understanding the Launch Settings File.....	351
Determining the Environment in the Startup Class	356
Storing User Secrets.....	357
Using the Logging Service.....	361
Generating Logging Messages	361
Configuring Minimum Logging Levels	364
Using Static Content and Client-Side Packages	365
Adding the Static Content Middleware	366
Using Client-Side Packages.....	369
Summary	372
■ Chapter 16: Using the Platform Features, Part 2	373
Preparing for This Chapter	373
Using Cookies.....	374
Enabling Cookie Consent Checking	376
Managing Cookie Consent.....	378
Using Sessions	380
Configuring the Session Service and Middleware	380
Using Session Data.....	383
Working with HTTPS Connections	384
Enabling HTTP Connections.....	384
Detecting HTTPS Requests	386

Enforcing HTTPS Requests	387
Enabling HTTP Strict Transport Security	389
Handling Exceptions and Errors	391
Returning an HTML Error Response	393
Enriching Status Code Responses	395
Filtering Requests Using the Host Header	397
Summary	399
■ Chapter 17: Working with Data	401
Preparing for This Chapter	402
Caching Data	404
Caching Data Values	406
Using a Shared and Persistent Data Cache	409
Caching Responses	412
Using Entity Framework Core	415
Installing Entity Framework Core	415
Creating the Data Model	416
Configuring the Database Service	417
Creating and Applying the Database Migration	418
Seeding the Database	419
Using Data in an Endpoint	422
Summary	424
■ Part III: ASP.NET Core Applications	425
■ Chapter 18: Creating the Example Project	427
Creating the Project	427
Adding a Data Model	428
Adding NuGet Packages to the Project	428
Creating the Data Model	428
Preparing the Seed Data	430
Configuring Entity Framework Core Services and Middleware	431
Creating and Applying the Migration	432
Adding the CSS Framework	433

Configuring the Request Pipeline	433
Running the Example Application.....	435
Summary	435
■ Chapter 19: Creating RESTful Web Services	437
Preparing for This Chapter	438
Understanding RESTful Web Services	438
Understanding Request URLs and Methods	438
Understanding JSON.....	439
Creating a Web Service Using a Custom Endpoint	439
Creating a Web Service Using a Controller	442
Enabling the MVC Framework	443
Creating a Controller.....	444
Improving the Web Service	453
Using Asynchronous Actions.....	453
Preventing Over-Binding.....	455
Using Action Results	456
Validating Data	461
Applying the API Controller Attribute	463
Omitting Null Properties	464
Summary	467
■ Chapter 20: Advanced Web Service Features	469
Preparing for This Chapter	469
Dropping the Database.....	470
Running the Example Application	470
Dealing with Related Data.....	471
Breaking Circular References in Related Data	473
Supporting the HTTP PATCH Method	474
Understanding JSON Patch.....	474
Installing and Configuring the JSON Patch Package	475
Defining the Action Method	475
Understanding Content Formatting	477
Understanding the Default Content Policy	477
Understanding Content Negotiation.....	478
Specifying an Action Result Format.....	482

Requesting a Format in the URL.....	483
Restricting the Formats Received by an Action Method.....	484
Documenting and Exploring Web Services.....	486
Resolving Action Conflicts.....	486
Installing and Configuring the Swashbuckle Package.....	487
Fine-Tuning the API Description.....	490
Summary.....	493
■ Chapter 21: Using Controllers with Views, Part I.....	495
Preparing for This Chapter.....	496
Dropping the Database.....	497
Running the Example Application.....	497
Getting Started with Views.....	497
Configuring the Application.....	498
Creating an HTML Controller.....	499
Creating a Razor View.....	501
Selecting a View by Name.....	504
Working with Razor Views.....	507
Setting the View Model Type.....	510
Understanding the Razor Syntax.....	515
Understanding Directives.....	515
Understanding Content Expressions.....	516
Setting Element Content.....	516
Setting Attribute Values.....	518
Using Conditional Expressions.....	518
Enumerating Sequences.....	522
Using Razor Code Blocks.....	524
Summary.....	525
■ Chapter 22: Using Controllers with Views, Part II.....	527
Preparing for This Chapter.....	527
Dropping the Database.....	529
Running the Example Application.....	529
Using the View Bag.....	529
Using Temp Data.....	531

Working with Layouts	534
Configuring Layouts Using the View Bag.....	535
Using a View Start File.....	537
Overriding the Default Layout.....	538
Using Layout Sections	542
Using Partial Views	548
Enabling Partial Views	548
Creating a Partial View	548
Applying a Partial View	549
Understanding Content-Encoding	552
Understanding HTML Encoding	552
Understanding JSON Encoding.....	554
Summary	555
■ Chapter 23: Using Razor Pages	557
Preparing for This Chapter	558
Running the Example Application	558
Understanding Razor Pages	559
Configuring Razor Pages	559
Creating a Razor Page	560
Understanding Razor Pages Routing	564
Specifying a Routing Pattern in a Razor Page.....	566
Adding Routes for a Razor Page.....	568
Understanding the Page Model Class	569
Using a Code-Behind Class File.....	570
Understanding Action Results in Razor Pages.....	572
Handling Multiple HTTP Methods	575
Selecting a Handler Method	578
Understanding the Razor Page View	580
Creating a Layout for Razor Pages	580
Using Partial Views in Razor Pages	582
Creating Razor Pages Without Page Models.....	583
Summary	584

■ Chapter 24: Using View Components	585
Preparing for This Chapter	585
Dropping the Database	588
Running the Example Application	588
Understanding View Components	588
Creating and Using a View Component	589
Applying a View Component	589
Understanding View Component Results	593
Returning a Partial View	593
Returning HTML Fragments	596
Getting Context Data	598
Providing Context from the Parent View Using Arguments	600
Creating Asynchronous View Components	603
Creating View Components Classes	604
Creating a Hybrid Controller Class	607
Summary	609
■ Chapter 25: Using Tag Helpers	611
Preparing for This Chapter	612
Dropping the Database	613
Running the Example Application	614
Creating a Tag Helper	614
Defining the Tag Helper Class	615
Registering Tag Helpers	617
Using a Tag Helper	618
Narrowing the Scope of a Tag Helper	619
Widening the Scope of a Tag Helper	620
Advanced Tag Helper Features	622
Creating Shorthand Elements	622
Creating Elements Programmatically	625
Prepending and Appending Content and Elements	625
Getting View Context Data	629
Working with Model Expressions	631
Coordinating Between Tag Helpers	635
Suppressing the Output Element	637

Using Tag Helper Components	639
Creating a Tag Helper Component	639
Expanding Tag Helper Component Element Selection	641
Summary	642
■ Chapter 26: Using the Built-in Tag Helpers	643
Preparing for This Chapter	643
Adding an Image File	645
Installing a Client-Side Package	646
Dropping the Database	646
Running the Example Application	646
Enabling the Built-in Tag Helpers	647
Transforming Anchor Elements	647
Using Anchor Elements for Razor Pages	649
Using the JavaScript and CSS Tag Helpers	650
Managing JavaScript Files	650
Managing CSS Stylesheets	657
Working with Image Elements	660
Using the Data Cache	661
Setting Cache Expiry	663
Using the Hosting Environment Tag Helper	666
Summary	667
■ Chapter 27: Using the Forms Tag Helpers	669
Preparing for This Chapter	669
Dropping the Database	671
Running the Example Application	671
Understanding the Form Handling Pattern	672
Creating a Controller to Handle Forms	672
Creating a Razor Page to Handle Forms	674
Using Tag Helpers to Improve HTML Forms	676
Working with Form Elements	676
Transforming Form Buttons	678
Working with input Elements	679
Transforming the input Element type Attribute	680
Formatting input Element Values	682

Displaying Values from Related Data in input Elements	685
Working with Label Elements.....	688
Working with Select and Option Elements	690
Populating a select Element.....	692
Working with Text Areas	694
Using the Anti-forgery Feature	695
Enabling the Anti-forgery Feature in a Controller.....	696
Enabling the Anti-forgery Feature in a Razor Page.....	697
Using Anti-forgery Tokens with JavaScript Clients.....	699
Summary.....	701
■ Chapter 28: Using Model Binding	703
Preparing for This Chapter	704
Dropping the Database.....	705
Running the Example Application	705
Understanding Model Binding	705
Binding Simple Data Types.....	707
Binding Simple Data Types in Razor Pages	708
Understanding Default Binding Values	710
Binding Complex Types	712
Binding to a Property.....	713
Binding Nested Complex Types.....	715
Selectively Binding Properties.....	719
Binding to Arrays and Collections.....	722
Binding to Arrays	722
Binding to Simple Collections.....	725
Binding to Dictionaries	726
Binding to Collections of Complex Types.....	728
Specifying a Model Binding Source.....	730
Selecting a Binding Source for a Property.....	733
Using Headers for Model Binding	733
Using Request Bodies as Binding Sources.....	735
Manually Model Binding.....	736
Summary.....	738

■ Chapter 29: Using Model Validation	739
Preparing for This Chapter	740
Dropping the Database	741
Running the Example Application	741
Understanding the Need for Model Validation	742
Explicitly Validating Data in a Controller	742
Displaying Validation Errors to the User	745
Displaying Validation Messages	747
Displaying Property-Level Validation Messages	751
Displaying Model-Level Messages	752
Explicitly Validating Data in a Razor Page	754
Specifying Validation Rules Using Metadata	757
Creating a Custom Property Validation Attribute	761
Performing Client-Side Validation	765
Performing Remote Validation	767
Performing Remote Validation in Razor Pages	770
Summary	771
■ Chapter 30: Using Filters	773
Preparing for This Chapter	773
Enabling HTTPS Connections	775
Dropping the Database	776
Running the Example Application	776
Using Filters	777
Using Filters in Razor Pages	780
Understanding Filters	782
Creating Custom Filters	783
Understanding Authorization Filters	783
Understanding Resource Filters	786
Understanding Action Filters	789
Understanding Page Filters	793
Understanding Result Filters	797
Understanding Exception Filters	802
Creating an Exception Filter	802

Managing the Filter Lifecycle	804
Creating Filter Factories	806
Using Dependency Injection Scopes to Manage Filter Lifecycles.....	808
Creating Global Filters	810
Understanding and Changing Filter Order	811
Changing Filter Order	814
Summary	815
■ Chapter 31: Creating Form Applications	817
Preparing for This Chapter	817
Dropping the Database	820
Running the Example Application	820
Creating an MVC Forms Application	821
Preparing the View Model and the View	821
Reading Data	822
Creating Data.....	824
Editing Data	828
Deleting Data	830
Creating a Razor Pages Forms Application	832
Creating Common Functionality	834
Defining Pages for the CRUD Operations.....	836
Creating New Related Data Objects	840
Providing the Related Data in the Same Request.....	840
Breaking Out to Create New Data.....	843
Summary.....	847
■ Part IV: Advanced ASP.NET Core Features	849
■ Chapter 32: Creating the Example Project.....	851
Creating the Project.....	851
Adding NuGet Packages to the Project.....	852
Adding a Data Model	853
Preparing the Seed Data.....	854
Configuring Entity Framework Core Services and Middleware	856
Creating and Applying the Migration	857
Adding the Bootstrap CSS Framework.....	857

Configuring the Services and Middleware	858
Creating a Controller and View	859
Creating a Razor Page	861
Running the Example Application.....	863
Summary	864
■ Chapter 33: Using Blazor Server, Part 1	865
Preparing for This Chapter	866
Understanding Blazor Server	867
Understanding the Blazor Server Advantages	867
Understanding the Blazor Server Disadvantages	868
Choosing Between Blazor Server and Angular/React/Vue.js	868
Getting Started with Blazor	868
Configuring ASP.NET Core for Blazor Server	868
Creating a Razor Component.....	870
Understanding the Basic Razor Component Features.....	875
Understanding Blazor Events and Data Bindings	875
Working with Data Bindings	883
Using Class Files to Define Components	888
Using a Code-Behind Class.....	888
Defining a Razor Component Class	890
Summary	892
■ Chapter 34: Using Blazor Server, Part 2	893
Preparing for This Chapter	893
Combining Components	894
Configuring Components with Attributes.....	896
Creating Custom Events and Bindings.....	901
Displaying Child Content in a Component	905
Creating Template Components.....	907
Using Generic Type Parameters in Template Components	909
Cascading Parameters.....	915
Handling Errors	918
Handling Connection Errors.....	918
Handling Uncaught Application Errors.....	920
Summary	922

■ Chapter 35: Advanced Blazor Features	923
Preparing for This Chapter	924
Using Component Routing	924
Preparing the Razor Page	925
Adding Routes to Components	926
Navigating Between Routed Components	929
Receiving Routing Data	932
Defining Common Content Using Layouts	933
Understanding the Component Lifecycle Methods	935
Using the Lifecycle Methods for Asynchronous Tasks	938
Managing Component Interaction	939
Using References to Child Components	939
Interacting with Components from Other Code	942
Interacting with Components Using JavaScript	946
Summary	954
■ Chapter 36: Blazor Forms and Data	955
Preparing for This Chapter	955
Dropping the Database and Running the Application	958
Using the Blazor Form Components	959
Creating Custom Form Components	961
Validating Form Data	964
Handling Form Events	967
Using Entity Framework Core with Blazor	969
Understanding the Entity Framework Core Context Scope Issue	969
Understanding the Repeated Query Issue	973
Performing Create, Read, Update, and Delete Operations	978
Creating the List Component	978
Creating the Details Component	979
Creating the Editor Component	980
Extending the Blazor Form Features	983
Creating a Custom Validation Constraint	983
Creating a Valid-Only Submit Button Component	986
Summary	988

■ Chapter 37: Using Blazor Web Assembly	989
Preparing for This Chapter	990
Dropping the Database and Running the Application	991
Setting Up Blazor WebAssembly	992
Creating the Shared Project.....	992
Creating the Blazor WebAssembly Project.....	992
Preparing the ASP.NET Core Project.....	993
Adding the Solution References	993
Opening the Projects	994
Completing the Blazor WebAssembly Configuration.....	994
Testing the Placeholder Components	997
Creating a Blazor WebAssembly Component	997
Importing the Data Model Namespace	997
Creating a Component.....	998
Creating a Layout.....	1001
Defining CSS Styles	1002
Completing the Blazor WebAssembly Form Application	1003
Creating the Details Component.....	1003
Creating the Editor Component.....	1004
Summary	1006
■ Chapter 38: Using ASP.NET Core Identity	1007
Preparing for This Chapter	1008
Preparing the Project for ASP.NET Core Identity	1009
Preparing the ASP.NET Core Identity Database.....	1009
Configuring the Application	1010
Creating and Applying the Identity Database Migration	1011
Creating User Management Tools	1012
Preparing for User Management Tools	1013
Enumerating User Accounts	1013
Creating Users	1015
Editing Users	1023
Deleting Users	1026

Creating Role Management Tools.....	1027
Preparing for Role Management Tools.....	1027
Enumerating and Deleting Roles	1028
Creating Roles	1029
Assigning Role Membership	1030
Summary.....	1033
■ Chapter 39: Applying ASP.NET Core Identity.....	1035
Preparing for This Chapter	1035
Authenticating Users	1037
Creating the Login Feature	1037
Inspecting the ASP.NET Core Identity Cookie.....	1039
Creating a Sign-Out Page	1040
Testing the Authentication Feature.....	1041
Enabling the Identity Authentication Middleware.....	1041
Authorizing Access to Endpoints	1044
Applying the Authorization Attribute.....	1044
Enabling the Authorization Middleware.....	1045
Creating the Access Denied Endpoint.....	1045
Creating the Seed Data.....	1046
Testing the Authentication Sequence	1048
Authorizing Access to Blazor Applications	1049
Performing Authorization in Blazor Components.....	1051
Displaying Content to Authorized Users.....	1053
Authenticating and Authorizing Web Services.....	1054
Building a Simple JavaScript Client.....	1057
Restricting Access to the Web Service	1059
Using Cookie Authentication.....	1060
Using Bearer Token Authentication.....	1062
Creating Tokens	1063
Authenticating with Tokens	1066
Restricting Access with Tokens	1068
Using Tokens to Request Data	1068
Summary.....	1070
■ Index.....	1071

About the Author



Adam Freeman is an experienced IT professional who has held senior positions in a range of companies, most recently serving as chief technology officer and chief operating officer of a global bank. Now retired, he spends his time writing and long-distance running.

About the Technical Reviewer

Fabio Claudio Ferracchiati is a senior consultant and a senior analyst/developer using Microsoft technologies. He works for BluArancio (www.bluarancio.com). He is a Microsoft Certified Solution Developer for .NET, a Microsoft Certified Application Developer for .NET, a Microsoft Certified Professional, and a prolific author and technical reviewer. Over the past ten years, he's written articles for Italian and international magazines and coauthored more than ten books on a variety of computer topics.

PART I



Introducing ASP.NET Core

CHAPTER 1



Putting ASP.NET Core in Context

Understanding ASP.NET Core

ASP.NET Core is Microsoft's web development platform. The original ASP.NET was introduced in 2002, and it has been through several reinventions and reincarnations to become ASP.NET Core 3, which is the topic of this book.

ASP.NET Core consists of a platform for processing HTTP requests, a series of principal frameworks for creating applications, and secondary utility frameworks that provide supporting features, as illustrated by Figure 1-1.

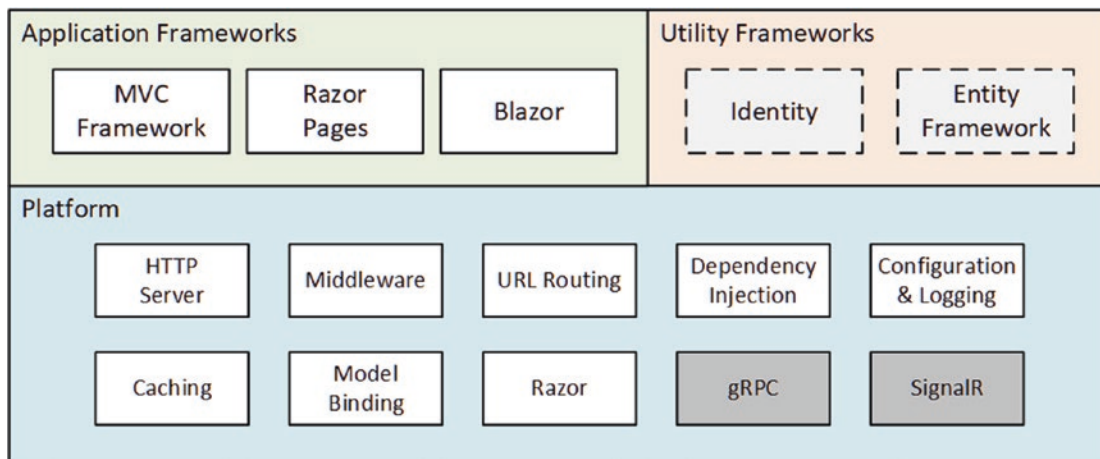


Figure 1-1. The structure of ASP.NET Core

Understanding the Application Frameworks

When you start using ASP.NET Core, it can be confusing to find that there are different application frameworks available. As you will learn, these frameworks are complementary and solve different problems, or, for some features, solve the same problems in different ways. Understanding the relationship between these frameworks means understanding the changing design patterns that Microsoft has supported, as I explain in the sections that follow.

Understanding the MVC Framework

The MVC Framework was introduced in the pre-Core days of ASP.NET. The original ASP.NET relied on a development model called Web Pages, which re-created the experience of writing desktop applications but resulted in unwieldy web projects that did not scale well. The MVC Framework was introduced alongside Web Pages with a development model that embraced the character of HTTP and HTML, rather than trying to hide it.

MVC stands for Model-View-Controller, which is a design pattern that describes the shape of an application. The MVC pattern emphasizes *separation of concerns*, where areas of functionality are defined independently, which was an effective antidote to the indistinct architectures that Web Pages led to.

Early versions of the MVC Framework were built on the ASP.NET foundations that were originally designed for Web Pages, which led to some awkward features and workarounds. With the move to .NET Core, ASP.NET became ASP.NET Core, and the MVC Framework was rebuilt on an open, extensible, and cross-platform foundation.

The MVC Framework remains an important part of ASP.NET Core, but the way it is commonly used has changed with the rise of single-page applications (SPAs). In an SPA, the browser makes a single HTTP request and receives an HTML document that delivers a rich client, typically written in a JavaScript client such as Angular or React. The shift to SPAs means that the clean separation that the MVC Framework was originally intended for is not as important, and the emphasis placed on following the MVC pattern is no longer essential, even though the MVC Framework remains useful (and is used to support SPAs through web services, as described in Chapter 19).

PUTTING PATTERNS IN THEIR PLACE

Design patterns provoke strong reactions, as the emails I receive from readers will testify. A substantial proportion of the messages I receive are complaints that I have not applied a pattern correctly.

Patterns are just other people's solutions to the problems they encountered in other projects. If you find yourself facing the same problem, understanding how it has been solved before can be helpful. But that doesn't mean you have to follow the pattern exactly, or at all, as long as you understand the consequences. If a pattern is intended to make projects manageable, for example, and you choose to deviate from that pattern, then you must accept that your project may be more difficult to manage. But a pattern followed slavishly can be worse than no pattern at all, and no pattern is suited to every project.

My advice is to use patterns freely, adapt them as necessary, and ignore zealots who confuse patterns with commandments.

Understanding Razor Pages

One drawback of the MVC Framework is that it can require a lot of preparatory work before an application can start producing content. Despite its structural problems, one advantage of Web Pages was that simple applications could be created in a couple of hours.

Razor Pages takes the development ethos of Web Pages and implements it using the platform features originally developed for the MVC Framework. Code and content are mixed to form self-contained pages; this re-creates the speed of Web Pages development without some of the underlying technical problems (although the issue of scaling up complex projects can still be an issue).

Razor Pages can be used alongside the MVC Framework, which is how I tend to use them. I write the main parts of the application using the MVC Framework and use Razor Pages for the secondary features, such as administration and reporting tools. You can see this approach in Chapters 7–11, where I develop a realistic ASP.NET Core application called SportsStore.

Understanding Blazor

The rise of JavaScript client-side frameworks can be a barrier for C# developers, who must learn a different—and somewhat idiosyncratic—programming language. I have come to love JavaScript, which is as fluid and expressive as C#. But it takes time and commitment to become proficient in a new programming language, especially one that has fundamental differences from C#.

Blazor attempts to bridge this gap by allowing C# to be used to write client-side applications. There are two versions of Blazor: Blazor Server and Blazor WebAssembly. Blazor Server is a stable and supported part of ASP.NET Core, and it works by using a persistent HTTP connection to the ASP.NET Core server, where the application's C# code is executed. Blazor WebAssembly is an experimental release that goes one step further and executes the application's C# code in the browser. Neither version of Blazor is suited for all situations, as I explain in Chapter 33, but they both give a sense of direction for the future of ASP.NET Core development.

Understanding the Utility Frameworks

Two frameworks are closely associated with ASP.NET Core but are not used directly to generate HTML content or data. Entity Framework Core is Microsoft's object-relational mapping (ORM) framework, which represents data stored in a relational database as .NET objects. Entity Framework Core can be used in any .NET Core application, and it is commonly used to access databases in ASP.NET Core applications.

ASP.NET Core Identity is Microsoft’s authentication and authorization framework, and it is used to validate user credentials in ASP.NET Core applications and restrict access to application features.

I describe only the basic features of both frameworks in this book, focusing on the capabilities required by most ASP.NET Core applications. But these are both complex frameworks that are too large to describe in detail in what is already a large book about ASP.NET Core.

TOPICS FOR FUTURE EDITIONS

I don’t have space in this book to cover every Entity Framework Core and ASP.NET Core Identity feature, so I have focused on those aspects that most projects require. If there are topics you think I should include in the next edition or in new deep-dive books, then please send me your suggestions at adam@adam-freeman.com.

Understanding the ASP.NET Core Platform

The ASP.NET Core platform contains the low-level features required to receive and process HTTP requests and create responses. There is an integrated HTTP server, a system of middleware components to handle requests, and core features that the application frameworks depend on, such as URL routing and the Razor view engine.

Most of your development time will be spent with the application frameworks, but effective ASP.NET Core use requires an understanding of the powerful capabilities that the platform provides, without which the higher-level frameworks could not function. I demonstrate how the ASP.NET Core platform works in detail in Part 2 of this book and explain how the features it provides underpin every aspect of ASP.NET Core development.

I have not described two notable platform features in this book: SignalR and gRPC. SignalR is used to create low-latency communication channels between applications. It provides the foundation for the Blazor Server framework that I describe in Part 4 of this book, but SignalR is rarely used directly, and there are better alternatives for those few projects that need low-latency messaging, such as Azure Event Grid or Azure Service Bus.

gRPC is an emerging standard for cross-platform remote procedure calls (RPCs) over HTTP that was originally created by Google (the *g* in gRPC) and offers efficiency and scalability benefits. gRPC may be the future standard for web services, but it cannot be used in web applications because it requires low-level control of the HTTP messages that it sends, which browsers do not allow. (There is a browser library that allows gRPC to be used via a proxy server, but that undermines the benefits of using gRPC.) Until gRPC can be used in the browser, its inclusion in ASP.NET Core is of interest only for projects that use it for communication between back-end servers, for which many alternative protocols exist. I may cover gRPC in future editions of this book but not until it can be used in the browser or becomes the dominant data-center protocol.

Understanding This Book

To get the most from this book, you should be familiar with the basics of web development, understand how HTML and CSS work, and have a working knowledge of C#. Don’t worry if you haven’t done any client-side development, such as JavaScript. The emphasis in this book is on C# and ASP.NET Core, and you will be able to pick up everything you need to know as you progress through the chapters. In Chapter 5, I summarize the most important C# features for ASP.NET Core development, which you will find useful if you are coming to ASP.NET Core from earlier versions of .NET Core or the .NET Framework.

What Software Do I Need to Follow the Examples?

You need a code editor (either Visual Studio or Visual Studio Code), the .NET Core Software Development Kit, and SQL Server LocalDB. All are available for use from Microsoft without charge, and Chapter 2 contains instructions for installing everything you need.

What Platform Do I Need to Follow the Examples?

This book is written for Windows. I used Windows 10 Pro, but any version of Windows supported by Visual Studio, Visual Studio Code, and .NET Core should work. ASP.NET Core is supported on other platforms, but the examples in this book rely on the SQL Server LocalDB feature, which is specific to Windows. You can contact me at adam@adam-freeman.com if you are trying to use another platform, and I will give you some general pointers for adapting the examples, albeit with the caveat that I won’t be able to provide detailed help if you get stuck.

What If I Have Problems Following the Examples?

The first thing to do is to go back to the start of the chapter and begin again. Most problems are caused by missing a step or not fully following a listing. Pay close attention to the emphasis in code listings, which highlights the changes that are required.

Next, check the errata/corrections list, which is included in the book's GitHub repository. Technical books are complex, and mistakes are inevitable, despite my best efforts and those of my editors. Check the errata list for the list of known errors and instructions to resolve them.

If you still have problems, then download the project for the chapter you are reading from the book's GitHub repository, <https://github.com/apress/pro-asp.net-core-3>, and compare it to your project. I create the code for the GitHub repository by working through each chapter, so you should have the same files with the same contents in your project.

If you still can't get the examples working, then you can contact me at adam@adam-freeman.com for help. Please make it clear in your email which book you are reading and which chapter/example is causing the problem. Please remember that I get a lot of emails and that I may not respond immediately.

What If I Find an Error in the Book?

You can report errors to me by email at adam@adam-freeman.com, although I ask that you first check the errata/corrections list for this book, which you can find in the book's GitHub repository at <https://github.com/apress/pro-asp.net-core-3>, in case it has already been reported.

I add errors that are likely to cause confusion to readers, especially problems with example code, to the errata/corrections file on the GitHub repository, with a grateful acknowledgment to the first reader who reported them. I keep a list of less serious issues, which usually means errors in the text surrounding examples, and I fix them when I write a new edition.

What Does This Book Cover?

I have tried to cover the features that will be required by most ASP.NET Core projects. This book is split into four parts, each of which covers a set of related topics.

Part 1: Introducing ASP.NET Core

This part of the book—which includes this chapter—introduces ASP.NET Core. In addition to setting up your development environment and creating your first application, you'll learn about the most important C# features for ASP.NET Core development and how to use the ASP.NET Core development tools. But most of Part 1 is given over to the development of a project called SportsStore, through which I show you a realistic development process from inception to deployment, touching on all the main features of ASP.NET Core and showing how they fit together—something that can be lost in the deep-dive chapters in the rest of the book.

Part 2: The ASP.NET Core Platform

The chapters in this part of the book describe the key features of the ASP.NET Core platform. I explain how HTTP requests are processed, how to create and use middleware components, how to create routes, how to define and consume services, and how to work with Entity Framework Core. These chapters explain the foundations of ASP.NET Core, and understanding them is essential for effective ASP.NET Core development.

Part 3: ASP.NET Core Applications

The chapters in this part of the book explain how to create different types of applications, including RESTful web services and HTML applications using controllers and Razor Pages. These chapters also describe the features that make it easy to generate HTML, including the views, view components, and tag helpers.

Part 4: Advanced ASP.NET Core Features

The final part of the book explains how to create applications using Blazor Server, how to use the experimental Blazor WebAssembly, and how to authenticate users and authorize access using ASP.NET Core Identity.

What Doesn't This Book Cover?

This book doesn't cover basic web development topics, such as HTML and CSS, and doesn't teach basic C# (although Chapter 5 does describe C# features useful for ASP.NET Core development that may not be familiar to developers using older versions of .NET).

As much as I like to dive into the details in my books, not every ASP.NET Core feature is useful in mainstream development, and I have to keep my books to a printable size. When I decide to omit a feature, it is because I don't think it is important or because the same outcome can be achieved using a technique that I do cover.

As noted earlier, I have not described the ASP.NET Core support for SignalR and gRPC, and I note other features in later chapters that I don't describe, either because they are not broadly applicable or because there are better alternatives available. In each case, I explain why I have omitted a description and provide a reference to the Microsoft documentation for that topic.

How Do I Contact the Author?

You can email me at adam@adam-freeman.com. It has been a few years since I first published an email address in my books. I wasn't entirely sure that it was a good idea, but I am glad that I did it. I have received emails from around the world, from readers working or studying in every industry, and—for the most part anyway—the emails are positive, polite, and a pleasure to receive.

I try to reply promptly, but I get a lot of email, and sometimes I get a backlog, especially when I have my head down trying to finish writing a book. I always try to help readers who are stuck with an example in the book, although I ask that you follow the steps described earlier in this chapter before contacting me.

While I welcome reader emails, there are some common questions for which the answers will always be no. I am afraid that I won't write the code for your new startup, help you with your college assignment, get involved in your development team's design dispute, or teach you how to program.

What If I Really Enjoyed This Book?

Please email me at adam@adam-freeman.com and let me know. It is always a delight to hear from a happy reader, and I appreciate the time it takes to send those emails. Writing these books can be difficult, and those emails provide essential motivation to persist at an activity that can sometimes feel impossible.

What If This Book Has Made Me Angry and I Want to Complain?

You can still email me at adam@adam-freeman.com, and I will still try to help you. Bear in mind that I can only help if you explain what the problem is and what you would like me to do about it. You should understand that sometimes the only outcome is to accept I am not the writer for you and that we will have closure only when you return this book and select another. I'll give careful thought to whatever has upset you, but after 25 years of writing books, I have come to understand that not everyone enjoys reading the books I like to write.

Summary

In this chapter, I set the scene for the rest of the book. I provided a brief overview of ASP.NET Core, explained the requirements for and the content of this book, and explained how you can contact me. In the next chapter, I show you how to prepare for ASP.NET Core development.

CHAPTER 2



Getting Started

The best way to appreciate a software development framework is to jump right in and use it. In this chapter, I explain how to prepare for ASP.NET Core development and how to create and run an ASP.NET Core application.

UPDATES TO THIS BOOK

Microsoft has an active development schedule for .NET Core and ASP.NET Core, which means that there may be new releases available by the time you read this book. It doesn't seem fair to expect readers to buy a new book every few months, especially since most changes are relatively minor. Instead, I will post free updates to the GitHub repository for this book (<https://github.com/apress/pro-asp.net-core-3>) for breaking changes.

This kind of update is an ongoing experiment for me (and for Apress), and it continues to evolve—not least because I don't know what the future major releases of ASP.NET Core will contain—but the goal is to extend the life of this book by supplementing the examples it contains.

I am not making any promises about what the updates will be like, what form they will take, or how long I will produce them before folding them into a new edition of this book. Please keep an open mind and check the repository for this book when new ASP.NET Core versions are released. If you have ideas about how the updates could be improved, then email me at adam@adam-freeman.com and let me know.

Choosing a Code Editor

Microsoft provides a choice of tools for ASP.NET Core development: Visual Studio and Visual Studio Code. Visual Studio is the traditional development environment for .NET applications, and it offers an enormous range of tools and features for developing all sorts of applications. But it can be resource-hungry and slow, and some of the features are so determined to be helpful they get in the way of development.

Visual Studio Code is a light-weight alternative that doesn't have the bells and whistles of Visual Studio but is perfectly capable of handling ASP.NET Core development.

All the examples in this book include instructions for both editors, and both Visual Studio and Visual Studio Code can be used without charge, so you can use whichever suits your development style.

If you are new to .NET Core development, then start with Visual Studio. It provides more structured support for creating the different types of files used in ASP.NET Core development, which will help ensure you get the expected results from the code examples.

■ **Note** This book describes ASP.NET Core development for Windows. It is possible to develop and run ASP.NET Core applications on Linux and macOS, but most readers use Windows, and that is what I have chosen to focus on. Almost all the examples in this book rely on LocalDB, which is a Windows-only feature provided by SQL Server that is not available on other platforms. If you want to follow this book on another platform, then you can contact me using the email address in Chapter 1, and I will try to help you get started.

Installing Visual Studio

ASP.NET Core 3 requires Visual Studio 2019. I use the free Visual Studio 2019 Community Edition, which can be downloaded from www.visualstudio.com. Run the installer, and you will see the prompt shown in Figure 2-1.

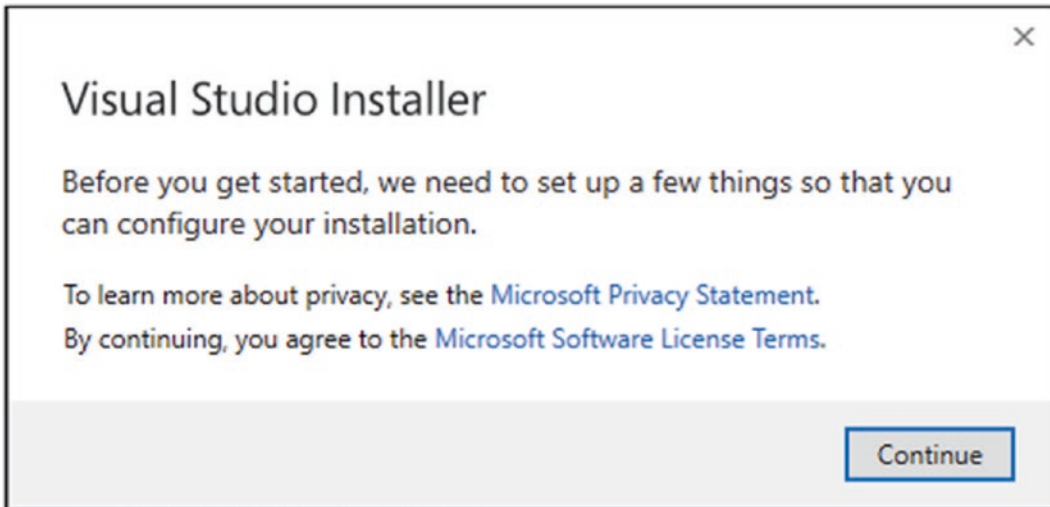


Figure 2-1. Starting the Visual Studio installer

Click the Continue button, and the installer will download the installation files, as shown in Figure 2-2.

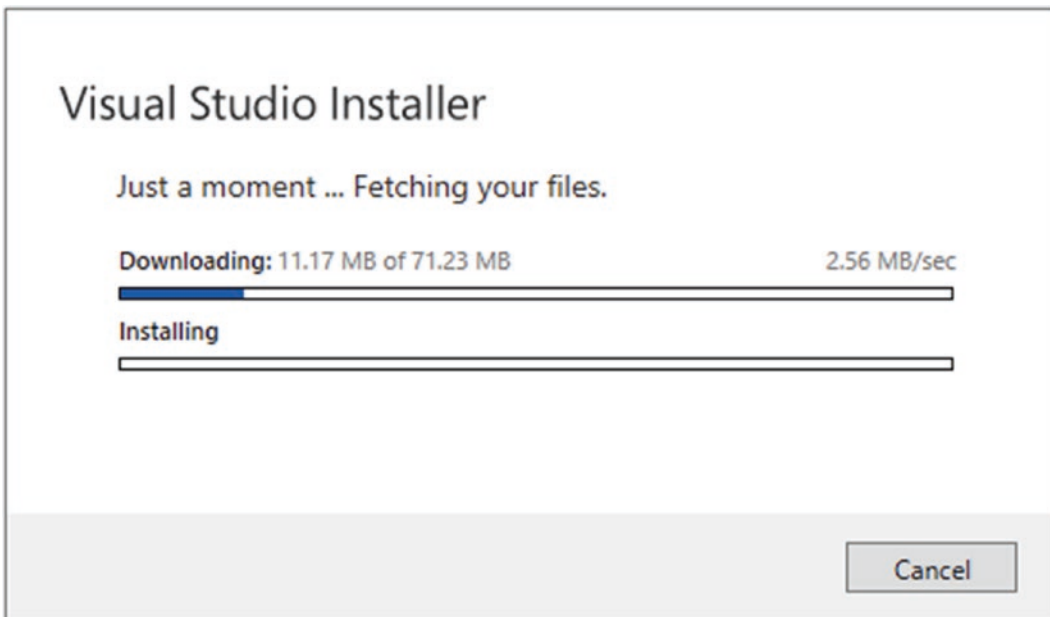


Figure 2-2. Downloading the Visual Studio installer files

When the installer files have been downloaded, you will be presented with a set of installation options, grouped into workloads. Ensure that the “ASP.NET and web development” workload is checked, as shown in Figure 2-3.

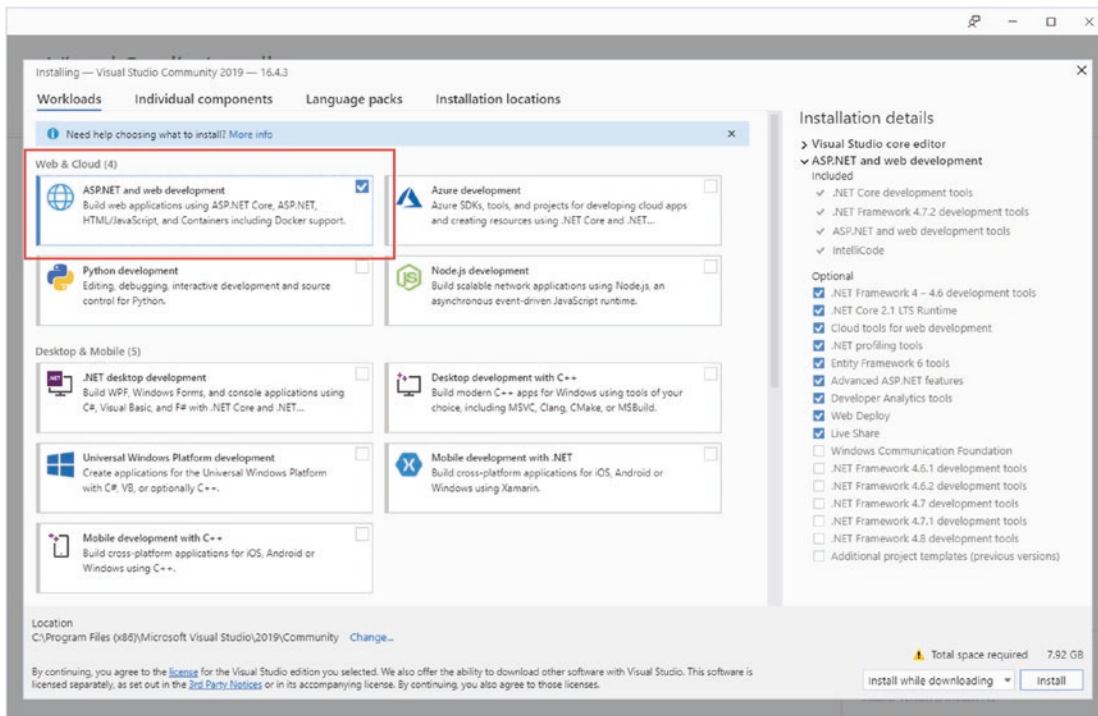


Figure 2-3. Selecting the workload

Select the “Individual components” section at the top of the window and ensure the SQL Server Express 2016 LocalDB option is checked, as shown in Figure 2-4. This is the database component that I will be using to store data in later chapters.

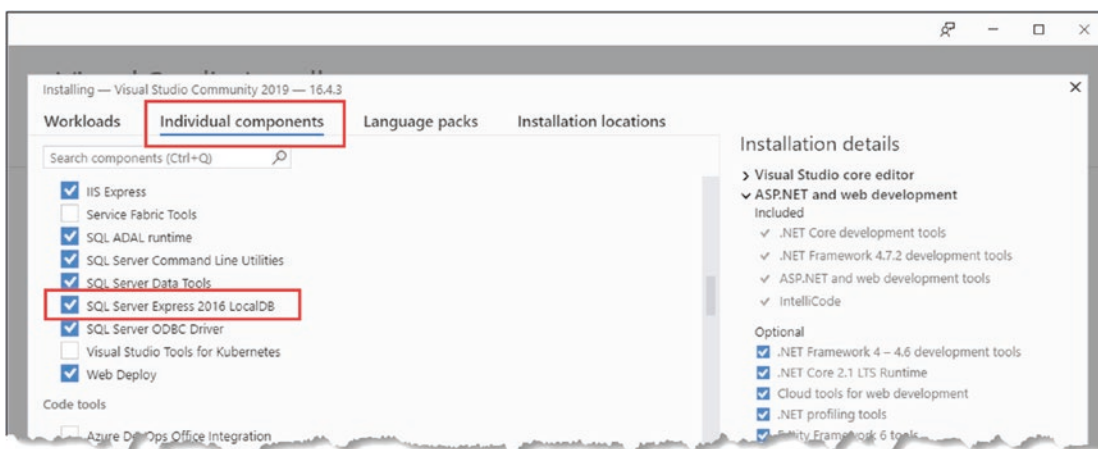


Figure 2-4. Ensuring LocalDB is installed

Click the Install button, and the files required for the selected workload will be downloaded and installed. To complete the installation, a reboot is required, as shown in Figure 2-5.

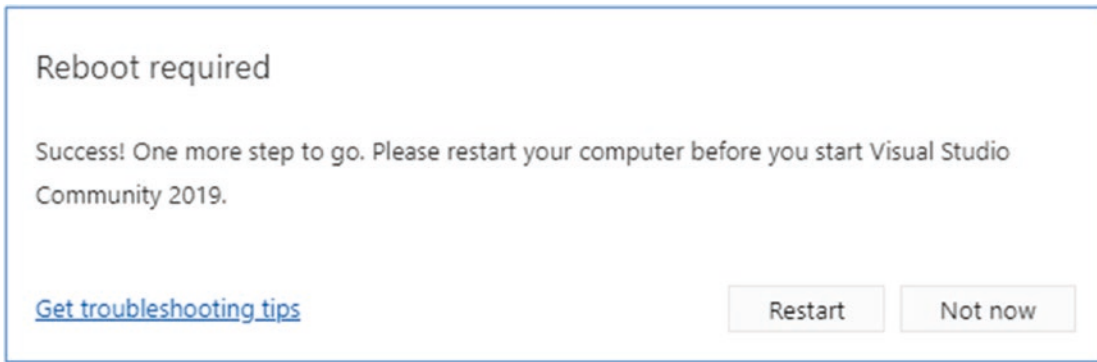


Figure 2-5. Completing the installation

Installing the .NET Core SDK

The Visual Studio installer will install the .NET Core Software Development Kit (SDK), but it may not install the version required for the examples in this book. Go to <https://dotnet.microsoft.com/download/dotnet-core/3.1> and download the installer for version 3.1.1 of the .NET Core SDK, which is the long-term support release at the time of writing. Run the installer; once the installation is complete, open a new PowerShell command prompt from the Windows Start menu and run the command shown in Listing 2-1, which displays a list of the installed .NET Core SDKs.

Listing 2-1. Listing the Installed SDKs

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

Here is the output from a fresh installation on a Windows machine that has not been used for .NET Core:

```
3.1.101 [C:\Program Files\dotnet\sdk]
```

If you have been working with different versions of .NET Core, you may see a longer list, like this one:

```
2.1.401 [C:\Program Files\dotnet\sdk]
2.1.502 [C:\Program Files\dotnet\sdk]
2.1.505 [C:\Program Files\dotnet\sdk]
2.1.602 [C:\Program Files\dotnet\sdk]
2.1.802 [C:\Program Files\dotnet\sdk]
3.0.100 [C:\Program Files\dotnet\sdk]
3.1.100 [C:\Program Files\dotnet\sdk]
3.1.101 [C:\Program Files\dotnet\sdk]
```

Regardless of how many entries there are, you must ensure there is one for the 3.1.1xx version, where the last two digits may differ.

Installing Visual Studio Code

If you have chosen to use Visual Studio Code, download the installer from <https://code.visualstudio.com>. No specific version is required, and you should select the current stable build. Run the installer and ensure you check the Add to PATH option, as shown in Figure 2-6.

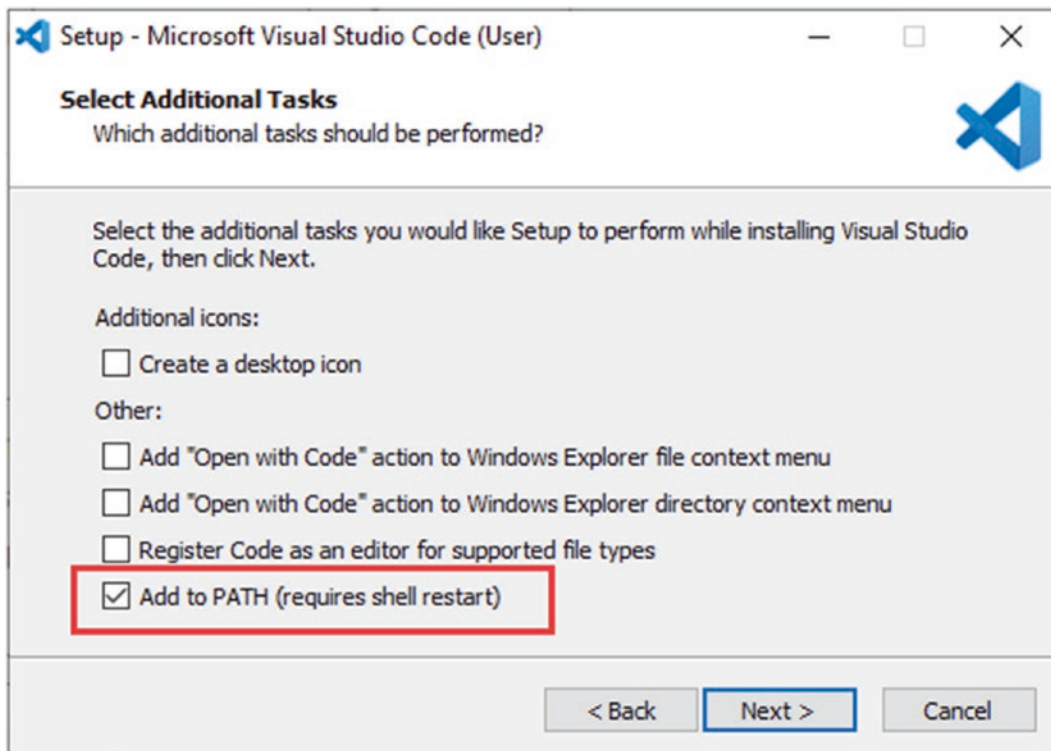


Figure 2-6. Configuring the Visual Studio Code installation

Installing the .NET Core SDK

The Visual Studio installer does not include the .NET Core SDK, which must be installed separately. Go to <https://dotnet.microsoft.com/download/dotnet-core/3.1> and download the installer for version 3.1.1 of the .NET Core SDK, which is the long-term support release at the time of writing. Run the installer; once the installation is complete, open a new PowerShell command prompt from the Windows Start menu and run the command shown in Listing 2-2, which displays a list of the installed .NET Core SDKs.

Listing 2-2. Listing the Installed SDKs

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

Here is the output from a fresh installation on a Windows machine that has not been used for .NET Core:

```
3.1.101 [C:\Program Files\dotnet\sdk]
```

If you have been working with different versions of .NET Core, you may see a longer list, like this one:

```
2.1.401 [C:\Program Files\dotnet\sdk]
2.1.502 [C:\Program Files\dotnet\sdk]
2.1.505 [C:\Program Files\dotnet\sdk]
2.1.602 [C:\Program Files\dotnet\sdk]
2.1.802 [C:\Program Files\dotnet\sdk]
3.0.100 [C:\Program Files\dotnet\sdk]
3.1.100 [C:\Program Files\dotnet\sdk]
3.1.101 [C:\Program Files\dotnet\sdk]
```

Regardless of how many entries there are, you must ensure there is one for the 3.1.1xx version, where the last two digits may differ.

Installing SQL Server LocalDB

The database examples in this book require LocalDB, which is a zero-configuration version of SQL Server that can be installed as part of the SQL Server Express edition, which is available for use without charge from <https://www.microsoft.com/en-in/sql-server/sql-server-downloads>. Download and run the Express edition installer and select the Custom option, as shown in Figure 2-7.



Figure 2-7. Selecting the installation option for SQL Server

Once you have selected the Custom option, you will be prompted to select a download location for the installation files. Click the Install button, and the download will begin.

When prompted, select the option to create a new SQL Server installation, as shown in Figure 2-8.

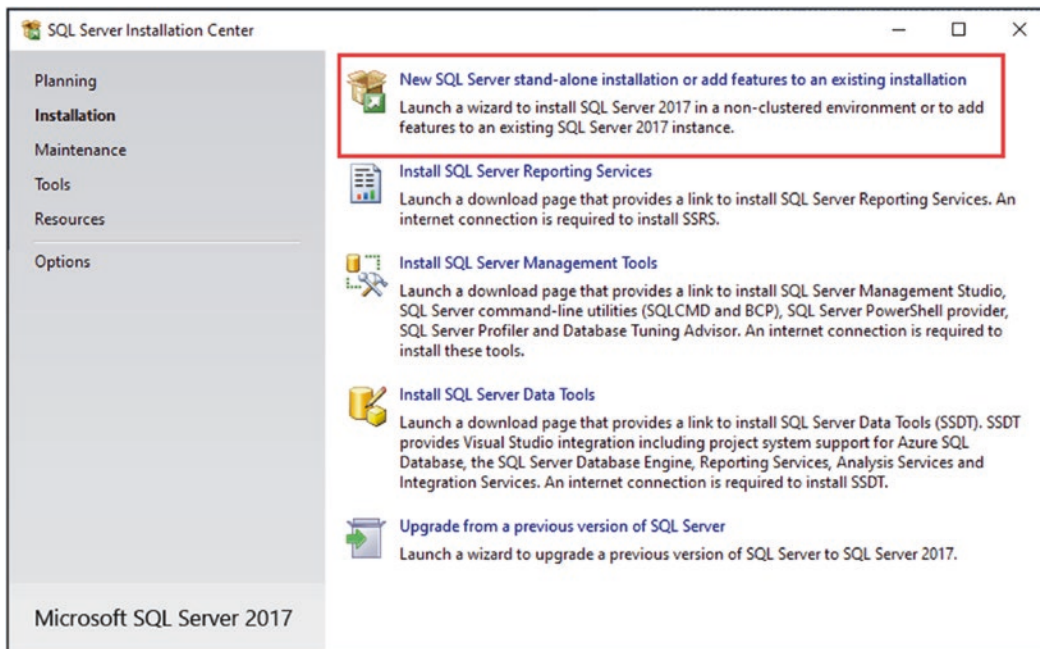


Figure 2-8. Selecting an installation option

Work through the installation process, selecting the default options as they are presented. When you reach the Feature Selection page, ensure that the LocalDB option is checked, as shown in Figure 2-9. (You may want to uncheck the options for R and Python, which are not used in this book and take a long time to download and install.)

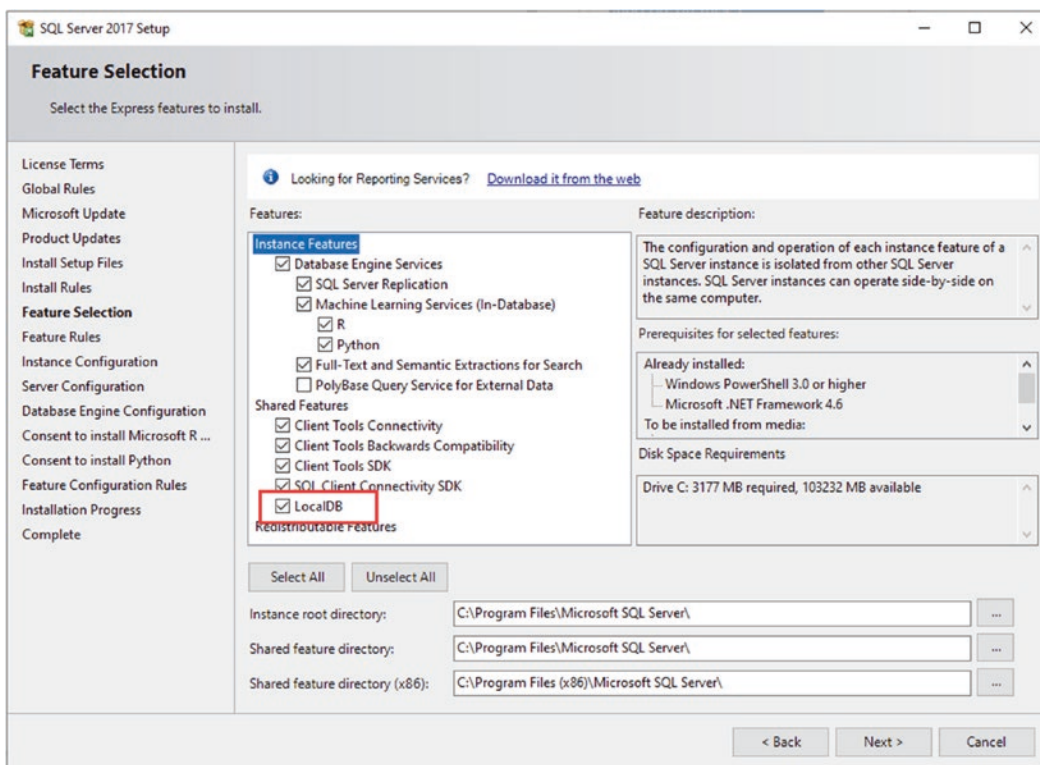


Figure 2-9. Selecting the LocalDB feature

On the Instance Configuration page, select the “Default instance” option, as shown in Figure 2-10.

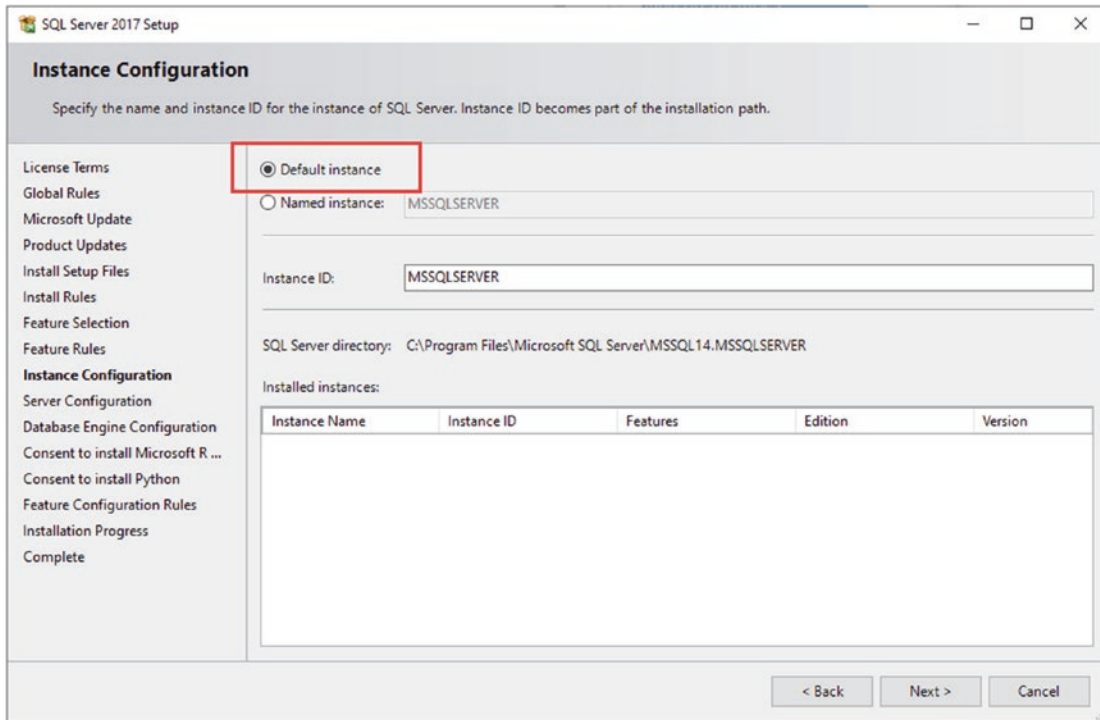


Figure 2-10. Configuring the database

Continue to work through the installation process, selecting the default values. Once the installation is complete, install the latest cumulative update for SQL Server. At the time of writing, the latest update is available at <https://support.microsoft.com/en-us/help/4527377/cumulative-update-18-for-sql-server-2017>, although newer updates may have been released by the time you read this chapter.

■ **Caution** It can be tempting to skip the update stage, but it is important to perform this step to get the expected results from the examples in this book. As an example, the base installation of SQL Server has a bug that prevents LocalDB from creating database files, which will cause problems when you reach Chapter 7.

Creating an ASP.NET Core Project

The most direct way to create a project is to use the command line, although Visual Studio provides a wizard system that I demonstrate in Chapter 4. Open a new PowerShell command prompt from the Windows Start menu, navigate to the folder where you want to create your ASP.NET Core projects, and run the commands shown in Listing 2-3.

■ **Tip** You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/apress/pro-asp.net-core-3>. See Chapter 1 for how to get help if you have problems running the examples.

Listing 2-3. Creating a New Project

```
dotnet new globaljson --sdk-version 3.1.101 --output FirstProject
dotnet new mvc --no-https --output FirstProject --framework netcoreapp3.1
```

The first command creates a folder named `FirstProject` and adds to it a file named `global.json`, which specifies the version of .NET Core that the project will use; this ensures you get the expected results when following the examples. The second command creates a new ASP.NET Core project. The .NET Core SDK includes a range of templates for starting new projects, and the `mvc` template is one of the options available for ASP.NET Core applications. This project template creates a project that is configured for the MVC Framework, which is one of the application types supported by ASP.NET Core. Don't be intimidated by the idea of choosing a framework, and don't worry if you have not heard of MVC—by the end of the book, you will understand the features that each offers and how they fit together.

■ **Note** This is one of a small number of chapters in which I use a project template that contains placeholder content. I don't like using predefined project templates because they encourage developers to treat important features, such as authentication, as black boxes. My goal in this book is to give you the knowledge to understand and manage every aspect of your ASP.NET Core applications, and that's why I start with an empty ASP.NET Core project. This chapter is about getting started quickly, for which the `mvc` template is well-suited.

Opening the Project Using Visual Studio

Start Visual Studio and click the “Open a project or solution” button, as shown in Figure 2-11.

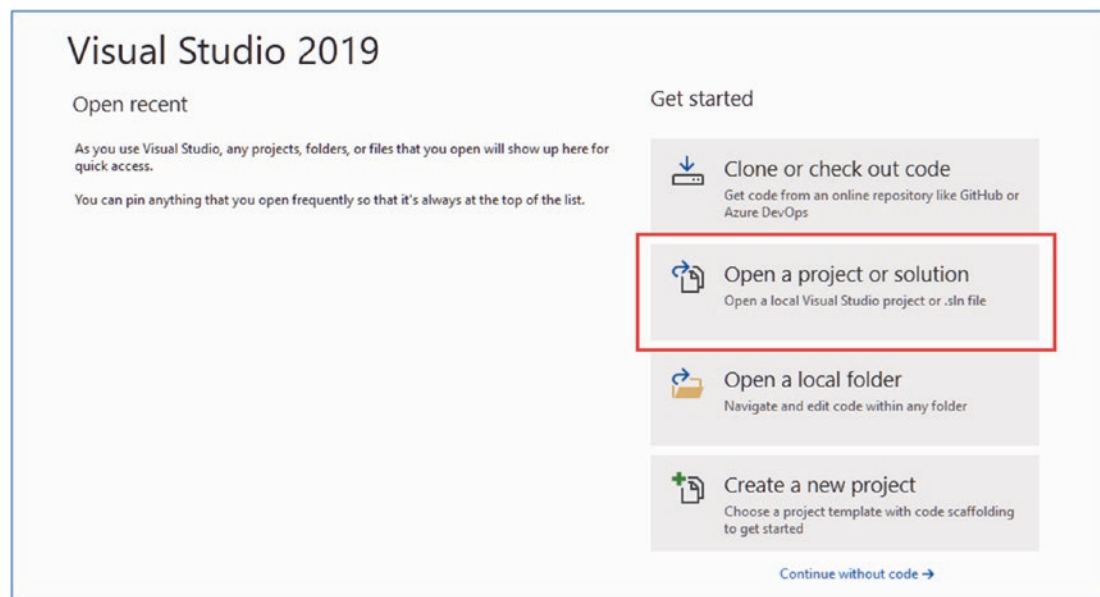


Figure 2-11. Opening the ASP.NET Core project

Navigate to the `FirstProject` folder, select the `FirstProject.csproj` file, and click the Open button. Visual Studio will open the project and display its contents in the Solution Explorer window, as shown in Figure 2-12. The files in the project were created by the project template.

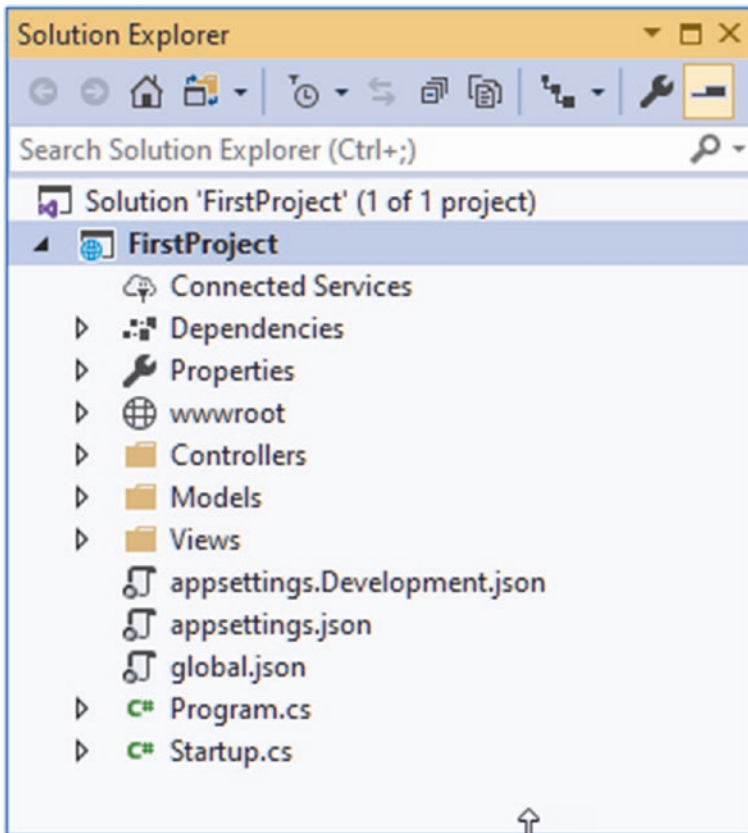


Figure 2-12. Opening the project in Visual Studio

Choosing a Browser

Visual Studio will open a browser window automatically when the project is run. To select the browser that is used, click the small arrow to the right of the IIS Express drop-down and select your preferred browser from the Web Browser menu, as shown in Figure 2-13. I use Google Chrome throughout this book.

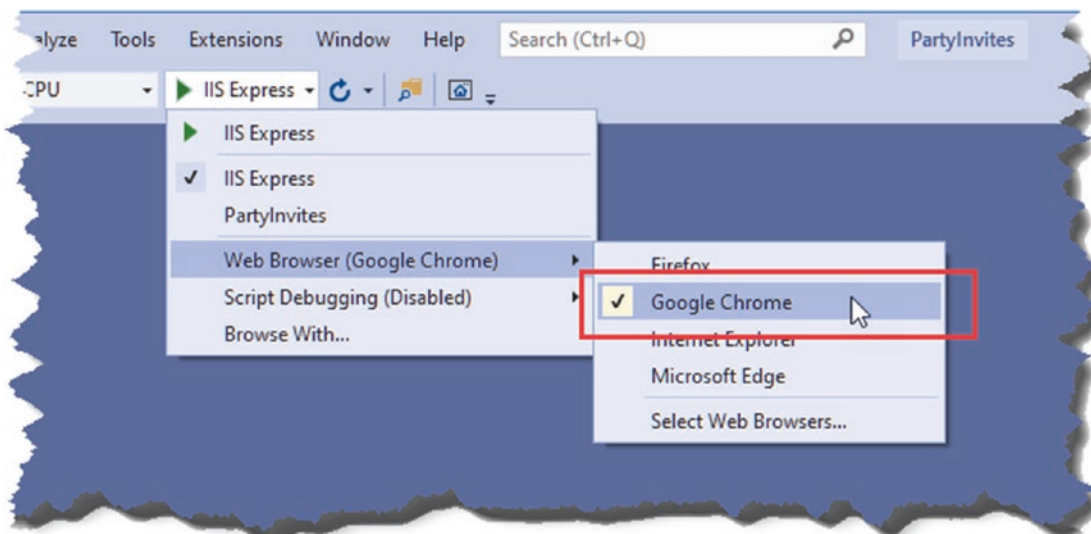


Figure 2-13. Selecting the browser

Opening the Project with Visual Studio Code

Start Visual Studio Code and select **File** ► **Open Folder**. Navigate to the `FirstProject` folder and click the **Select Folder** button. Visual Studio Code will open the project and display its contents in the Explorer pane, as shown in Figure 2-14. (The default dark theme used in Visual Studio Code doesn't show well on the page, so I have changed to the light theme for the screenshots in this book.)

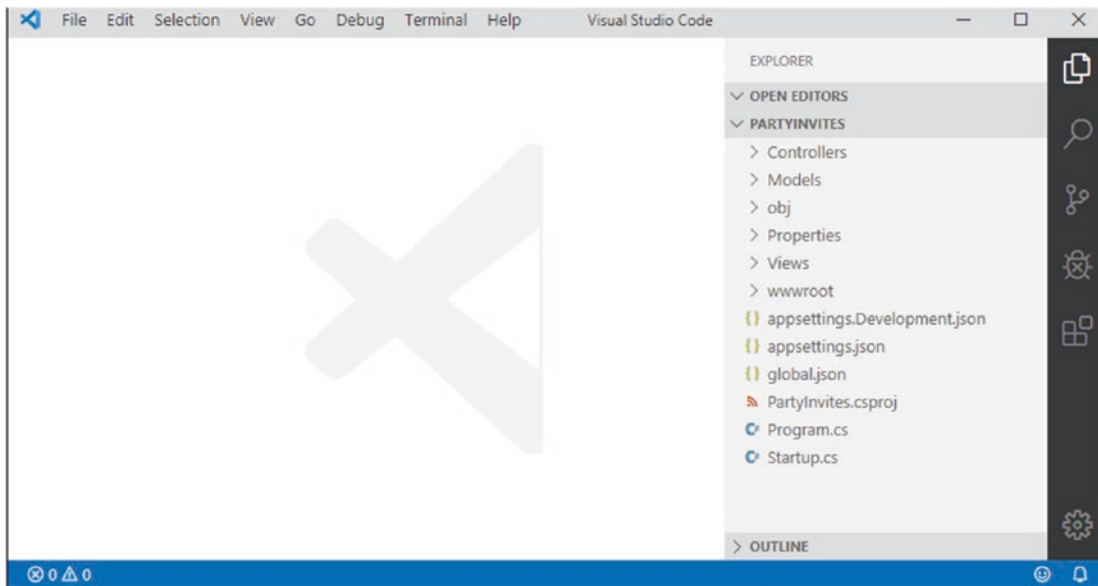


Figure 2-14. Opening the project in Visual Studio Code

Additional configuration is required the first time you open a .NET Core project in Visual Studio Code. The first step is to click the `Startup.cs` file in the Explorer pane. This will trigger a prompt from Visual Studio Code to install the features required for C# development, as shown in Figure 2-15. If you have opened a C# project before, you will see a prompt that offers to install the required assets, also shown in Figure 2-15.

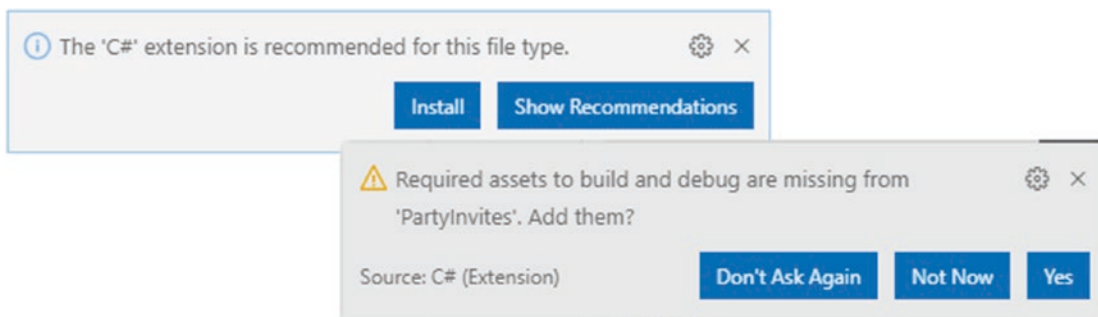


Figure 2-15. Installing Visual Studio Code C# features

Click the **Install** or **Yes** button, as appropriate, and Visual Studio Code will download and install the features required for .NET Core projects.

Running the ASP.NET Core Application

The template creates a project that contains everything needed to build and run the application. Select **Start Without Debugging** from the **Debug** menu, and Visual Studio will compile and start the example application and then open a new browser window to send the application an HTTP request, as shown in Figure 2-17. (If you don't see the **Start Without Debugging** item in the **Debug** menu, then click the `Startup.cs` file in the **Solution Explorer** window and check the menu again.)

If you are using Visual Studio Code, select **Run Without Debugging** in the **Debug** menu. Since this is the first time the project has been started, you will be prompted to select an execution environment. Select the **.NET Core** option, as shown in Figure 2-16.

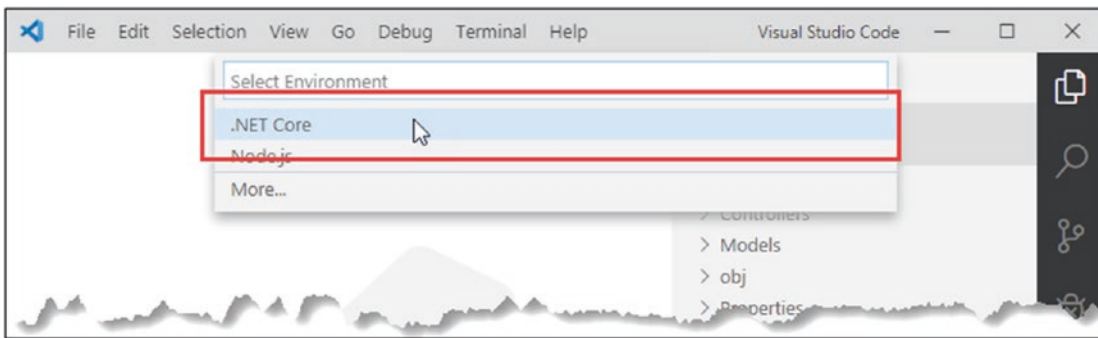


Figure 2-16. Selecting an execution environment

Visual Studio Code will create a `launch.json` file that contains the startup settings for the project and that you can ignore for this book. Select **Run Without Debugging** from the **Debug** menu again, and Visual Studio Code will compile the project, start executing the application, and open a new browser window, as shown in Figure 2-17.

You can also start the application from the command line. Open a new PowerShell command prompt from the Windows Start menu; navigate to the `FirstProject` project folder, which is the folder that contains the `FirstProject.csproj` file; and run the command shown in Listing 2-4.

Listing 2-4. Starting the Example Application

```
dotnet run
```

Once the application has started, you will need to open a new browser window and request `http://localhost:5000`, which will produce the response shown in Figure 2-17.

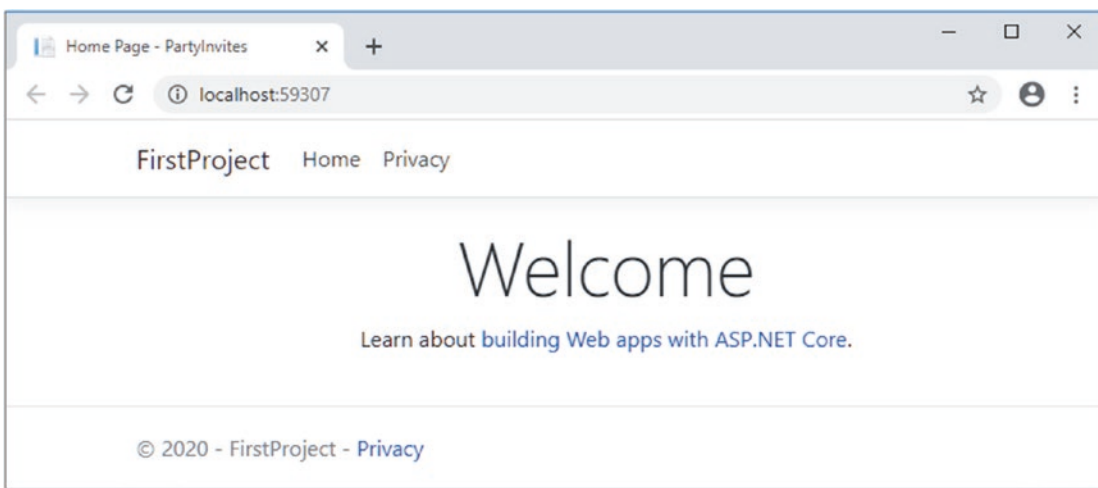


Figure 2-17. Running the example project

■ **Tip** If you are using Visual Studio Code or running the application from the command line, then ASP.NET Core listens for HTTP requests on port 5000. If you are using Visual Studio, you will notice that the browser requests a different port, which is chosen when the project is created. If you look in the Windows taskbar notification area, you will find an icon for IIS Express. This is a cut-down version of the full IIS application server that is included with Visual Studio and is used to deliver ASP.NET Core content and services during development. In later chapters, I show you how to change the project configuration to use the same HTTP port as Visual Studio Code.

When you are finished, close the browser window that Visual Studio opened. If you are using Visual Studio Code, you click the stop button displayed in the window that pops up over the code editor. If you started the application from the command line, then use Control+C to stop execution.

Understanding Endpoints

In an ASP.NET Core application, incoming requests are handled by *endpoints*. The endpoint that produced the response in Figure 2-17 is an *action*, which is a method that is written in C#. An action is defined in a *controller*, which is a C# class that is derived from the `Microsoft.AspNetCore.Mvc.Controller` class, the built-in controller base class.

Each public method defined by a controller is an action, which means you can invoke the action method to handle an HTTP request. The convention in ASP.NET Core projects is to put controller classes in a folder named `Controllers`, which was created by the template used to set up the project in Listing 2-3.

The project template added a controller to the `Controllers` folder to help jump-start development. The controller is defined in the class file named `HomeController.cs`. Controller classes contain a name followed by the word `Controller`, which means that when you see a file called `HomeController.cs`, you know that it contains a controller called `Home`, which is the default controller that is used in ASP.NET Core applications.

■ **Tip** Don't worry if the terms *controller* and *action* don't make immediate sense. Just keep following the example, and you will see how the HTTP request sent by the browser is handled by C# code.

Find the `HomeController.cs` file in the Solution Explorer or Explorer pane and click it to open it for editing. You will see the following code:

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.Extensions.Logging;
using FirstProject.Models;

namespace FirstProject.Controllers {
    public class HomeController : Controller {
        private readonly ILogger<HomeController> _logger;

        public HomeController(ILogger<HomeController> logger) {
            _logger = logger;
        }

        public IActionResult Index() {
            return View();
        }

        public IActionResult Privacy() {
            return View();
        }
    }
}
```

```

    [ResponseCache(Duration = 0, Location = ResponseCacheLocation.None,
        NoStore = true)]
    public IActionResult Error() {
        return View(new ErrorViewModel { RequestId = Activity.Current?.Id
            ?? HttpContext.TraceIdentifier });
    }
}
}

```

Using the code editor, replace the contents of the `HomeController.cs` file so that it matches Listing 2-5. I have removed all but one of the methods, changed the result type and its implementation, and removed the using statements for unused namespaces.

Listing 2-5. Changing the `HomeController.cs` File in the Controllers Folder

```

using Microsoft.AspNetCore.Mvc;

namespace FirstProject.Controllers {

    public class HomeController : Controller {

        public string Index() {
            return "Hello World";
        }
    }
}

```

The result is that the Home controller defines a single action, named `Index`. These changes don't produce a dramatic effect, but they make for a nice demonstration. I have changed the method named `Index` so that it returns the string `Hello World`. Run the project again by selecting `Start Without Debugging` or `Run Without Debugging` from the `Debug` menu.

The browser will make an HTTP request to the ASP.NET Core server. The configuration of the project created by the template in Listing 2-5 means the HTTP request will be processed by the `Index` action defined by the Home controller. Put another way, the request will be processed by the `Index` method defined by the `HomeController` class. The string produced by the `Index` method is used as the response to the browser's HTTP request, as shown in Figure 2-18.

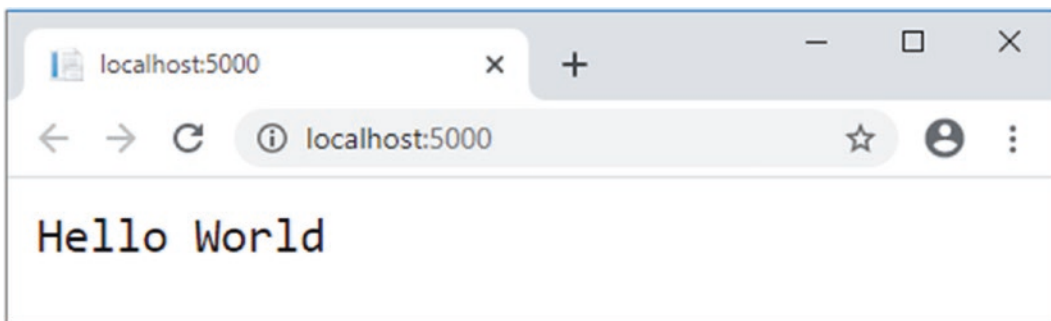


Figure 2-18. The output from the action method

Understanding Routes

The ASP.NET Core *routing* system is responsible for selecting the endpoint that will handle an HTTP request. A *route* is a rule that is used to decide how a request is handled. When the project was created, a default rule was created to get started. You can request any of the following URLs, and they will be dispatched to the `Index` action defined by the Home controller:

- /
- /Home
- /Home/Index

So, when a browser requests `http://yoursite/` or `http://yoursite/Home`, it gets back the output from `HomeController`'s `Index` method. You can try this yourself by changing the URL in the browser. At the moment, it will be `http://localhost:5000/`, except that the port part may be different if you are using Visual Studio. If you append `/Home` or `/Home/Index` to the URL and press Return, you will see the same `Hello World` result from the application.

Understanding HTML Rendering

The output from the previous example wasn't HTML—it was just the string `Hello World`. To produce an HTML response to a browser request, I need a *view*, which tells ASP.NET Core how to process the result produced by the `Index` method into an HTML response that can be sent to the browser.

Creating and Rendering a View

The first thing I need to do is modify my `Index` action method, as shown in Listing 2-6. The changes are shown in bold, which is a convention I follow throughout this book to make the examples easier to follow.

Listing 2-6. Rendering a View in the `HomeController.cs` File in the `Controllers` Folder

```
using Microsoft.AspNetCore.Mvc;

namespace FirstProject.Controllers {

    public class HomeController : Controller {

        public IActionResult Index() {
            return View("MyView");
        }
    }
}
```

When I return a `ViewResult` object from an action method, I am instructing ASP.NET Core to *render* a view. I create the `ViewResult` by calling the `View` method, specifying the name of the view that I want to use, which is `MyView`. If you run the application, you can see ASP.NET Core trying to find the view, as shown by the error message displayed in Figure 2-19.

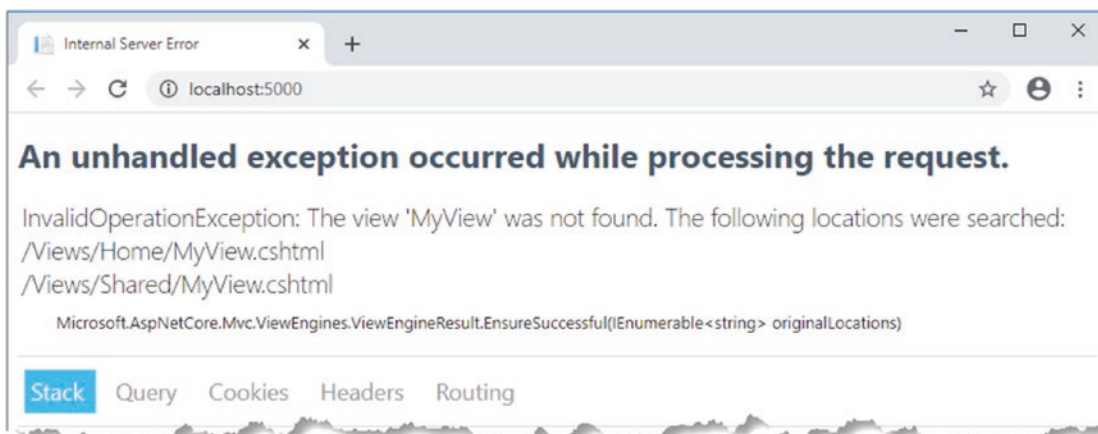


Figure 2-19. Trying to find a view

This is a helpful error message. It explains that ASP.NET Core could not find the view I specified for the action method and explains where it looked. Views are stored in the Views folder, organized into subfolders. Views that are associated with the Home controller, for example, are stored in a folder called Views/Home. Views that are not specific to a single controller are stored in a folder called Views/Shared. The template used to create the project added the Home and Shared folders automatically and added some placeholder views to get the project started.

If you are using Visual Studio, right-click the Views/Home folder in the Solution Explorer and select Add ► New Item from the popup menu. Visual Studio will present you with a list of templates for adding items to the project. Locate the Razor View item, which can be found in the ASP.NET Core ► Web ► ASP.NET section, as shown in Figure 2-20. Set the name of the new file to MyView.cshtml and click the Add button. Visual Studio will add a file named MyView.cshtml to the Views/Home folder and will open it for editing. Replace the contents of the file with those shown in Listing 2-7.

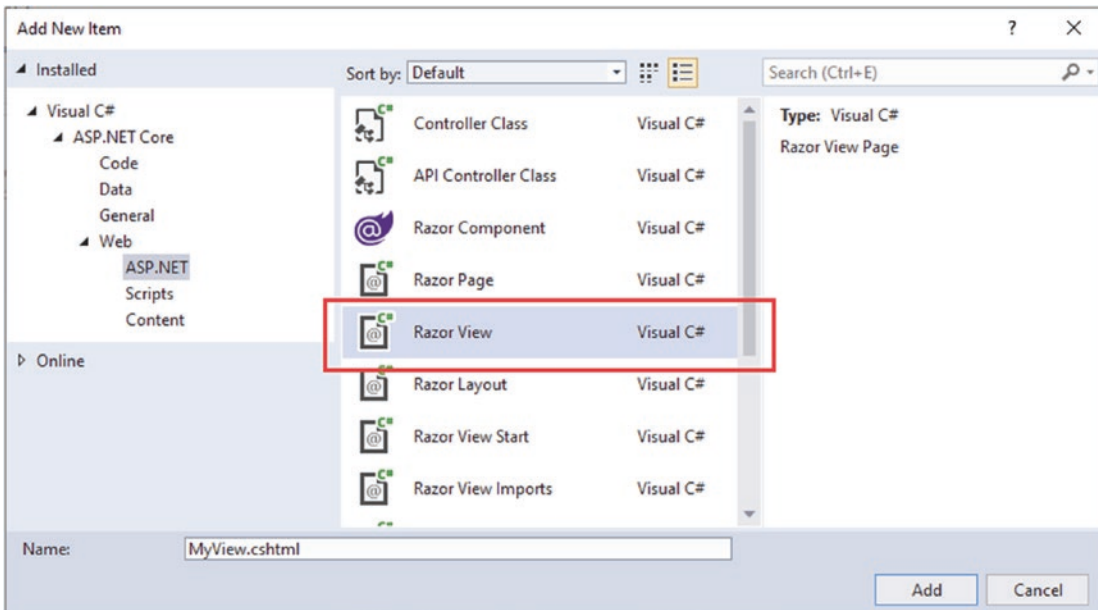


Figure 2-20. Selecting a Visual Studio item template

Visual Studio Code doesn't provide item templates. Instead, right-click the Views/Home folder in the file explorer pane and select New File from the popup menu. Set the name of the file to MyView.cshtml and press Return. The file will be created and opened for editing. Add the content shown in Listing 2-7.

■ **Tip** It is easy to end up creating the view file in the wrong folder. If you didn't end up with a file called MyView.cshtml in the Views/Home folder, then either drag the file into the correct folder or delete the file and try again.

Listing 2-7. The Contents of the MyView.cshtml File in the Views/Home Folder

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
```

```

<body>
  <div>
    Hello World (from the view)
  </div>
</body>
</html>

```

The new contents of the view file are mostly HTML. The exception is the part that looks like this:

```

...
@{
  Layout = null;
}
...

```

This is an expression that will be interpreted by Razor, which is the component that processes the contents of views and generates HTML that is sent to the browser. Razor is a *view engine*, and the expressions in views are known as *Razor expressions*.

The Razor expression in Listing 2-7 tells Razor that I chose not to use a layout, which is like a template for the HTML that will be sent to the browser (and which I describe in Chapter 22). To see the effect of creating the view, stop ASP.NET Core if it is running and select Start Without Debugging (for Visual Studio) or Run Without Debugging (for Visual Studio Code) from the Debug menu. A new browser window will open and produce the response shown in Figure 2-21.

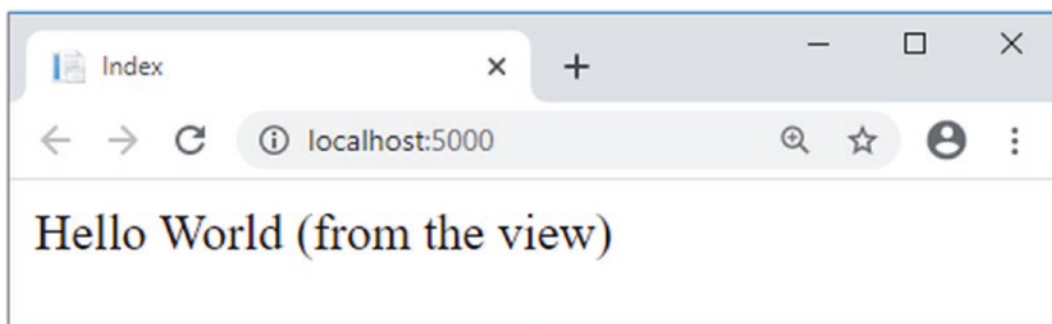


Figure 2-21. Rendering a view

When I first edited the Index action method, it returned a string value. This meant that ASP.NET Core did nothing except pass the string value as is to the browser. Now that the Index method returns a `ViewResult`, Razor is used to process a view and render an HTML response. Razor was able to locate the view because I followed the standard naming convention, which is to put view files in a folder whose name matched the controller that contains the action method. In this case, this meant putting the view file in the `Views/Home` folder, since the action method is defined by the Home controller.

I can return other results from action methods besides strings and `ViewResult` objects. For example, if I return a `RedirectResult`, the browser will be redirected to another URL. If I return an `HttpUnauthorizedResult`, I can prompt the user to log in. These objects are collectively known as *action results*. The action result system lets you encapsulate and reuse common responses in actions. I'll tell you more about them and explain the different ways they can be used in Chapter 19.

Adding Dynamic Output

The whole point of a web application is to construct and display *dynamic* output. The job of the action method is to construct data and pass it to the view so it can be used to create HTML content based on the data values. Action methods provide data to views by passing arguments to the View method, as shown in Listing 2-8. The data provided to the view is known as the *view model*.

Listing 2-8. Providing a View Model in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System;

namespace FirstProject.Controllers {

    public class HomeController : Controller {

        public IActionResult Index() {
            int hour = DateTime.Now.Hour;
            string viewModel = hour < 12 ? "Good Morning" : "Good Afternoon";
            return View("MyView", viewModel);
        }
    }
}
```

The view model in this example is a `string`, and it is provided to the view as the second argument to the `View` method. Listing 2-9 updates the view so that it receives and uses the view model in the HTML it generates.

Listing 2-9. Using a View Model in the MyView.cshtml File in the Views/Home Folder

```
@model string
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Index</title>
</head>
<body>
    <div>
        @Model World (from the view)
    </div>
</body>
</html>
```

The type of the view model is specified using the `@model` expression, with a lowercase *m*. The view model value is included in the HTML output using the `@Model` expression, with an uppercase *M*. (It can be difficult at first to remember which is lowercase and which is uppercase, but it soon becomes second nature.)

When the view is rendered, the view model data provided by the action method is inserted into the HTML response. Select Start Without Debugging (using Visual Studio) or Run Without Debugging (using Visual Studio Code), and you will see the output shown in Figure 2-22 (although you may see the afternoon greeting if you are following this example after midday).

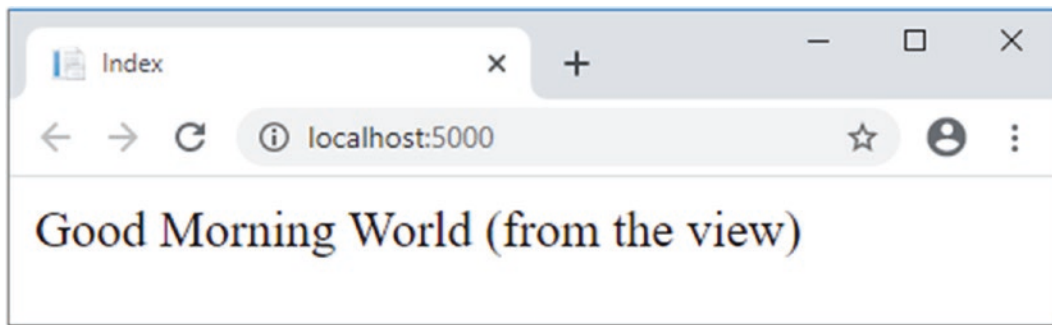


Figure 2-22. *Generating dynamic content*

Putting the Pieces Together

It is a simple result, but this example reveals all the building blocks you need to create a simple ASP.NET Core web application and to generate a dynamic response. The ASP.NET Core platform receives an HTTP request and uses the routing system to match the request URL to an endpoint. The endpoint, in this case, is the `Index` action method defined by the `Home` controller. The method is invoked and produces a `ViewResult` object that contains the name of a view and a view model object. The Razor view engine locates and processes the view, evaluating the `@Model` expression to insert the data provided by the action method into the response, which is returned to the browser and displayed to the user. There are, of course, many other features available, but this is the essence of ASP.NET Core, and it is worth bearing this simple sequence in mind as you read the rest of the book.

Summary

In this chapter, I explained how to get set up for ASP.NET Core development by installing Visual Studio or Visual Studio Code and the .NET Core SDK. I showed you how to create a simple project and briefly explained how the endpoint, the view, and the URL routing system work together. In the next chapter, I show you how to create a simple data-entry application.

CHAPTER 3



Your First ASP.NET Core Application

Now that you are set up for ASP.NET Core development, it is time to create a simple application. In this chapter, you'll create a data-entry application using ASP.NET Core. My goal is to demonstrate ASP.NET Core in action, so I will pick up the pace a little and skip over some of the explanations as to how things work behind the scenes. But don't worry; I'll revisit these topics in depth in later chapters.

Setting the Scene

Imagine that a friend has decided to host a New Year's Eve party and that she has asked me to create a web app that allows her invitees to electronically RSVP. She has asked for these four key features:

- A home page that shows information about the party
- A form that can be used to RSVP
- Validation for the RSVP form, which will display a thank-you page
- A summary page that shows who is coming to the party

In this chapter, I create an ASP.NET Core project and use it to create a simple application that contains these features; once everything works, I'll apply some styling to improve the appearance of the finished application.

Creating the Project

Open a PowerShell command prompt from the Windows Start menu, navigate to a convenient location, and run the commands in Listing 3-1 to create a project named PartyInvites.

■ **Tip** You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/apress/pro-asp.net-core-3>. See Chapter 1 for how to get help if you have problems running the examples.

Listing 3-1. Creating a New Project

```
dotnet new globaljson --sdk-version 3.1.101 --output PartyInvites
dotnet new mvc --no-https --output PartyInvites --framework netcoreapp3.1
```

These are the same commands I used to create the project in Chapter 2. If you are a Visual Studio user, I explain how you can use a wizard to create a project in Chapter 4, but these commands are simple and will ensure you get the right project starting point that uses the required version of .NET Core.

Open the project and edit the `HomeController.cs` file in the `Controllers` folder, replacing the contents with the code shown in Listing 3-2.

Listing 3-2. Replacing the Contents of the `HomeController.cs` File in the `Controllers` Folder

```
using Microsoft.AspNetCore.Mvc;

namespace PartyInvites.Controllers {
    public class HomeController : Controller {

        public IActionResult Index() {
            return View();
        }
    }
}
```

This provides a clean starting point for the new application, defining a single action method that selects the default view for rendering. To provide a welcome message to party invitees, open the `Index.cshtml` file in the `Views/Home` folder and replace the contents with those shown in Listing 3-3.

Listing 3-3. Replacing the Contents of the `Index.cshtml` File in the `Views/Home` Folder

```
@{
    Layout = null;
}

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Party!</title>
</head>
<body>
    <div>
        <div>
            We're going to have an exciting party.<br />
            (To do: sell it better. Add pictures or something.)
        </div>
    </div>
</body>
</html>
```

Start the application by selecting `Start Without Debugging` (for Visual Studio) or `Run Without Debugging` (for Visual Studio Code), and you will see the details of the party (well, the placeholder for the details, but you get the idea), as shown in Figure 3-1.

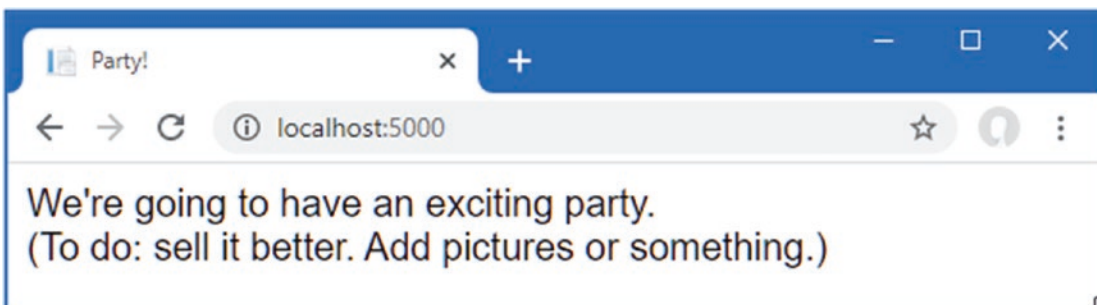


Figure 3-1. Adding to the view HTML

Adding a Data Model

The data model is the most important part of any ASP.NET Core application. The model is the representation of the real-world objects, processes, and rules that define the subject, known as the *domain*, of the application. The model, often referred to as a *domain model*, contains the C# objects (known as *domain objects*) that make up the universe of the application and the methods that manipulate them. In most projects, the job of the ASP.NET Core application is to provide the user with access to the data model and the features that allow the user to interact with it.

The convention for an ASP.NET Core application is that the data model classes are defined in a folder named `Models`, which was added to the project by the template used in Listing 3-1.

I don't need a complex model for the `PartyInvites` project because it is such a simple application. In fact, I need just one domain class that I will call `GuestResponse`. This object will represent an RSVP from an invitee.

If you are using Visual Studio, right-click the `Models` folder and select `Add ► Class` from the popup menu. Set the name of the class to `GuestResponse.cs` and click the `Add` button. If you are using Visual Studio Code, right-click the `Models` folder, select `New File`, and enter `GuestResponse.cs` as the file name. Use the new file to define the class shown in Listing 3-4.

Listing 3-4. The Contents of the `GuestResponse.cs` File in the `Models` Folder

```
namespace PartyInvites.Models {
    public class GuestResponse {
        public string Name { get; set; }
        public string Email { get; set; }
        public string Phone { get; set; }
        public bool? WillAttend { get; set; }
    }
}
```

■ **Tip** You may have noticed that the `WillAttend` property is a nullable `bool`, which means that it can be `true`, `false`, or `null`. I explain the rationale for this in the “Adding Validation” section later in the chapter.

Creating a Second Action and View

One of my application goals is to include an RSVP form, which means I need to define an action method that can receive requests for that form. A single controller class can define multiple action methods, and the convention is to group related actions together in the same controller. Listing 3-5 adds a new action method to the `Home` controller.

Listing 3-5. Adding an Action Method in the `HomeController.cs` File in the `Controllers` Folder

```
using Microsoft.AspNetCore.Mvc;

namespace PartyInvites.Controllers {
    public class HomeController : Controller {
        public IActionResult Index() {
            return View();
        }

        public IActionResult RsvpForm() {
            return View();
        }
    }
}
```


Both action methods invoke the View method without arguments, which may seem odd, but remember that the Razor view engine will use the name of the action method when looking for a view file. That means the result from the Index action method tells Razor to look for a view called `Index.cshtml`, while the result from the `RsvpForm` action method tells Razor to look for a view called `RsvpForm.cshtml`.

If you are using Visual Studio, right-click the `Views/Home` folder and select `Add ► New Item` from the popup menu. Select the `Razor View` item, set the name to `RsvpForm.cshtml`, and click the `Add` button to create the file. Replace the contents with those shown in Listing 3-6.

If you are using Visual Studio Code, right-click the `Views/Home` folder and select `New File` from the popup menu. Set the name of the file to `RsvpForm.cshtml` and add the contents shown in Listing 3-6.

Listing 3-6. The Contents of the `RsvpForm.cshtml` File in the `Views/Home` Folder

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>RsvpForm</title>
</head>
<body>
    <div>
        This is the RsvpForm.cshtml View
    </div>
</body>
</html>
```

This content is just static HTML for the moment. To test the new action method and view, start the application by selecting `Start Without Debugging` or `Run Without Debugging` from the `Debug` menu.

Using the browser window that is opened, request `http://localhost:5000/home/rsvpform`. (If you are using Visual Studio, you will have to change the port to the one assigned when the project was created.) The Razor view engine locates the `RsvpForm.cshtml` file and uses it to produce a response, as shown in Figure 3-2.

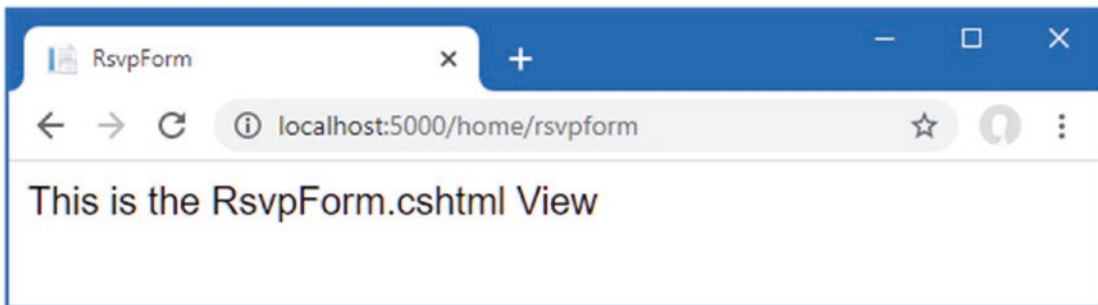


Figure 3-2. Rendering a second view

Linking Action Methods

I want to be able to create a link from the `Index` view so that guests can see the `RsvpForm` view without having to know the URL that targets a specific action method, as shown in Listing 3-7.

Listing 3-7. Adding a Link in the Index.cshtml File in the Views/Home Folder

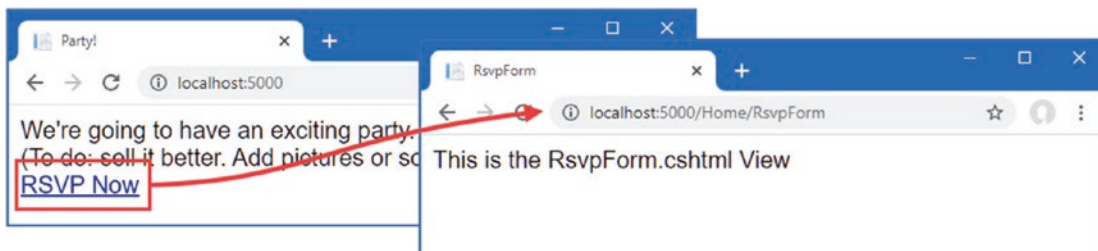
```

@{
    Layout = null;
}

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Party!</title>
</head>
<body>
    <div>
        <div>
            We're going to have an exciting party.<br />
            (To do: sell it better. Add pictures or something.)
        </div>
        <a asp-action="RsvpForm">RSVP Now</a>
    </div>
</body>
</html>

```

The addition to the listing is an `a` element that has an `asp-action` attribute. The attribute is an example of a *tag helper* attribute, which is an instruction for Razor that will be performed when the view is rendered. The `asp-action` attribute is an instruction to add an `href` attribute to the `a` element that contains a URL for an action method. I explain how tag helpers work in Chapters 25–27, but this tag helper tells Razor to insert a URL for an action method defined by the same controller for which the current view is being rendered. You can see the link that the helper creates by running the project, as shown in Figure 3-3.

**Figure 3-3.** Linking between action methods

Roll the mouse over the RSVP Now link in the browser. You will see that the link points to the following URL (allowing for the different port number that Visual Studio will have assigned to your project):

```
http://localhost:5000/Home/RsvpForm
```

There is an important principle at work here, which is that you should use the features provided by ASP.NET Core to generate URLs, rather than hard-code them into your views. When the tag helper created the `href` attribute for the `a` element, it inspected the configuration of the application to figure out what the URL should be. This allows the configuration of the application to be changed to support different URL formats without needing to update any views.

Building the Form

Now that I have created the view and can reach it from the Index view, I am going to build out the contents of the `RsvpForm.cshtml` file to turn it into an HTML form for editing `GuestResponse` objects, as shown in Listing 3-8.

Listing 3-8. Creating a Form View in the RsvpForm.cshtml File in the Views/Home Folder

```
@model PartyInvites.Models.GuestResponse
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>RsvpForm</title>
</head>
<body>
    <form asp-action="RsvpForm" method="post">
        <div>
            <label asp-for="Name">Your name:</label>
            <input asp-for="Name" />
        </div>
        <div>
            <label asp-for="Email">Your email:</label>
            <input asp-for="Email" />
        </div>
        <div>
            <label asp-for="Phone">Your phone:</label>
            <input asp-for="Phone" />
        </div>
        <div>
            <label>Will you attend?</label>
            <select asp-for="WillAttend">
                <option value="">Choose an option</option>
                <option value="true">Yes, I'll be there</option>
                <option value="false">No, I can't come</option>
            </select>
        </div>
        <button type="submit">Submit RSVP</button>
    </form>
</body>
</html>
```

The `@model` expression specifies that the view expects to receive a `GuestResponse` object as its view model. I have defined a label and input element for each property of the `GuestResponse` model class (or, in the case of the `WillAttend` property, a select element). Each element is associated with the model property using the `asp-for` attribute, which is another tag helper attribute. The tag helper attributes configure the elements to tie them to the view model object. Here is an example of the HTML that the tag helpers produce:

```
<p>
    <label for="Name">Your name:</label>
    <input type="text" id="Name" name="Name" value="">
</p>
```

The `asp-for` attribute on the label element sets the value of the `for` attribute. The `asp-for` attribute on the input element sets the `id` and `name` elements. This may not look especially useful, but you will see that associating elements with a model property offers additional advantages as the application functionality is defined.

Of more immediate use is the `asp-action` attribute applied to the form element, which uses the application's URL routing configuration to set the action attribute to a URL that will target a specific action method, like this:

```
<form method="post" action="/Home/RsvpForm">
```

As with the helper attribute I applied to the `a` element, the benefit of this approach is that you can change the system of URLs that the application uses, and the content generated by the tag helpers will reflect the changes automatically.

You can see the form by running the application and clicking the RSVP Now link, as shown in Figure 3-4.

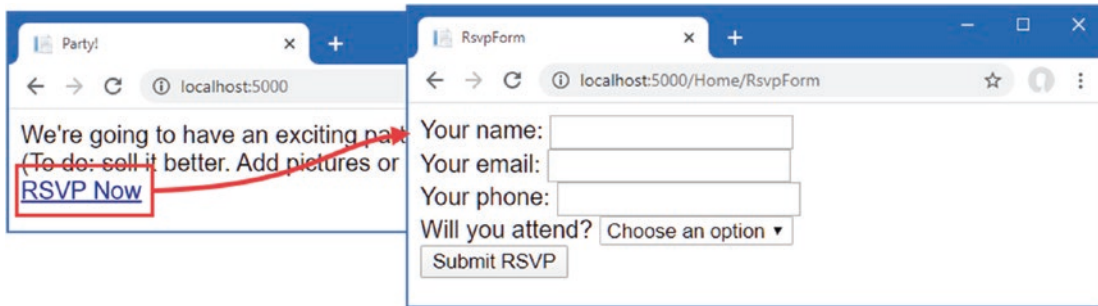


Figure 3-4. Adding an HTML form to the application

Receiving Form Data

I have not yet told ASP.NET Core what I want to do when the form is posted to the server. As things stand, clicking the Submit RSVP button just clears any values you have entered in the form. That is because the form posts back to the `RsvpForm` action method in the Home controller, which just renders the view again. To receive and process submitted form data, I am going to use an important feature of controllers. I will add a second `RsvpForm` action method to create the following:

- *A method that responds to HTTP GET requests:* A GET request is what a browser issues normally each time someone clicks a link. This version of the action will be responsible for displaying the initial blank form when someone first visits `/Home/RsvpForm`.
- *A method that responds to HTTP POST requests:* By default, forms rendered using `Html.BeginForm()` are submitted by the browser as a POST request. This version of the action will be responsible for receiving submitted data and deciding what to do with it.

Handling GET and POST requests in separate C# methods helps to keep my controller code tidy since the two methods have different responsibilities. Both action methods are invoked by the same URL, but ASP.NET Core makes sure that the appropriate method is called, based on whether I am dealing with a GET or POST request. Listing 3-9 shows the changes to the Home Controller class.

Listing 3-9. Adding a Method in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using PartyInvites.Models;

namespace PartyInvites.Controllers {
    public class HomeController : Controller {

        public IActionResult Index() {
            return View();
        }

        [HttpGet]
        public IActionResult RsvpForm() {
            return View();
        }
    }
}
```

```

    [HttpPost]
    public IActionResult RsvpForm(GuestResponse guestResponse) {
        // TODO: store response from guest
        return View();
    }
}

```

I have added the `HttpGet` attribute to the existing `RsvpForm` action method, which declares that this method should be used only for GET requests. I then added an overloaded version of the `RsvpForm` method, which accepts a `GuestResponse` object. I applied the `HttpPost` attribute to this method, which declares that the new method will deal with POST requests. I explain how these additions to the listing work in the following sections. I also imported the `PartyInvites.Models` namespace—this is just so I can refer to the `GuestResponse` model type without needing to qualify the class name.

Understanding Model Binding

The first overload of the `RsvpForm` action method renders the same view as before—the `RsvpForm.cshtml` file—to generate the form shown in Figure 3-4. The second overload is more interesting because of the parameter, but given that the action method will be invoked in response to an HTTP POST request and that the `GuestResponse` type is a C# class, how are the two connected?

The answer is *model binding*, a useful ASP.NET Core feature whereby incoming data is parsed and the key/value pairs in the HTTP request are used to populate properties of domain model types.

Model binding is a powerful and customizable feature that eliminates the grind of dealing with HTTP requests directly and lets you work with C# objects rather than dealing with individual data values sent by the browser. The `GuestResponse` object that is passed as the parameter to the action method is automatically populated with the data from the form fields. I dive into the details of model binding in Chapter 28.

To demonstrate how model binding works, I need to do some preparatory work. One of the application goals is to present a summary page with details of who is attending the party, which means that I need to keep track of the responses that I receive. I am going to do this by creating an in-memory collection of objects. This isn't useful in a real application because the response data will be lost when the application is stopped or restarted, but this approach will allow me to keep the focus on ASP.NET Core and create an application that can easily be reset to its initial state. Later chapters will demonstrate persistent data storage.

Add a class file named `Repository.cs` to the `Models` folder and use it to define the class shown in Listing 3-10.

Listing 3-10. The Contents of the `Repository.cs` File in the `Models` Folder

```

using System.Collections.Generic;

namespace PartyInvites.Models {
    public static class Repository {
        private static List<GuestResponse> responses = new List<GuestResponse>();

        public static IEnumerable<GuestResponse> Responses => responses;

        public static void AddResponse(GuestResponse response) {
            responses.Add(response);
        }
    }
}

```

The `Repository` class and its members are `static`, which will make it easy for me to store and retrieve data from different places in the application. ASP.NET Core provides a more sophisticated approach for defining common functionality, called *dependency injection*, which I describe in Chapter 14, but a static class is a good way to get started for a simple application like this one.

Storing Responses

Now that I have somewhere to store the data, I can update the action method that receives the HTTP POST requests, as shown in Listing 3-11.

Listing 3-11. Updating an Action Method in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using PartyInvites.Models;

namespace PartyInvites.Controllers {
    public class HomeController : Controller {

        public IActionResult Index() {
            return View();
        }

        [HttpGet]
        public IActionResult RsvpForm() {
            return View();
        }

        [HttpPost]
        public IActionResult RsvpForm(GuestResponse guestResponse) {
            Repository.AddResponse(guestResponse);
            return View("Thanks", guestResponse);
        }
    }
}
```

Before the POST version of the `RsvpForm` method is invoked, the ASP.NET Core model binding feature extracts values from the HTML form and assigns them to the properties of the `GuestResponse` object. The result is used as the argument when the method is invoked to handle the HTTP request, and all I have to do to deal with the form data sent in a request is to work with the `GuestResponse` object that is passed to the action method—in this case, to pass it as an argument to the `Repository.AddResponse` method so that the response can be stored.

Adding the Thanks View

The call to the `View` method in the `RsvpForm` action method creates a `ViewResult` that selects a view called `Thanks` and uses the `GuestResponse` object created by the model binder as the view model. Add a Razor View named `Thanks.cshtml` to the `Views/Home` folder with the content shown in Listing 3-12 to present a response to the user.

Listing 3-12. The Contents of the `Thanks.cshtml` File in the `Views/Home` Folder

```
@model PartyInvites.Models.GuestResponse
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Thanks</title>
</head>
<body>
    <div>
        <h1>Thank you, @Model.Name!</h1>
        @if (Model.WillAttend == true) {
            @:It's great that you're coming. The drinks are already in the fridge!
        } else {
```

```

        @:Sorry to hear that you can't make it, but thanks for letting us know.
    }
</div>
Click <a asp-action="ListResponses">here</a> to see who is coming.
</body>
</html>

```

The HTML produced by the `Thanks.cshtml` view depends on the values assigned to the `GuestResponse` view model provided by the `RsvpForm` action method. To access the value of a property in the domain object, I use an `@Model.<PropertyName>` expression. So, for example, to get the value of the `Name` property, I use the `@Model.Name` expression. Don't worry if the Razor syntax doesn't make sense—I explain it in more detail in Chapter 21.

Now that I have created the `Thanks` view, I have a basic working example of handling a form. Start the application, click the `RSVP Now` link, add some data to the form, and click the `Submit RSVP` button. You will see the response shown in Figure 3-5 (although it will differ if your name is not Joe or you said you could not attend).

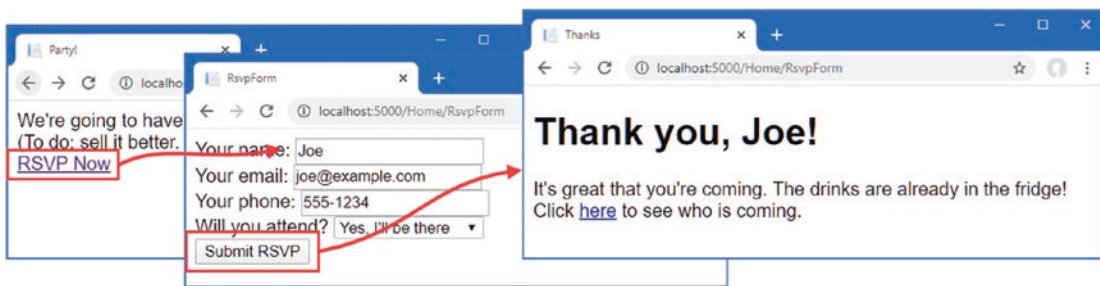


Figure 3-5. The `Thanks` view

Displaying the Responses

At the end of the `Thanks.cshtml` view, I added an `a` element to create a link to display the list of people who are coming to the party. I used the `asp-action` tag helper attribute to create a URL that targets an action method called `ListResponses`, like this:

```

...
<div>Click <a asp-action="ListResponses">here</a> to see who is coming.</div>
...

```

If you hover the mouse over the link that is displayed by the browser, you will see that it targets the `/Home/ListResponses` URL. This doesn't correspond to any of the action methods in the `Home` controller, and if you click the link, you will see a 404 Not Found error response.

To add an endpoint that will handle the URL, I need to add another action method to the `Home` controller, as shown in Listing 3-13.

Listing 3-13. Adding an Action Method in the `HomeController.cs` File in the `Controllers` Folder

```

using Microsoft.AspNetCore.Mvc;
using PartyInvites.Models;
using System.Linq;

namespace PartyInvites.Controllers {
    public class HomeController : Controller {

        public IActionResult Index() {
            return View();
        }
    }
}

```

```

[HttpGet]
public IActionResult RsvpForm() {
    return View();
}

[HttpPost]
public IActionResult RsvpForm(GuestResponse guestResponse) {
    Repository.AddResponse(guestResponse);
    return View("Thanks", guestResponse);
}

public IActionResult ListResponses() {
    return View(Repository.Responses.Where(r => r.WillAttend == true));
}
}

```

The new action method is called `ListResponses`, and it calls the `View` method, using the `Repository.Responses` property as the argument. This will cause Razor to render the default view, using the action method name as the name of the view file, and to use the data from the repository as the view model. The view model data is filtered using LINQ so that only positive responses are provided to the view.

Add a Razor View named `ListResponses.cshtml` to the `Views/Home` folder with the content shown in Listing 3-14.

Listing 3-14. Displaying Acceptances in the `ListResponses.cshtml` File in the `Views/Home` Folder

```

@model IEnumerable<PartyInvites.Models.GuestResponse>
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Responses</title>
</head>
<body>
    <h2>Here is the list of people attending the party</h2>
    <table>
        <thead>
            <tr><th>Name</th><th>Email</th><th>Phone</th></tr>
        </thead>
        <tbody>
            @foreach (PartyInvites.Models.GuestResponse r in Model) {
                <tr>
                    <td>@r.Name</td>
                    <td>@r.Email</td>
                    <td>@r.Phone</td>
                </tr>
            }
        </tbody>
    </table>
</body>
</html>

```


Razor view files have the `.cshtml` file extension because they are a mix of C# code and HTML elements. You can see this in Listing 3-14 where I have used a `@foreach` expression to process each of the `GuestResponse` objects that the action method passes to the view using the `View` method. Unlike a normal C# `foreach` loop, the body of a Razor `@foreach` expression contains HTML elements that are added to the response that will be sent back to the browser. In this view, each `GuestResponse` object generates a `tr` element that contains `td` elements populated with the value of an object property.

Start the application, submit some form data, and click the link to see the list of responses. You will see a summary of the data you have entered since the application was started, as shown in Figure 3-6. The view does not present the data in an appealing way, but it is enough for the moment, and I will address the styling of the application later in this chapter.

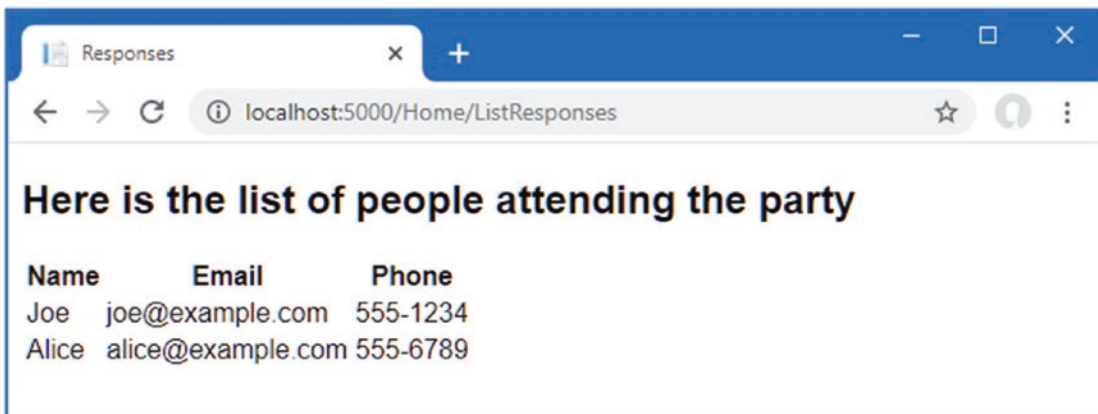


Figure 3-6. Showing a list of party attendees

Adding Validation

I can now add data validation to the application. Without validation, users could enter nonsense data or even submit an empty form. In an ASP.NET Core application, validation rules are defined by applying attributes to model classes, which means the same validation rules can be applied in any form that uses that class. ASP.NET Core relies on attributes from the `System.ComponentModel.DataAnnotations` namespace, which I have applied to the `GuestResponse` class in Listing 3-15.

Listing 3-15. Applying Validation in the `GuestResponse.cs` File in the Models Folder

```
using System.ComponentModel.DataAnnotations;
```

```
namespace PartyInvites.Models {
    public class GuestResponse {
        [Required(ErrorMessage = "Please enter your name")]
        public string Name { get; set; }

        [Required(ErrorMessage = "Please enter your email address")]
        [EmailAddress]
        public string Email { get; set; }

        [Required(ErrorMessage = "Please enter your phone number")]
        public string Phone { get; set; }

        [Required(ErrorMessage = "Please specify whether you'll attend")]
        public bool? WillAttend { get; set; }
    }
}
```

ASP.NET Core detects the attributes and uses them to validate data during the model-binding process.

■ **Tip** As noted earlier, I used a nullable `bool` for the `WillAttend` property. I did this so that I could apply the `Required` validation attribute. If I had used a regular `bool`, the value I received through model binding could be only `true` or `false`, and I would not be able to tell whether the user had selected a value. A nullable `bool` has three possible values: `true`, `false`, and `null`. The browser sends a `null` value if the user has not selected a value, and this causes the `Required` attribute to report a validation error. This is a nice example of how ASP.NET Core elegantly blends C# features with HTML and HTTP.

I check to see whether there has been a validation problem using the `ModelState.IsValid` property in the action method that receives the form data, as shown in Listing 3-16.

Listing 3-16. Checking for Validation Errors in the `HomeController.cs` File in the `Controllers` Folder

```
using Microsoft.AspNetCore.Mvc;
using PartyInvites.Models;
using System.Linq;

namespace PartyInvites.Controllers {
    public class HomeController : Controller {

        public IActionResult Index() {
            return View();
        }

        [HttpGet]
        public IActionResult RsvpForm() {
            return View();
        }

        [HttpPost]
        public IActionResult RsvpForm(GuestResponse guestResponse) {
            if (ModelState.IsValid) {
                Repository.AddResponse(guestResponse);
                return View("Thanks", guestResponse);
            } else {
                return View();
            }
        }

        public IActionResult ListResponses() {
            return View(Repository.Responses.Where(r => r.WillAttend == true));
        }
    }
}
```

The `Controller` base class provides a property called `ModelState` that provides details of the outcome of the model binding process. If the `ModelState.IsValid` property returns `true`, then I know that the model binder has been able to satisfy the validation constraints I specified through the attributes on the `GuestResponse` class. When this happens, I render the `Thanks` view, just as I did previously.

If the `ModelState.IsValid` property returns `false`, then I know that there are validation errors. The object returned by the `ModelState` property provides details of each problem that has been encountered, but I don't need to get into that level of detail because I can rely on a useful feature that automates the process of asking the user to address any problems by calling the `View` method without any parameters.

When it renders a view, Razor has access to the details of any validation errors associated with the request, and tag helpers can access the details to display validation errors to the user. Listing 3-17 shows the addition of validation tag helper attributes to the RsvpForm view.

Listing 3-17. Adding a Validation Summary to the RsvpForm.cshtml File in the Views/Home Folder

```
@model PartyInvites.Models.GuestResponse
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>RsvpForm</title>
</head>
<body>
    <form asp-action="RsvpForm" method="post">
        <div asp-validation-summary="All"></div>
        <div>
            <label asp-for="Name">Your name:</label>
            <input asp-for="Name" />
        </div>
        <div>
            <label asp-for="Email">Your email:</label>
            <input asp-for="Email" />
        </div>
        <div>
            <label asp-for="Phone">Your phone:</label>
            <input asp-for="Phone" />
        </div>
        <div>
            <label>Will you attend?</label>
            <select asp-for="WillAttend">
                <option value="">Choose an option</option>
                <option value="true">Yes, I'll be there</option>
                <option value="false">No, I can't come</option>
            </select>
        </div>
        <button type="submit">Submit RSVP</button>
    </form>
</body>
</html>
```

The `asp-validation-summary` attribute is applied to a `div` element, and it displays a list of validation errors when the view is rendered. The value for the `asp-validation-summary` attribute is a value from an enumeration called `ValidationSummary`, which specifies what types of validation errors the summary will contain. I specified `All`, which is a good starting point for most applications, and I describe the other values and explain how they work in Chapter 29.

To see how the validation summary works, run the application, fill out the Name field, and submit the form without entering any other data. You will see a summary of validation errors, as shown in Figure 3-7.

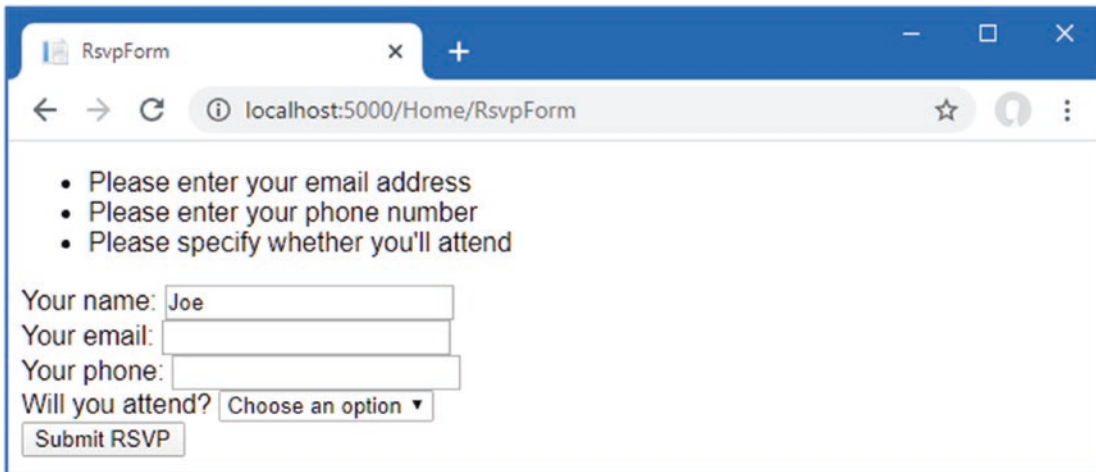


Figure 3-7. Displaying validation errors

The `RsvpForm` action method will not render the `Thanks` view until all the validation constraints applied to the `GuestResponse` class have been satisfied. Notice that the data entered in the `Name` field was preserved and displayed again when Razor rendered the view with the validation summary. This is another benefit of model binding, and it simplifies working with form data.

Highlighting Invalid Fields

The tag helper attributes that associate model properties with elements have a handy feature that can be used in conjunction with model binding. When a model class property has failed validation, the helper attributes will generate slightly different HTML. Here is the `input` element that is generated for the `Phone` field when there is no validation error:

```
<input type="text" data-val="true" data-val-required="Please enter your phone number" id="Phone" name="Phone" value="">
```

For comparison, here is the same HTML element after the user has submitted the form without entering data into the text field (which is a validation error because I applied the `Required` attribute to the `Phone` property of the `GuestResponse` class):

```
<input type="text" class="input-validation-error" data-val="true" data-val-required="Please enter your phone number" id="Phone" name="Phone" value="">
```

I have highlighted the difference: the `asp-for` tag helper attribute added the `input` element to a class called `input-validation-error`. I can take advantage of this feature by creating a stylesheet that contains CSS styles for this class and the others that different HTML helper attributes use.

The convention in ASP.NET Core projects is that static content delivered to clients is placed into the `wwwroot` folder and organized by content type so that CSS stylesheets go into the `wwwroot/css` folder, JavaScript files go into the `wwwroot/js` folder, and so on.

■ **Tip** Visual Studio creates a `site.css` file in the `wwwroot/css` folder when a project is created using the Web Application template. You can ignore this file, which I don't use in this chapter.

If you are using Visual Studio, right-click the `wwwroot/css` folder and select `Add ► New Item` from the popup menu. Locate the `Style Sheet` item template, as shown in Figure 3-8; set the name of the file to `styles.css`; and click the `Add` button.

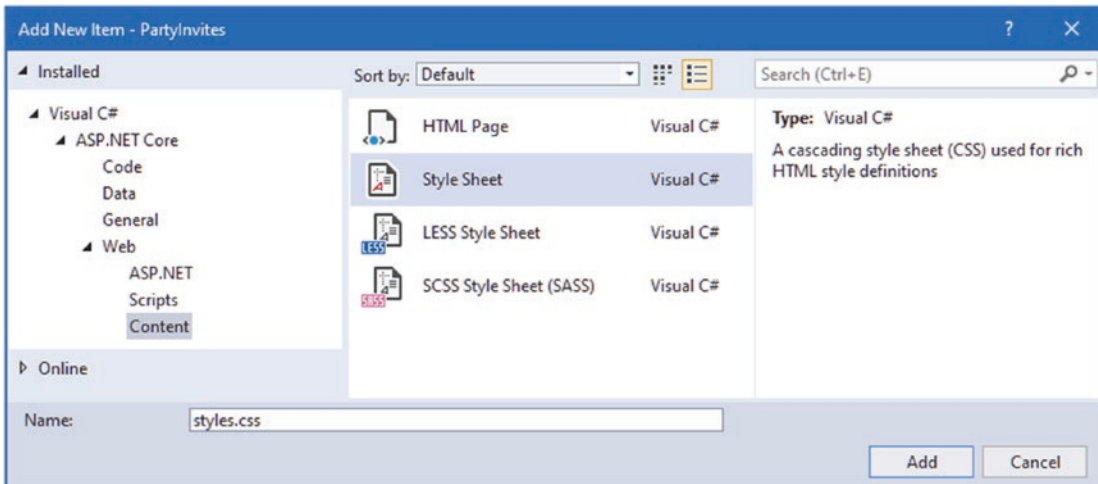


Figure 3-8. Creating a CSS stylesheet

If you are using Visual Studio Code, right-click the `wwwroot/css` folder, select New File from the popup menu, and use `styles.css` as the file name. Regardless of which editor you use, replace the contents of the file with the styles shown in Listing 3-18.

Listing 3-18. The Contents of the `styles.css` File in the `wwwroot/css` Folder

```
.field-validation-error    {color: #f00;}
.field-validation-valid   { display: none;}
.input-validation-error   { border: 1px solid #f00; background-color: #fee; }
.validation-summary-errors { font-weight: bold; color: #f00;}
.validation-summary-valid { display: none;}
```

To apply this stylesheet, I added a link element to the head section of the `RsvpForm` view, as shown in Listing 3-19.

Listing 3-19. Applying a Stylesheet in the `RsvpForm.cshtml` File in the `Views/Home` Folder

```
...
<head>
  <meta name="viewport" content="width=device-width" />
  <title>RsvpForm</title>
  <link rel="stylesheet" href="/css/styles.css" />
</head>
...
```

The link element uses the `href` attribute to specify the location of the stylesheet. Notice that the `wwwroot` folder is omitted from the URL. The default configuration for ASP.NET includes support for serving static content, such as images, CSS stylesheets, and JavaScript files, and it maps requests to the `wwwroot` folder automatically. With the application of the stylesheet, a more obvious validation error will be displayed when data is submitted that causes a validation error, as shown in Figure 3-9.

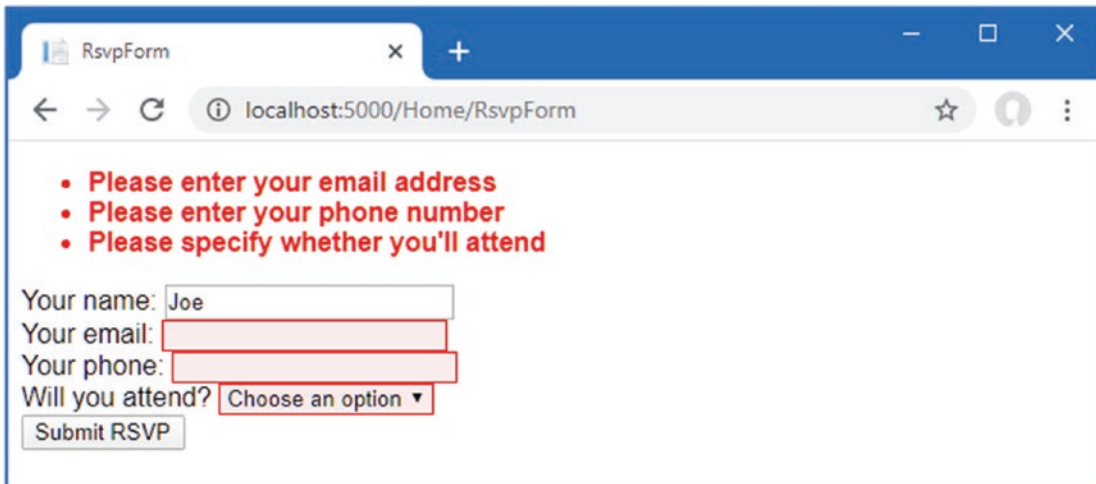


Figure 3-9. Automatically highlighted validation errors

Styling the Content

All the functional goals for the application are complete, but the overall appearance of the application is poor. When you create a project using the `mvc` template, as I did for the example in this chapter, some common client-side development packages are installed. While I am not a fan of using template projects, I do like the client-side libraries that Microsoft has chosen. One of them is called Bootstrap, which is a good CSS framework originally developed by Twitter that has become a major open source project and a mainstay of web application development.

Styling the Welcome View

The basic Bootstrap features work by applying classes to elements that correspond to CSS selectors defined in the files added to the `wwwroot/lib/bootstrap` folder. You can get full details of the classes that Bootstrap defines from <http://getbootstrap.com>, but you can see how I have applied some basic styling to the `Index.cshtml` view file in Listing 3-20.

Listing 3-20. Adding Bootstrap to the `Index.cshtml` File in the `Views/Home` Folder

```
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <link rel="stylesheet" href="/lib/bootstrap/dist/css/bootstrap.css" />
    <title>Index</title>
</head>
<body>
    <div class="text-center">
        <h3> We're going to have an exciting party!</h3>
        <h4>And YOU are invited!</h4>
        <a class="btn btn-primary" asp-action="RsvpForm">RSVP Now</a>
    </div>
</body>
</html>
```

I have added a link element whose href attribute loads the bootstrap.css file from the wwwroot/lib/bootstrap/dist/css folder. The convention is that third-party CSS and JavaScript packages are installed into the wwwroot/lib folder, and I describe the tool that is used to manage these packages in Chapter 12.

Having imported the Bootstrap stylesheets, I need to style my elements. This is a simple example, so I need to use only a small number of Bootstrap CSS classes: text-center, btn, and btn-primary.

The text-center class centers the content of an element and its children. The btn class styles a button, input, or a element as a pretty button, and the btn-primary class specifies which of a range of colors I want the button to be. You can see the effect by running the application, as shown in Figure 3-10.

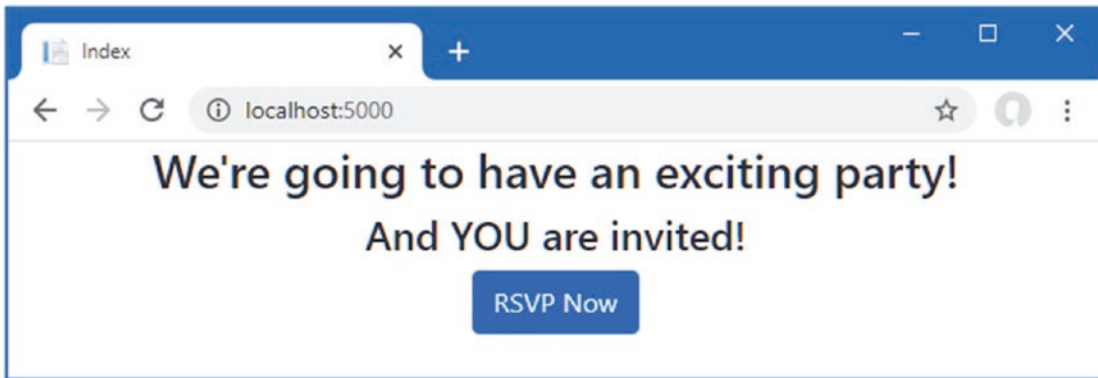


Figure 3-10. Styling a view

It will be obvious to you that I am not a web designer. In fact, as a child, I was excused from art lessons on the basis that I had absolutely no talent whatsoever. This had the happy result of making more time for math lessons but meant that my artistic skills have not developed beyond those of the average 10-year-old. For a real project, I would seek a professional to help design and style the content, but for this example, I am going it alone, and that means applying Bootstrap with as much restraint and consistency as I can muster.

Styling the Form View

Bootstrap defines classes that can be used to style forms. I am not going to go into detail, but you can see how I have applied these classes in Listing 3-21.

Listing 3-21. Adding Bootstrap to the RsvpForm.cshtml File in the Views/Home Folder

```
@model PartyInvites.Models.GuestResponse
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>RsvpForm</title>
    <link rel="stylesheet" href="/css/styles.css" />
    <link rel="stylesheet" href="/lib/bootstrap/dist/css/bootstrap.css" />
</head>
<body>
    <h5 class="bg-primary text-white text-center m-2 p-2">RSVP</h5>
    <form asp-action="RsvpForm" method="post" class="m-2">
        <div asp-validation-summary="All"></div>
```

```

<div class="form-group">
  <label asp-for="Name">Your name:</label>
  <input asp-for="Name" class="form-control" />
</div>
<div class="form-group">
  <label asp-for="Email">Your email:</label>
  <input asp-for="Email" class="form-control" />
</div>
<div class="form-group">
  <label asp-for="Phone">Your phone:</label>
  <input asp-for="Phone" class="form-control" />
</div>
<div class="form-group">
  <label>Will you attend?</label>
  <select asp-for="WillAttend" class="form-control">
    <option value="">Choose an option</option>
    <option value="true">Yes, I'll be there</option>
    <option value="false">No, I can't come</option>
  </select>
</div>
<button type="submit" class="btn btn-primary">Submit RSVP</button>
</form>
</body>
</html>

```

The Bootstrap classes in this example create a header, just to give structure to the layout. To style the form, I have used the `form-group` class, which is used to style the element that contains the label and the associated input or select element, which is assigned to the `form-control` class. You can see the effect of the styles in Figure 3-11.

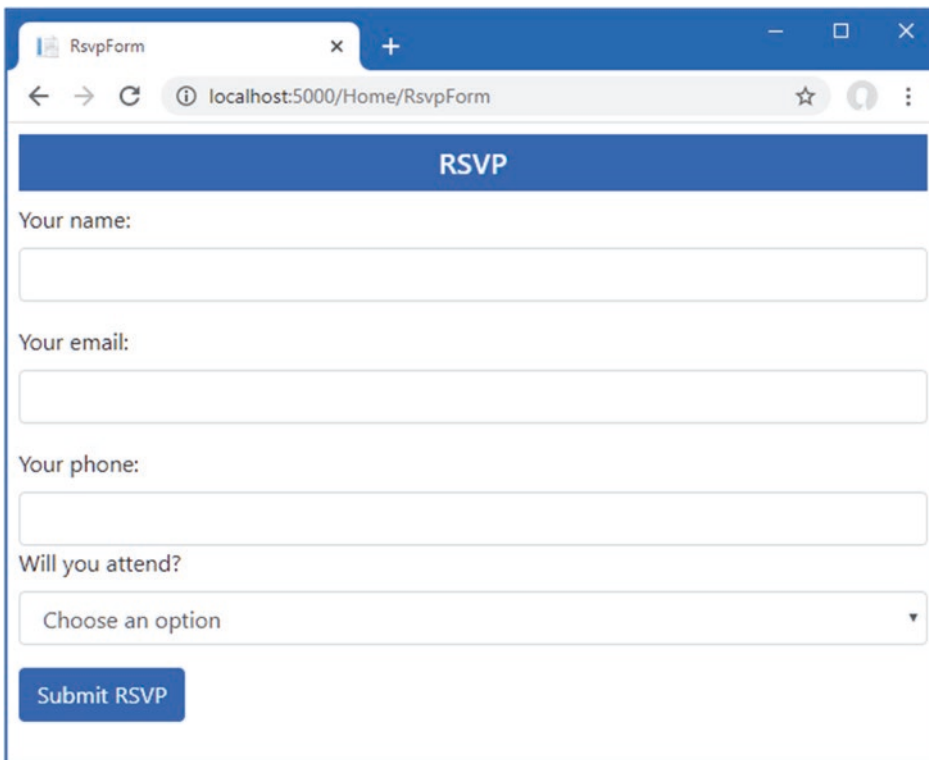


Figure 3-11. Styling the `RsvpForm` view

Styling the Thanks View

The next view file to style is `Thanks.cshtml`, and you can see how I have done this in Listing 3-22, using CSS classes that are similar to the ones I used for the other views. To make an application easier to manage, it is a good principle to avoid duplicating code and markup wherever possible. ASP.NET Core provides several features to help reduce duplication, which I describe in later chapters. These features include Razor layouts (Chapter 22), partial views (Chapter 22), and view components (Chapter 24).

Listing 3-22. Applying Bootstrap to the `Thanks.cshtml` File in the `Views/Home` Folder

```
@model PartyInvites.Models.GuestResponse
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Thanks</title>
    <link rel="stylesheet" href="/lib/bootstrap/dist/css/bootstrap.css" />
</head>
<body class="text-center">
    <div>
        <h1>Thank you, @Model.Name!</h1>
        @if (Model.WillAttend == true) {
            @:It's great that you're coming. The drinks are already in the fridge!
        } else {
            @:Sorry to hear that you can't make it, but thanks for letting us know.
        }
    </div>
    Click <a asp-action="ListResponses">here</a> to see who is coming.</div>
</body>
</html>
```

Figure 3-12 shows the effect of the styles.

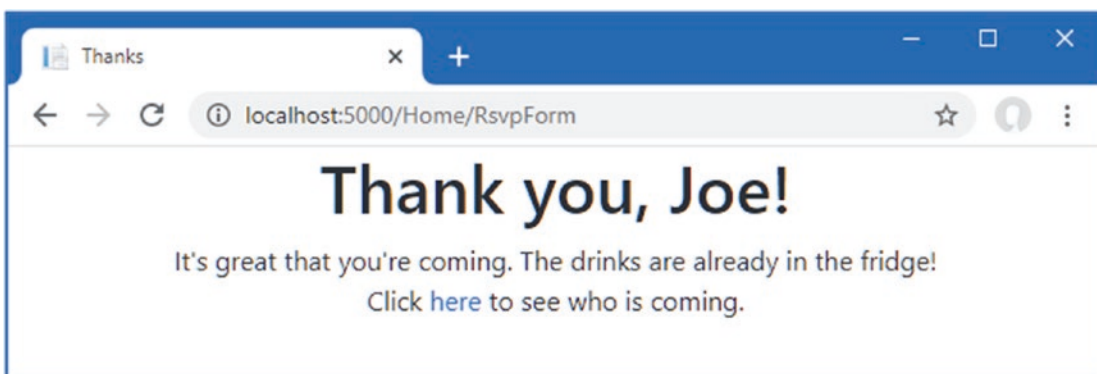


Figure 3-12. Styling the `Thanks` view

Styling the List View

The final view to style is `ListResponses`, which presents the list of attendees. Styling the content follows the same approach as used for the other views, as shown in Listing 3-23.

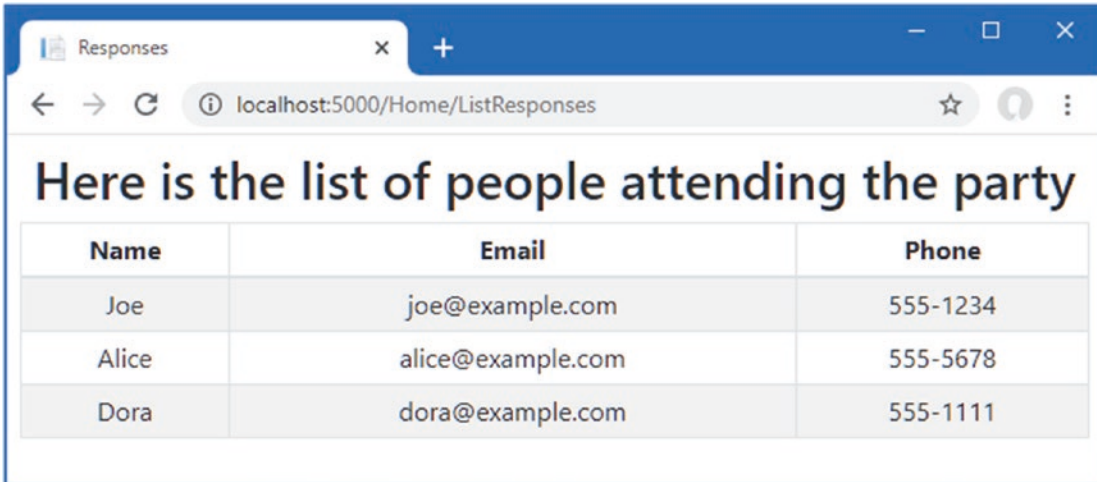
Listing 3-23. Adding Bootstrap to the `ListResponses.cshtml` File in the `Views/Home` Folder

```
@model IEnumerable<PartyInvites.Models.GuestResponse>
@{
    Layout = null;
}

<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Responses</title>
    <link rel="stylesheet" href="/lib/bootstrap/dist/css/bootstrap.css" />
</head>
<body>
    <div class="text-center p-2">
        <h2>Here is the list of people attending the party</h2>
        <table class="table table-bordered table-striped table-sm">
            <thead>
                <tr><th>Name</th><th>Email</th><th>Phone</th></tr>
            </thead>
            <tbody>
                @foreach (PartyInvites.Models.GuestResponse r in Model) {
                    <tr>
                        <td>@r.Name</td>
                        <td>@r.Email</td>
                        <td>@r.Phone</td>
                    </tr>
                }
            </tbody>
        </table>
    </div>
</body>
</html>
```

Figure 3-13 shows the way that the table of attendees is presented. Adding these styles to the view completes the example application, which now meets all the development goals and has an improved appearance.

A screenshot of a web browser window. The browser has a single tab titled "Responses". The address bar shows "localhost:5000/Home/ListResponses". The main content of the page is a heading "Here is the list of people attending the party" followed by a table with three columns: "Name", "Email", and "Phone". The table contains three rows of data: Joe (joe@example.com, 555-1234), Alice (alice@example.com, 555-5678), and Dora (dora@example.com, 555-1111).

Name	Email	Phone
Joe	joe@example.com	555-1234
Alice	alice@example.com	555-5678
Dora	dora@example.com	555-1111

Figure 3-13. Styling the ListResponses view

Summary

In this chapter, I created a new ASP.NET Core project and used it to construct a simple data-entry application, giving you a first glimpse of important ASP.NET features, such as tag helpers, model binding, and data validation. In the next chapter, I describe the development tools that are used for ASP.NET Core development.

CHAPTER 4



Using the Development Tools

In this chapter, I introduce the tools that Microsoft provides for ASP.NET Core development and that are used throughout this book.

Unlike earlier editions of this book, I rely on the command-line tools provided by the .NET Core SDK and additional tool packages that Microsoft publishes. In part, I have done this to help ensure you get the expected results from the examples but also because the command-line tools provide access to all the features required for ASP.NET Core development, regardless of which editor/IDE you have chosen.

Visual Studio—and, to a lesser extent, Visual Studio Code—offers access to some of the tools through user interfaces, which I describe in this chapter, but Visual Studio and Visual Studio Code don't support all the features that are required for ASP.NET Core development, so there are times that using the command line is inevitable.

As ASP.NET Core has evolved, I have gradually moved to using just the command-line tools, except for when I need to use a debugger (although, as I explain later in the chapter, this is a rare requirement). Your preferences may differ, especially if you are used to working entirely within an IDE, but my suggestion is to give the command-line tools a go. They are simple, concise, and predictable, which cannot be said for all the equivalent functionality provided by Visual Studio and Visual Studio Code. Table 4-1 summarizes the chapter.

Table 4-1. Chapter Summary

Problem	Solution	Listing
Creating a project	Use the <code>dotnet new</code> commands or the Visual Studio wizard	1-4
Building and running projects	Use the <code>dotnet build</code> and <code>dotnet run</code> commands or use the menus provided by Visual Studio and Visual Studio Code	5-7
Adding packages to a project	Use the <code>dotnet add package</code> command or use the Visual Studio package manager	8-10
Installing tool commands	Use the <code>dotnet tool</code> command	11, 12
Managing client-side packages	Use the <code>libman</code> command or the Visual Studio client-side package manager	13-16

Creating ASP.NET Core Projects

The .NET Core SDK includes a set of command-line tools for creating, managing, building, and running projects. Visual Studio provides integrated support for some of these tasks, but if you are using Visual Studio Code, then the command line is the only option.

I use the command-line tools even when working with Visual Studio because they are simple and concise, while the Visual Studio features tend to require more effort to locate the templates or settings I require. In the sections that follow, I show you how to create and use both sets of tools. The results are the same whichever approach you choose, and you can freely switch between Visual Studio and the command-line tools.

■ **Tip** You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/apress/pro-asp.net-core-3>. See Chapter 1 for how to get help if you have problems running the examples.

Creating a Project Using the Command Line

The `dotnet` command provides access to the .NET Core command-line features. The `dotnet new` command is used to create a new project, configuration file, or solution file. To see the list of templates available for creating new items, open a PowerShell command prompt and run the command shown in Listing 4-1.

Listing 4-1. Listing the .NET Core Templates

```
dotnet new
```

Each template has a short name that makes it easier to use. There are a lot of templates available, but Table 4-2 describes the ones that are most useful for creating ASP.NET Core projects.

Table 4-2. Useful ASP.NET Core Project Templates

Name	Description
<code>web</code>	This template creates a project that is set up with the minimum code and content required for ASP.NET Core development. This is the template I use for most of the chapters in this book.
<code>mvc</code>	This template creates an ASP.NET Core project configured to use the MVC Framework.
<code>webapp</code>	This template creates an ASP.NET Core project configured to use Razor Pages.
<code>blazorserver</code>	This template creates an ASP.NET Core project configured to use Blazor Server.
<code>angular</code>	This template creates an ASP.NET Core project that contains client-side features using the Angular JavaScript framework.
<code>react</code>	This template creates an ASP.NET Core project that contains client-side features using the React JavaScript framework.
<code>reactredux</code>	This template creates an ASP.NET Core project that contains client-side features using the React JavaScript framework and the popular Redux library.

There are also templates that create commonly required files used to configure projects, as described in Table 4-3.

UNDERSTANDING THE LIMITATIONS OF PROJECT TEMPLATES

The project templates described in Table 4-2 are intended to help jump-start development by taking care of basic configuration settings and adding placeholder content.

These templates can give you a sense of rapid progress, but they contain assumptions about how a project should be configured and developed. If you don't understand the impact of those assumptions, you won't be able to get the results you require for the specific demands of your project.

The `web` template creates a project with the minimum configuration required for ASP.NET Core development. This is the project template I use for most of the examples in this book so that I can explain how each feature is configured and how the features can be used together.

Once you understand how ASP.NET Core works, the other project templates can be useful because you will know how to adapt them to your needs. But, while you are learning, I recommend sticking to the `web` template, even though it can take a little more effort to get results.

Table 4-3. The Configuration Item Templates

Name	Description
globaljson	This template adds a <code>global.json</code> file to a project, specifying the version of .NET Core that will be used.
sln	This template creates a solution file, which is used to group multiple projects and is commonly used by Visual Studio. The solution file is populated with the <code>dotnet sln add</code> command, as shown in the following listing.
gitignore	This template creates a <code>.gitignore</code> file that excludes unwanted items from Git source control.

To create a project, open a new PowerShell command prompt and run the commands shown in Listing 4-2.

Listing 4-2. Creating a New Project

```
dotnet new globaljson --sdk-version 3.1.101 --output MySolution/MyProject
dotnet new web --no-https --output MySolution/MyProject --framework netcoreapp3.1
dotnet new sln -o MySolution

dotnet sln MySolution add MySolution/MyProject
```

The first command creates a `MySolution/MyProject` folder that contains a `global.json` file, which specifies that the project will use .NET Core version 3.1.1. The top-level folder, named `MySolution`, is used to group multiple projects together. The nested `MyProject` folder will contain a single project.

I use the `global.json` template to help ensure you get the expected results when following the examples in this book. Microsoft is good at ensuring backward compatibility with .NET Core releases, but breaking changes do occur, and it is a good idea to add a `global.json` file to projects so that everyone in the development team is using the same version.

The second command creates the project using the web template, which I use for most of the examples in this book. As noted in Table 4-3, this template creates a project with the minimum content required for ASP.NET Core development. Each template has its own set of arguments that influence the project that is created. The `--no-https` argument creates a project without support for HTTPS. (I explain how to use HTTPS in Chapter 16.) The `--framework` argument selects the .NET Core runtime that will be used for the project.

The other commands create a solution file that references the new project. Solution files are a convenient way of opening multiple related files at the same time. A `MySolution.sln` file is created in the `MySolution` folder, and opening this file in Visual Studio will load the project created with the web template. This is not essential, but it stops Visual Studio from prompting you to create the file when you exit the code editor.

Opening the Project

To open the project, start Visual Studio, select “Open a Project or Solution,” and open the `MySolution.sln` file in the `MySolution` folder. Visual Studio will open the solution file, discover the reference to the project that was added by the final command in Listing 4-2, and open the project as well.

Visual Studio Code works differently. Start Visual Studio Code, select **File** ► **Open Folder**, and navigate to the `MySolution` folder. Click **Select Folder**, and Visual Studio Code will open the project.

Although Visual Studio Code and Visual Studio are working with the same project, each displays the contents differently. Visual Studio Code shows you a simple list of files, ordered alphabetically, as shown on the left of Figure 4-1. Visual Studio hides some files and nests others within related file items, as shown on the right of Figure 4-1.

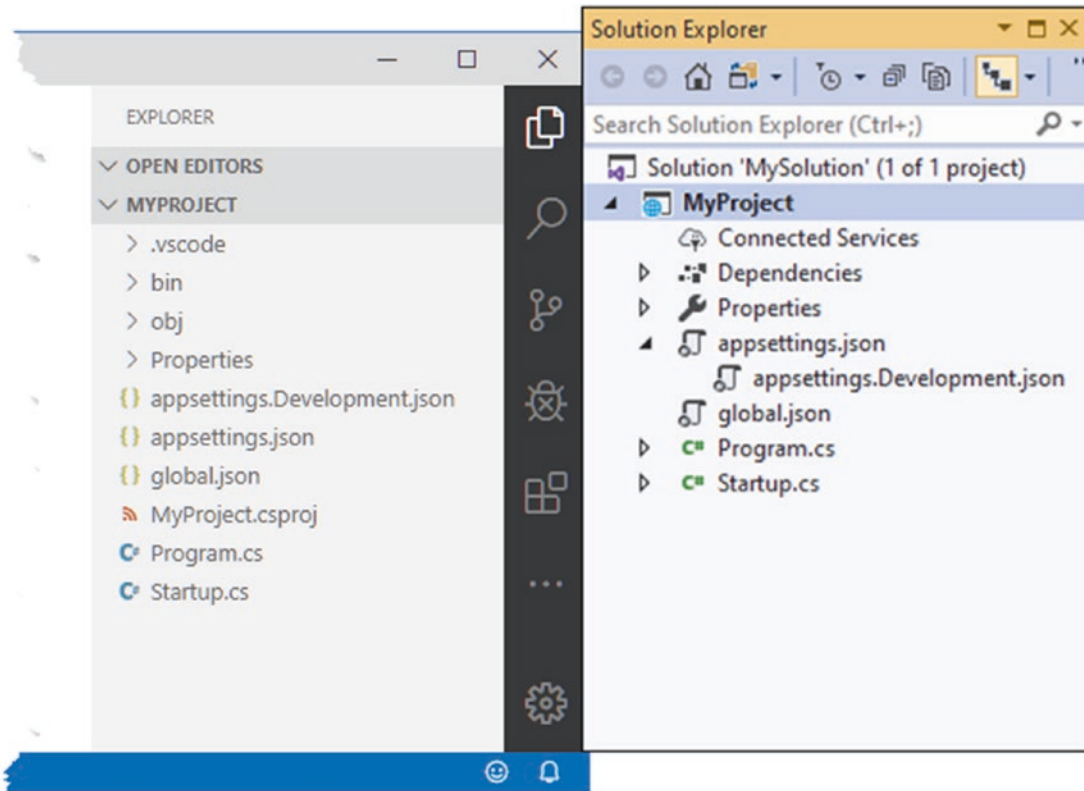


Figure 4-1. Opening a project in Visual Studio Code and Visual Studio

There are buttons at the top of the Visual Studio Solution Explorer that disable file nesting and show the hidden items in the project. When you open a project for the first time in Visual Studio Code, you may be prompted to add assets for building and debugging the project, as shown in Figure 4-2. Click the Yes button.

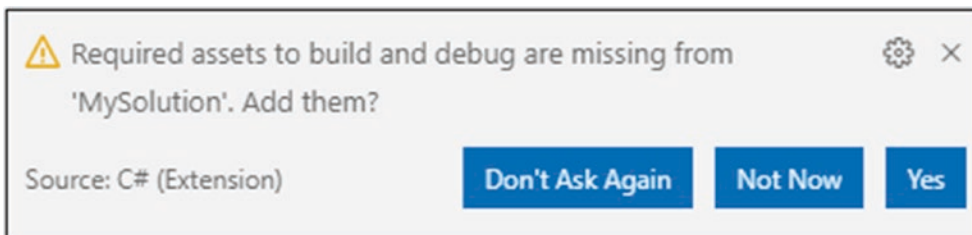


Figure 4-2. Adding assets in Visual Studio Code

Creating a Project Using Visual Studio

Visual Studio creates projects in the same way, using the same templates, but provides a wizard. Although my preference is to use the command line, the result is the same if you pay close attention to the options you choose along the way. Start Visual Studio and click “Create a New Project,” Then select the ASP.NET Core Web Application category, as shown in Figure 4-3, and click the Next button.

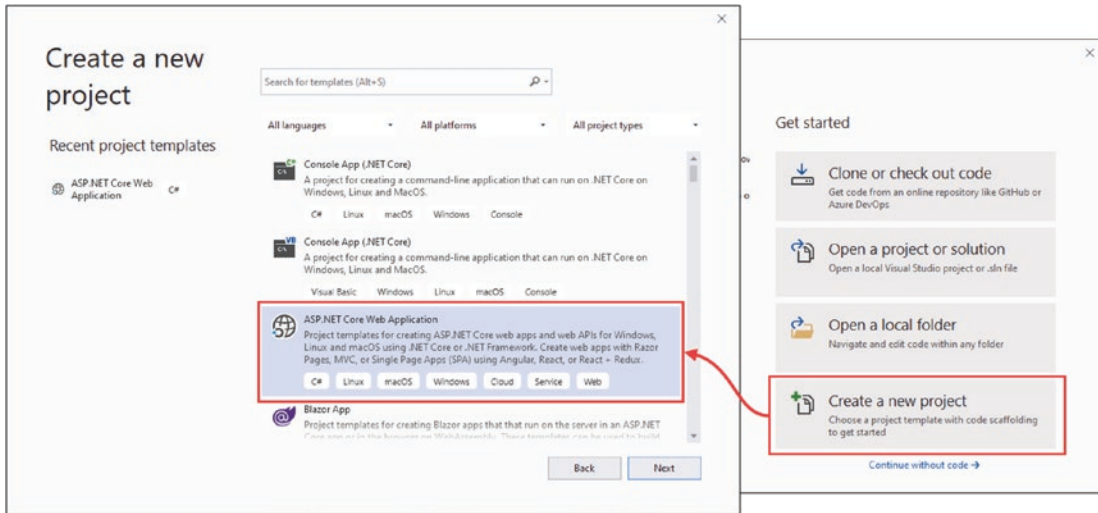


Figure 4-3. Creating a new project in Visual Studio

Enter **MyProject** in the “Project name” field and **MySolution** in the “Solution name” field, as shown in Figure 4-4. Use the Location field to select a convenient folder in which to create the project and click the Create button.

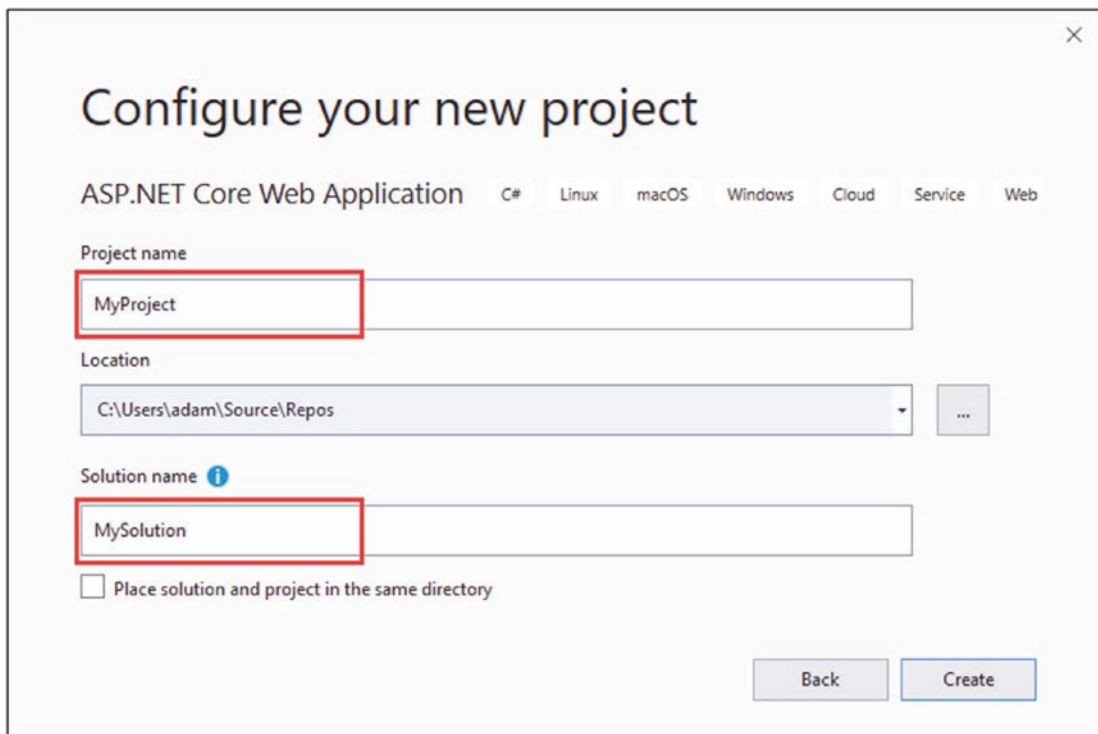


Figure 4-4. Selecting the project and solution names

The next step requires close attention, and because it is so prone to errors, I have used the command-line tools for the examples in this book. First, use the drop-downs at the top of the window to select .NET Core and ASP.NET Core 3.1. Next, select the Empty template from the list. Even though the name *Empty* is used, the content added to the project corresponds to the web template.

Uncheck the Configure for HTTPS option—which is equivalent to the `--no-https` command-line argument—and ensure the Enable Docker Support option is unchecked that the Authentication option is set to No Authentication, as shown in Figure 4-5.

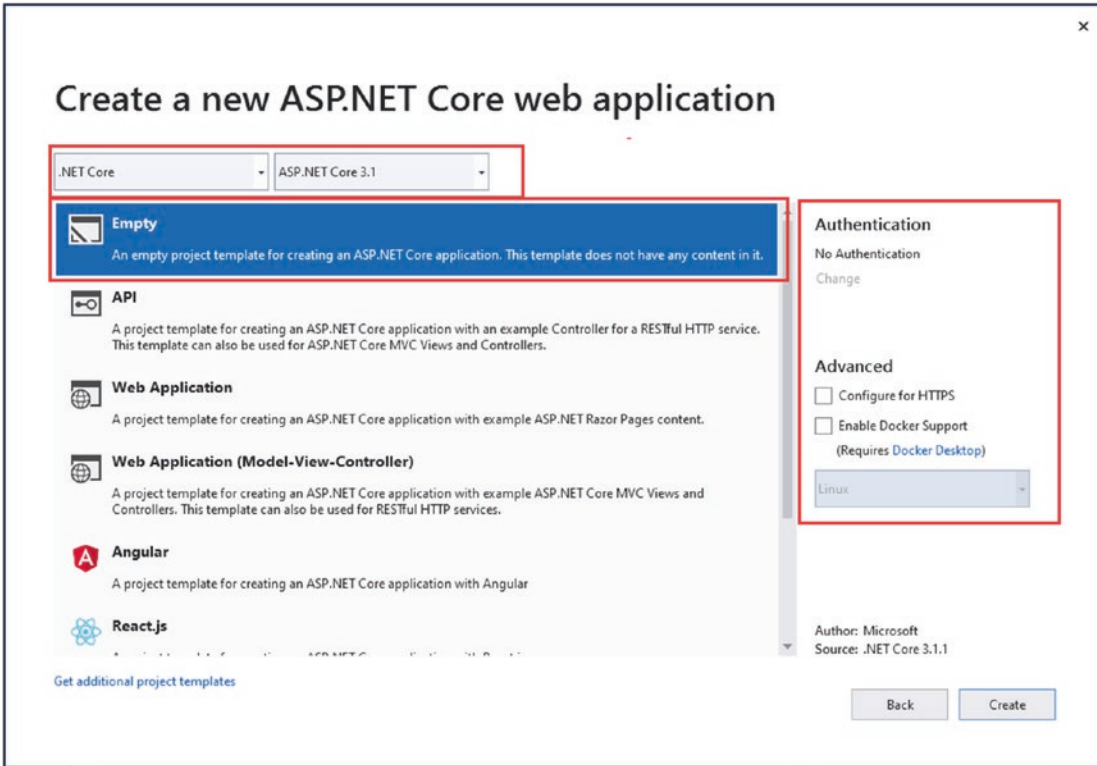


Figure 4-5. Selecting and configuring the project template

Click the Create button, and Visual Studio will create the project and the solution file and then open them for editing, as shown in Figure 4-6.

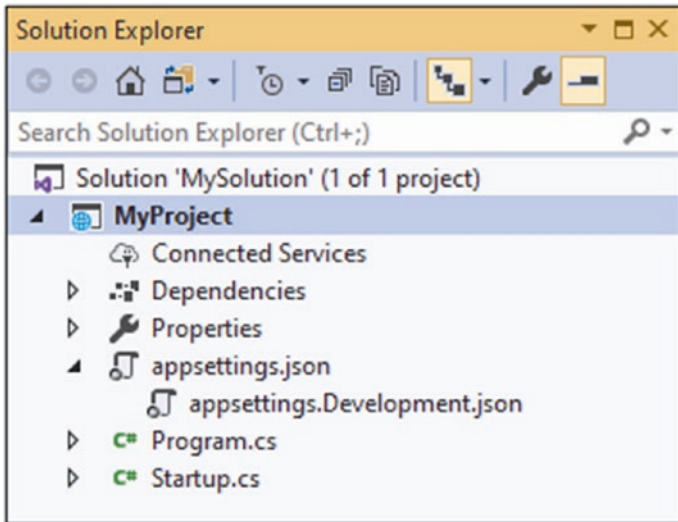


Figure 4-6. The new project in the Visual Studio Solution Explorer

Once the project has been opened, right-click the MyProject item in the Solution Explorer window and select Add ► New Item from the popup menu. Locate the JSON File item from the list of templates and set the Name field to global.json. Click the Add button to create the file and replace its contents with those shown in Listing 4-3.

Listing 4-3. The Contents of the `global.json` File in the `MyProject` Folder

```
{
  "sdk": {
    "version": "3.1.101"
  }
}
```

Adding the `global.json` file ensures the right version of the .NET Core SDK will be used by the project.

Adding Code and Content to Projects

If you are using Visual Studio Code, then you add items to the project by right-clicking the folder that should contain the file and selecting `New File` from the popup menu (or selecting `New Folder` if you are adding a folder).

■ **Note** You are responsible for ensuring that the file extension matches the type of item you want to add; for example, an HTML file must be added with the `.html` extension. I give the complete file name and the name of the containing folder for every item added to a project throughout this book, so you will always know exactly what files you need to add.

Right-click the `MyProject` item in the file explorer page, select `New Folder`, and set the name to `wwwroot`, which is where static content is stored in ASP.NET Core projects. Press `Enter`, and a folder named `wwwroot` will be added to the project. Right-click the new `wwwroot` folder, select `New Item`, and set the name to `demo.html`. Press `Enter` to create the HTML file and add the content shown in Listing 4-4.

Listing 4-4. The Contents of the `demo.html` File in the `wwwroot` Folder

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title></title>
</head>
<body>
  <h3>HTML File from MyProject</h3>
</body>
</html>
```

Visual Studio provides a more comprehensive approach that can be helpful but only when used selectively. To create a folder, right-click the `MyProject` item in the Solution Explorer and select `Add ► New Folder` from the popup menu. Set the name of the new item to `wwwroot` and press `Enter`; Visual Studio will create the folder.

Right-click the new `wwwroot` item in the Solution Explorer and select `Add ► New Item` from the popup menu. Visual Studio will present you with an extensive selection of templates for adding items to the project. These templates can be searched using the text field in the top-right corner of the window or filtered using the categories on the left of the window. The item template for an HTML file is named `HTML Page`, as shown in Figure 4-7.

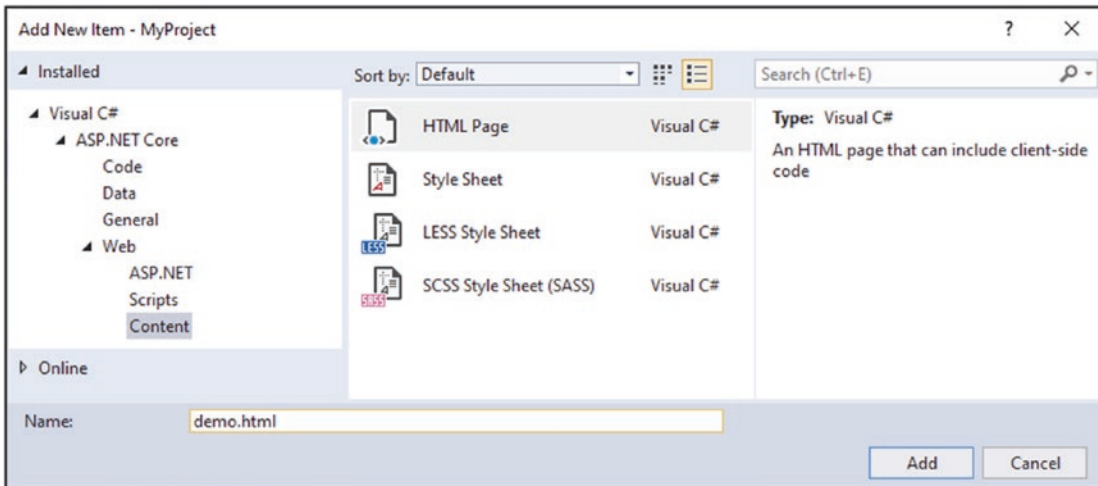


Figure 4-7. Adding an item to the example project

Enter **demo.html** in the Name field, click the Add button to create the new file, and replace the contents with the element shown in Listing 4-4. (If you omit the file extension, Visual Studio will add it for you based on the item template you have selected. If you entered just **demo** into the Name field when you created the file, Visual Studio would have created a file with the `.html` extension because you had selected the HTML Page item template.)

Understanding Item Scaffolding

The item templates presented by Visual Studio can be useful, especially for C# classes where it sets the namespace and class name automatically. But Visual Studio also provides *scaffolded items*, which I recommend against using. The Add ► New Scaffolded Item leads to a selection of items that guide you through a process to add more complex items. Visual Studio will also offer individual scaffolded items based on the name of the folder that you are adding an item to. For example, if you right-click a folder named Views, Visual Studio will helpfully add scaffolded items to the top of the menu, as shown in Figure 4-8.

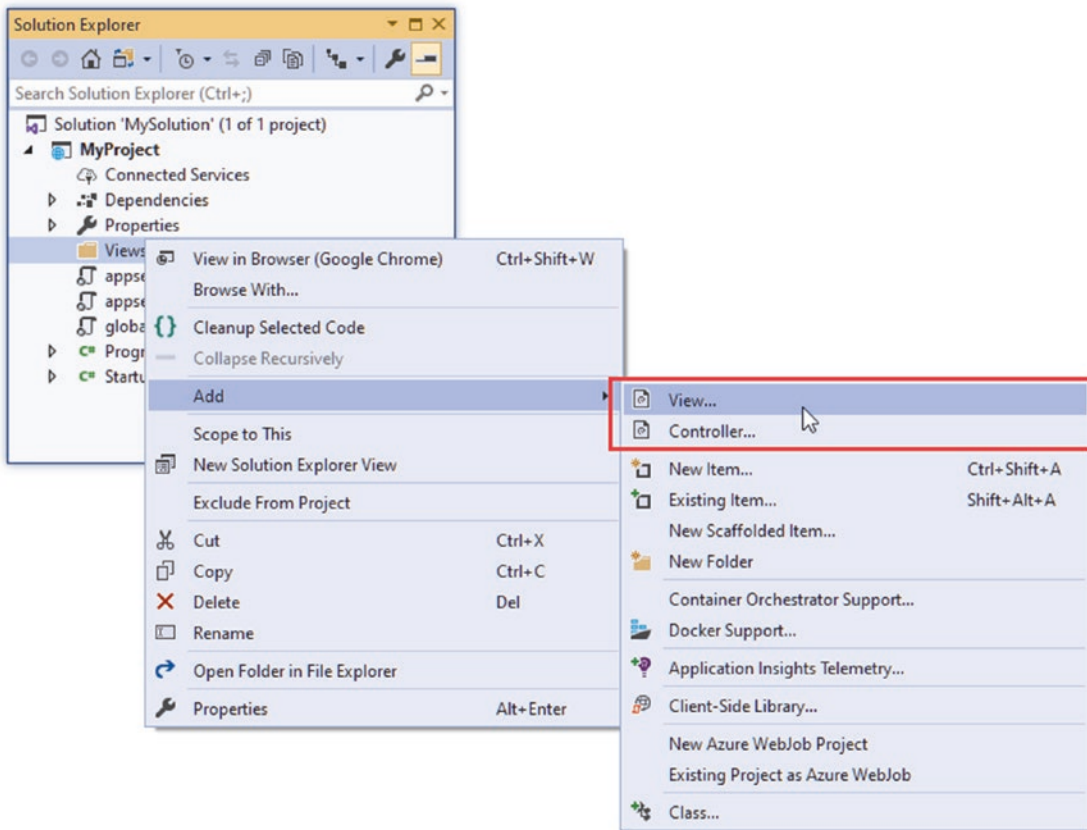


Figure 4-8. Scaffolded items in the Add menu

The View and Controller items are scaffolded, and selecting them will present you with choices that determine the content of the items you create.

Just like the project templates, I recommend against using scaffolded items, at least until you understand the content they create. In this book, I use only the Add ► New Item menu for the examples and change the placeholder content immediately.

Building and Running Projects

You can build and run projects from the command line or from within Visual Studio and Visual Studio Code. To prepare, add the statement shown in Listing 4-5 to the Startup.cs class file in the MyProject folder.

Listing 4-5. Adding a Statement in the Startup.cs File in the MyProject Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
```

```

namespace MyProject {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {

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

            app.UseStaticFiles();
            app.UseRouting();

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

```

This statement adds support for responding to HTTP requests with static content in the `wwwroot` folder, such as the HTML file created in the previous section. (I explain this feature in more detail in Chapter 15.)

Building and Running Projects Using the Command Line

To build the example project, run the command shown in Listing 4-6 in the `MyProject` or `MySolution` folder.

Listing 4-6. Building the Project

```
dotnet build
```

You can build and run the project in a single step by running the command shown in Listing 4-7 in the `MyProject` folder.

Listing 4-7. Building and Running the Project

```
dotnet run
```

The compiler will build the project and then start the integrated ASP.NET Core HTTP server to listen for HTTP requests on port 5000. You can see the contents of the static HTML file added to the project earlier in the chapter by opening a new browser window and requesting `http://localhost:5000/demo.html`, which produces the response shown in Figure 4-9.

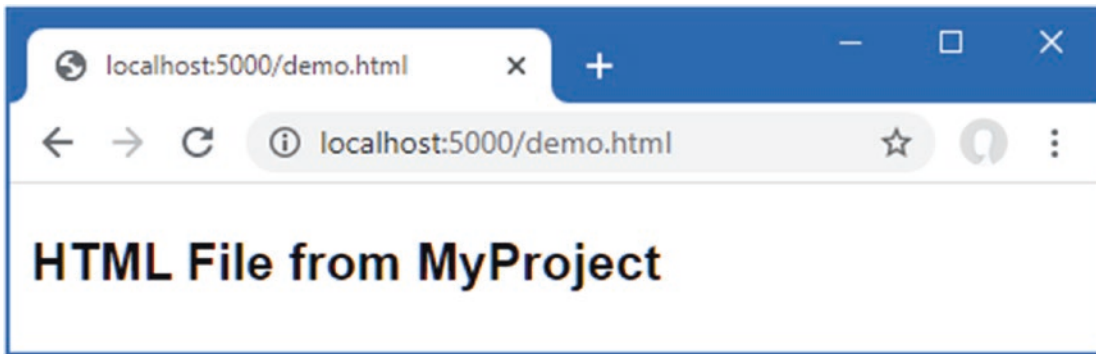


Figure 4-9. Running the example application

Building and Running Projects Using Visual Studio Code

Visual Studio Code can build and execute the project for you if you prefer not to use the command line. Select Terminal ► Run Build Task, and Visual Studio Code will compile the project.

To build and run the project in a single step, select Debug Run ► Without Debugging. Visual Studio Code will compile and run the project and open a new browser window that will send an HTTP request to the ASP.NET Core server and produce the placeholder response. Request `http://localhost:5000/demo.html`, and you will receive the response shown in Figure 4-9.

Building and Running Projects Using Visual Studio

Visual Studio uses IIS Express as a reverse proxy for the built-in ASP.NET Core HTTP server that is used directly when you use the `dotnet run` command. When the project is created, an HTTP port is picked for IIS Express to use. To change the HTTP port to the one used throughout this book, select Project ► MyProject Properties and select the Debug section. Locate the App URL field and change the port number in the URL to 5000, as shown in Figure 4-10.

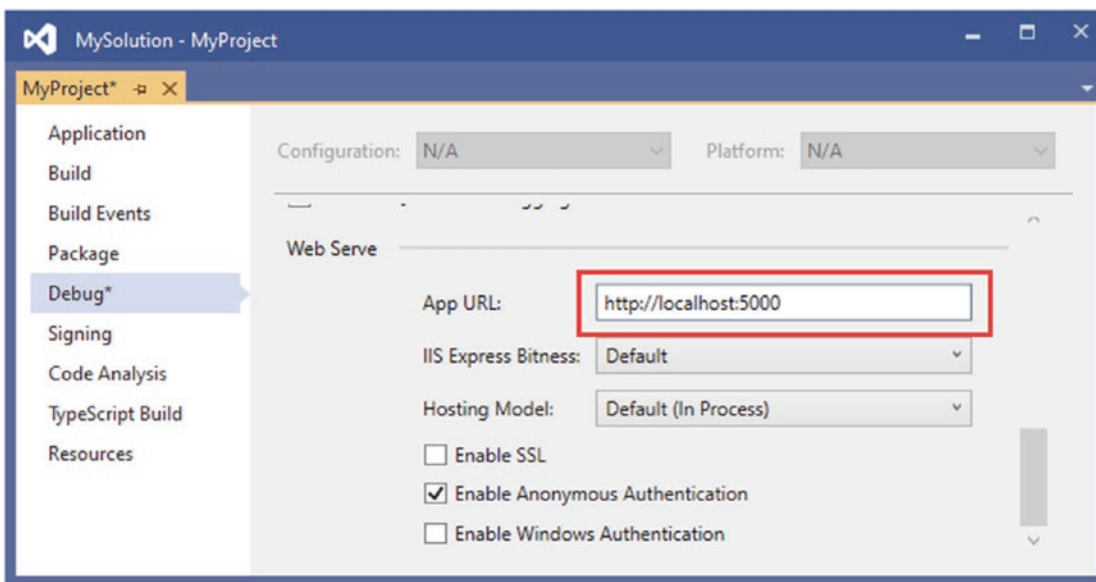


Figure 4-10. Changing the application port number

Select File ► Save All to apply the changes. To build the project, select Build Solution or Build MyProject from the Build menu. To build and run the project, select Debug ► Start Without Debugging. Once the project has been compiled, Visual Studio will open a new browser window, which sends an HTTP request that is received by IIS Express and passes it to the ASP.NET Core HTTP server, producing a placeholder response set up by the template used to create the project. Request `http://localhost:5000/demo.html`, and you will see the response shown in Figure 4-9.

■ **Tip** IIS Express is generally reliable, but if you have problems, right-click the IIS Express icon in the Windows taskbar's System Tray and select Exit from the popup menu.

Managing Packages

Most projects require additional features beyond those set up by the project templates, such as support for accessing databases or for making HTTP requests, neither of which is included in the standard ASP.NET Core packages added to the project by the template used to create the example project. In the sections that follow, I describe the tools available to manage the different types of packages that are used in ASP.NET Core development.

Managing NuGet Packages

.NET packages are added to a project with the `dotnet add package` command. Use a PowerShell command prompt to run the command shown in Listing 4-8 in the MyProject folder to add a package to the example project.

Listing 4-8. Adding a Package to the Example Project

```
dotnet add package Microsoft.EntityFrameworkCore.SqlServer --version 3.1.1
```

This command installs version 3.1.1 of the `Microsoft.EntityFrameworkCore.SqlServer` package. The package repository for .NET projects is [nuget.org](https://www.nuget.org), where you can search for the package and see the versions available. The package installed in Listing 4-8, for example, is described at <https://www.nuget.org/packages/Microsoft.EntityFrameworkCore.SqlServer/3.1.1>.

You can see the packages installed in a project by running the command shown in Listing 4-9.

■ **Tip** The project file—which is the file with the `.csproj` extension—is used to keep track of the packages added to a project. You can examine this file by opening it for editing in Visual Studio Code or by right-clicking the project item in the Visual Studio Solution Explorer and selecting Edit Project File from the popup menu.

Listing 4-9. Listing the Packages in a Project

```
dotnet list package
```

This command produces the following output when it is run in the MyProject folder, showing the package added in Listing 4-8:

```
[netcoreapp3.1]:
Top-level Package           Requested  Resolved
> Microsoft.EntityFrameworkCore.SqlServer 3.1.1     3.1.1
```

Packages are removed with the `dotnet remove package` command. To remove the package from the example project, run the command shown in Listing 4-10 in the MyProject folder.

Listing 4-10. Removing a Package from the Example Project

```
dotnet remove package Microsoft.EntityFrameworkCore.SqlServer
```

Managing Tool Packages

Tool packages install commands that can be used from the command line to perform operations on .NET Core projects. One common example is the Entity Framework Core tools package that installs commands that are used to manage databases in ASP.NET Core projects. Tool packages are managed using the `dotnet tool` command. To install the Entity Framework Core tools package, run the commands shown in Listing 4-11.

Listing 4-11. Installing a Tool Package

```
dotnet tool uninstall --global dotnet-ef
dotnet tool install --global dotnet-ef --version 3.1.1
```

The first command removes the `dotnet-ef` package, which is named `dotnet-ef`. This command will produce an error if the package has not already been installed, but it is a good idea to remove existing versions before installing a package. The `dotnet tool install` command installs version 3.1.1 of the `dotnet-ef` package, which is the version I use in this book. The commands installed by tool packages are used through the `dotnet` command. To test the package installed in Listing 4-11, run the command shown in Listing 4-12 in the `MyProject` folder.

■ **Tip** The `--global` arguments in Listing 4-11 mean the package is installed for global use and not just for a specific project. You can install tool packages into just one project, in which case the command is accessed with `dotnet tool run <command>`. The tools I use in this book are all installed globally.

Listing 4-12. Running a Tool Package Command

```
dotnet ef --help
```

The commands added by this tool package are accessed using `dotnet ef`, and you will see examples in later chapters that rely on these commands.

Managing Client-Side Packages

Client-side packages contain content that is delivered to the client, such as images, CSS stylesheets, JavaScript files, and static HTML. Client-side packages are added to ASP.NET Core using the Library Manager (LibMan) tool. To install the LibMan tool package, run the commands shown in Listing 4-13.

Listing 4-13. Installing the LibMan Tool Package

```
dotnet tool uninstall --global Microsoft.Web.LibraryManager.Cli
dotnet tool install --global Microsoft.Web.LibraryManager.Cli --version 2.0.96
```

These commands remove any existing LibMan package and install the version that is used throughout this book. The next step is to initialize the project, which creates the file that LibMan uses to keep track of the client packages it installs. Run the command shown in Listing 4-14 in the `MyProject` folder to initialize the example project.

Listing 4-14. Initializing the Example Project

```
libman init -p cdnjs
```

LibMan can download packages from different repositories. The `-p` argument in Listing 4-14 specifies the repository at <https://cdnjs.com>, which is the most widely used. Once the project is initialized, client-side packages can be installed. To install the Bootstrap CSS framework that I use to style HTML content throughout this book, run the command shown in Listing 4-15 in the `MyProject` folder.

Listing 4-15. Installing the Bootstrap CSS Framework

```
libman install twitter-bootstrap@4.3.1 -d wwwroot/lib/twitter-bootstrap
```

The command installs version 4.3.1 of the Bootstrap package, which is known by the name `twitter-bootstrap` on the CDNJS repository. There is some inconsistency in the way that popular packages are named on different repositories, and it is worth checking that you are getting the package you expect before adding to your project. The `-d` argument specifies the location into which the package is installed. The convention in ASP.NET Core projects is to install client-side packages into the `wwwroot/lib` folder.

Once the package has been installed, add the classes shown in Listing 4-16 to the elements in the `demo.html` file. This is how the features provided by the Bootstrap package are applied.

■ **Note** I don't get into the details of using the Bootstrap CSS framework in this book. See <https://getbootstrap.com> for the Bootstrap documentation.

Listing 4-16. Applying Bootstrap Classes in the `demo.html` File in the `wwwroot` Folder

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title></title>
  <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
  <h3 class="bg-primary text-white text-center p-2">
    HTML File from MyProject
  </h3>
</body>
</html>
```

Start ASP.NET Core and request `http://localhost:5000/demo.html`, and you will see the styled content shown in Figure 4-11.

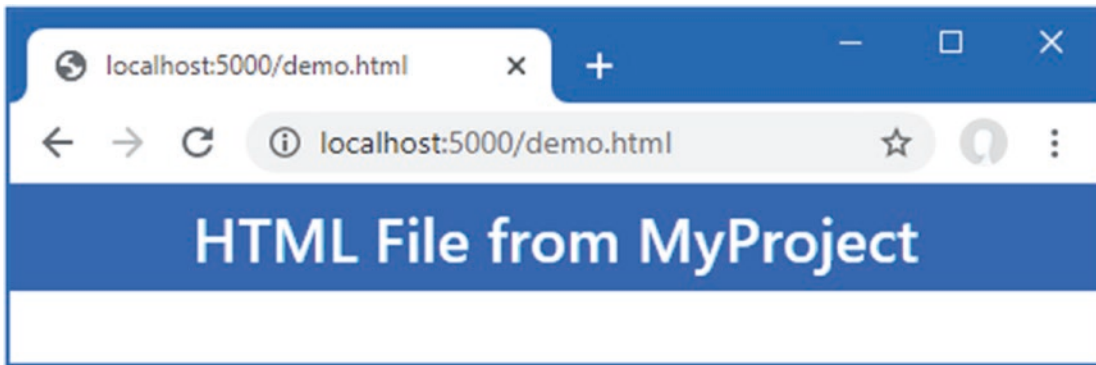


Figure 4-11. Using a client-side package

Managing Packages Using Visual Studio

Visual Studio provides tools for managing packages without using the command line. Select Project ► Manage NuGet Packages, and Visual Studio will open the NuGet package manager tool. Click Browse and enter **Microsoft.EntityFrameworkCore.SqlServer** into the search box to search for matching packages. Click the **Microsoft.EntityFrameworkCore.SqlServer** entry, which should be at the top of the list, and you will be able to choose a version and install the package, as shown in Figure 4-12.

■ **Caution** The Visual Studio NuGet package manager cannot be used to install global tool packages, which can be installed only from the command line.

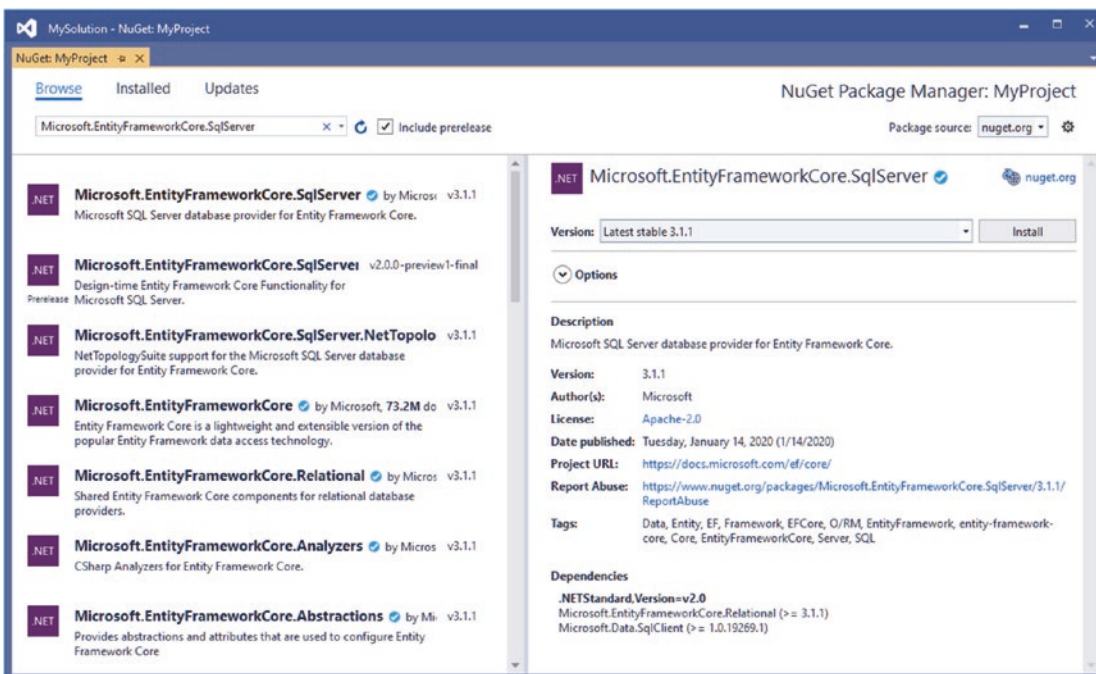


Figure 4-12. Using the Visual Studio package manager

The Visual Studio NuGet package manager can also be used to inspect the packages that have been installed in the project and check whether new versions of packages are available.

Managing Client-Side Packages Using Visual Studio

Visual Studio provides a separate tool for managing client-side packages. Right-click the MyProject item in the Solution Explorer and select Add ► Client Side Library from the popup menu. The client-side package tool is rudimentary, but it allows you to perform basic searches, select the files that are added to the project, and set the install location, as shown in Figure 4-13.

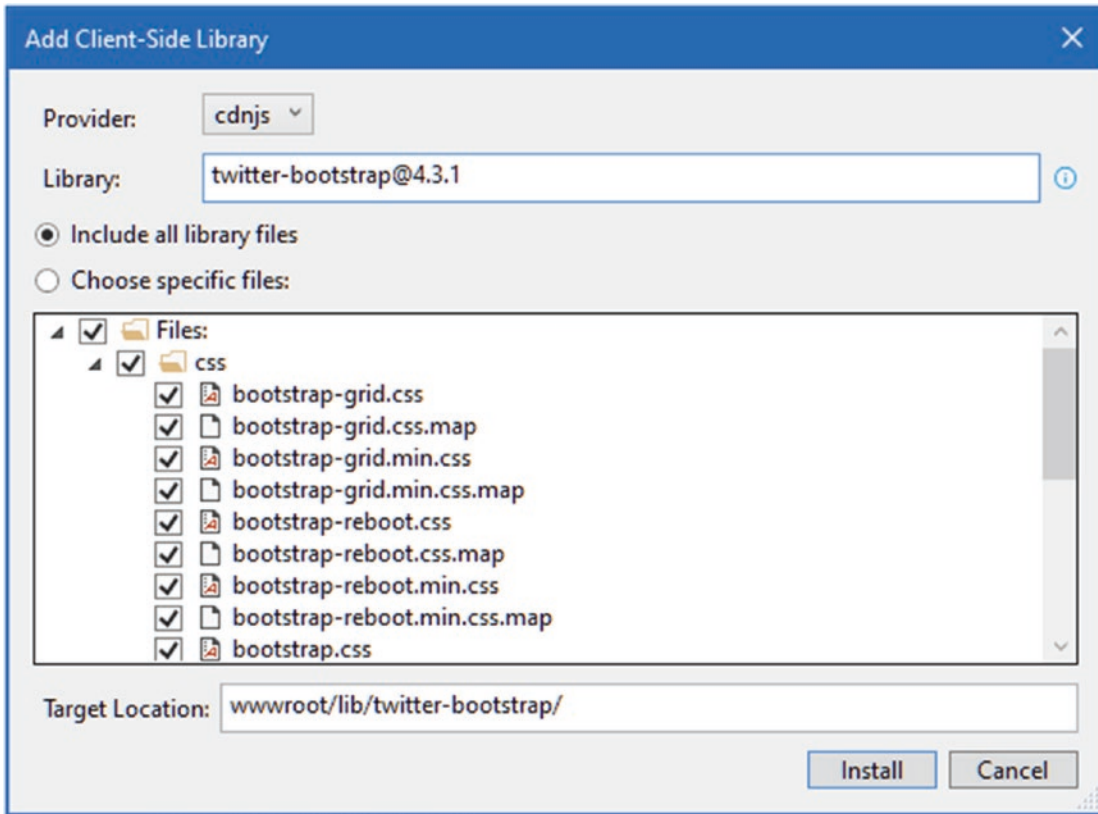


Figure 4-13. Using the Visual Studio client-side package manager

Debugging Projects

Visual Studio and Visual Studio Code both provide debuggers that can be used to control and inspect the execution of an ASP.NET Core application. Open the Startup.cs file in the MyProject folder, and click this statement in the code editor:

```
...
await context.Response.WriteAsync("Hello World!");
...
```

Select Debug ► Toggle Breakpoint, which is available in both Visual Studio and Visual Studio Code. A breakpoint is shown as a red dot alongside the code statement, as shown in Figure 4-14, and will interrupt execution and pass control to the user.

Start the project by selecting Debug ► Start Debugging, which is available in both Visual Studio and Visual Studio Code. (Choose .NET Core if Visual Studio Code prompts you to select an environment and then select the Start Debugging menu item again.)

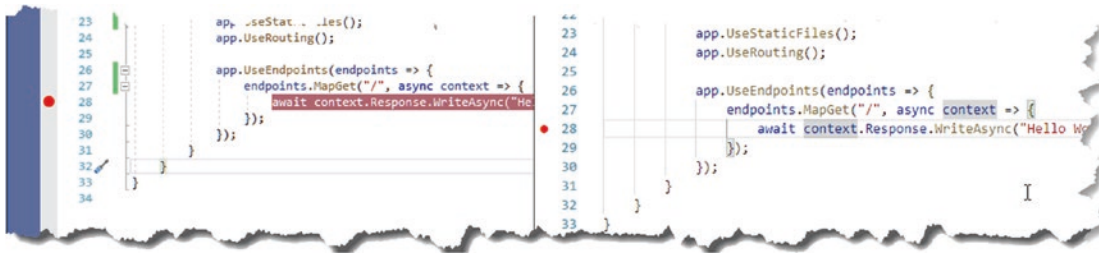


Figure 4-14. Setting a breakpoint

The application will be started and continue normally until the statement to which the breakpoint is reached, at which point execution is halted. Execution can be controlled using the Debug menu or the controls that Visual Studio and Visual Studio Code display. Both debuggers are packed with features—more so if you have a paid-for version of Visual Studio—and I don’t describe them in depth in this book. The Visual Studio 2019 debugger is described at <https://docs.microsoft.com/en-us/visualstudio/debugger/?view=vs-2019>, and the Visual Studio Code debugger is described at <https://code.visualstudio.com/docs/editor/debugging>.

HOW I DEBUG MY CODE

Debuggers are powerful tools, but I rarely use them. In most situations, I prefer to add `System.Console.WriteLine` statements to my code to figure out what is going on, which I can easily do because I tend to use the `dotnet run` command to run my projects from the command line. This is a rudimentary approach that works for me, not least because most of the errors in my code tend to be where statements are not being called because a condition in an `if` statement isn’t effective. If I want to examine an object in detail, I tend to serialize it to JSON and pass the result to the `WriteLine` method.

This may seem like madness if you are a dedicated user of the debugger, but it has the advantage of being quick and simple. When I am trying to figure out why code isn’t working, I want to explore and iterate quickly, and I find the amount of time taken to start the debugger to be a barrier. My approach is also reliable. The Visual Studio and Visual Studio Code debuggers are sophisticated, but they are not always entirely predictable, and .NET Core and ASP.NET Core change too quickly for the debugger features to have entirely settled down. When I am utterly confused by the behavior of some code, I want the simplest possible diagnostic tool, and that, for me, is a message written to the console.

I am not suggesting that this is the approach you should use, but it can be a good place to start when you are not getting the results you expect and you don’t want to battle with the debugger to figure out why.

Summary

In this chapter, I described the tools used for ASP.NET Core development. I explained that the command-line tools are the most concise and reliable way to work with ASP.NET Core projects, which is why I have used them in the examples in this book. I also demonstrated the alternative user interfaces that Visual Studio and Visual Studio Code provide, which can be a useful alternative for some—but not all—of the command-line tools. In the next chapter, I describe the C# features that are essential for effective ASP.NET Core development.

CHAPTER 5



Essential C# Features

In this chapter, I describe C# features used in web application development that are not widely understood or that often cause confusion. This is not a book about C#, however, so I provide only a brief example for each feature so that you can follow the examples in the rest of the book and take advantage of these features in your own projects. Table 5-1 summarizes this chapter.

Table 5-1. Chapter Summary

Problem	Solution	Listing
Managing null values	Use the null conditional and null coalescing operators	7, 10
Creating properties with getters and setters	Define automatically implemented properties	11–13
Mixing static and dynamic values in strings	Use string interpolation	14
Initializing and populate objects	Use the object and collection initializers	15–18
Assigning a value for specific types	Use pattern matching	19, 20
Extending the functionality of a class without modifying it	Define an extension method	21–28
Expressing functions and methods concisely	Use lambda expressions	29–36
Defining a variable without explicitly declaring its type	Use the var keyword	37–39
Modifying an interface without requiring changes in its implementation classes	Define a default implementation	40–44
Performing work asynchronously	Use tasks or the <code>async/await</code> keywords	45–47
Producing a sequence of values over time	Use an asynchronous enumerable	48–51
Getting the name of a class or member	Use a <code>nameof</code> expression	52, 53

Preparing for This Chapter

To create the example project for this chapter, open a new PowerShell command prompt and run the commands shown in Listing 5-1. If you are using Visual Studio and prefer not to use the command line, you can create the project using the process described in Chapter 4.

■ **Tip** You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/apress/pro-asp.net-core-3>. See Chapter 1 for how to get help if you have problems running the examples.

Listing 5-1. Creating the Example Project

```
dotnet new globaljson --sdk-version 3.1.101 --output LanguageFeatures
dotnet new web --no-https --output LanguageFeatures --framework netcoreapp3.1
dotnet new sln -o LanguageFeatures

dotnet sln LanguageFeatures add LanguageFeatures
```

Opening the Project

If you are using Visual Studio, select File ► Open ► Project/Solution, select the `LanguageFeatures.sln` file in the `LanguageFeatures` folder, and click the Open button to open the solution file and the project it references. If you are using Visual Studio Code, select File ► Open Folder, navigate to the `LanguageFeatures` folder, and click the Select Folder button.

Enabling the MVC Framework

The web project template creates a project that contains a minimal ASP.NET Core configuration. This means the placeholder content that is added by the `mvc` template used in Chapter 3 is not available and that extra steps are required to reach the point where the application can produce useful output. In this section, I make the changes required to set up the MVC Framework, which is one of the application frameworks supported by ASP.NET Core, as I explained in Chapter 1. First, to enable the MVC framework, make the changes shown in Listing 5-2 to the `Startup` class.

Listing 5-2. Enabling MVC in the `Startup.cs` File in the `LanguageFeatures` Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

namespace LanguageFeatures {
    public class Startup {
        public void ConfigureServices(IServiceCollection services) {
            services.AddControllersWithViews();
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
            if (env.IsDevelopment()) {
                app.UseDeveloperExceptionPage();
            }

            app.UseRouting();

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

I explain how to configure ASP.NET Core applications in Part 2, but the two statements added in Listing 5-2 provide a basic MVC framework setup using a default configuration.

Creating the Application Components

Now that the MVC framework is set up, I can add the application components that I will use to demonstrate important C# language features.

Creating the Data Model

I started by creating a simple model class so that I can have some data to work with. I added a folder called Models and created a class file called Product.cs within it, which I used to define the class shown in Listing 5-3.

Listing 5-3. The Contents of the Product.cs File in the Models Folder

```
namespace LanguageFeatures.Models {
    public class Product {

        public string Name { get; set; }
        public decimal? Price { get; set; }

        public static Product[] GetProducts() {

            Product kayak = new Product {
                Name = "Kayak", Price = 275M
            };

            Product lifejacket = new Product {
                Name = "Lifejacket", Price = 48.95M
            };

            return new Product[] { kayak, lifejacket, null };
        }
    }
}
```

The Product class defines Name and Price properties, and there is a static method called GetProducts that returns a Product array. One of the elements contained in the array returned by the GetProducts method is set to null, which I will use to demonstrate some useful language features later in the chapter.

Creating the Controller and View

For the examples in this chapter, I use a simple controller class to demonstrate different language features. I created a Controllers folder and added to it a class file called HomeController.cs, the contents of which are shown in Listing 5-4.

Listing 5-4. The Contents of the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public IActionResult Index() {
            return View(new string[] { "C#", "Language", "Features" });
        }
    }
}
```

The Index action method tells ASP.NET Core to render the default view and provides it with an array of strings as its view model, which will be included in the HTML sent to the client. To create the view, I added a Views/Home folder (by creating a Views folder and then adding a Home folder within it) and added a Razor View called Index.cshtml, the contents of which are shown in Listing 5-5.

Listing 5-5. The Contents of the Index.cshtml File in the Views/Home Folder

```
@model IEnumerable<string>
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>Language Features</title>
</head>
<body>
  <ul>
    @foreach (string s in Model) {
      <li>@s</li>
    }
  </ul>
</body>
</html>
```

Selecting the HTTP Port

If you are using Visual Studio, select Project ► LanguageFeatures Properties, select the Debug section, and change the HTTP port to 5000 in the App URL field, as shown in Figure 5-1. Select File ► Save All to save the new port. (This change is not required if you are using Visual Studio Code.)

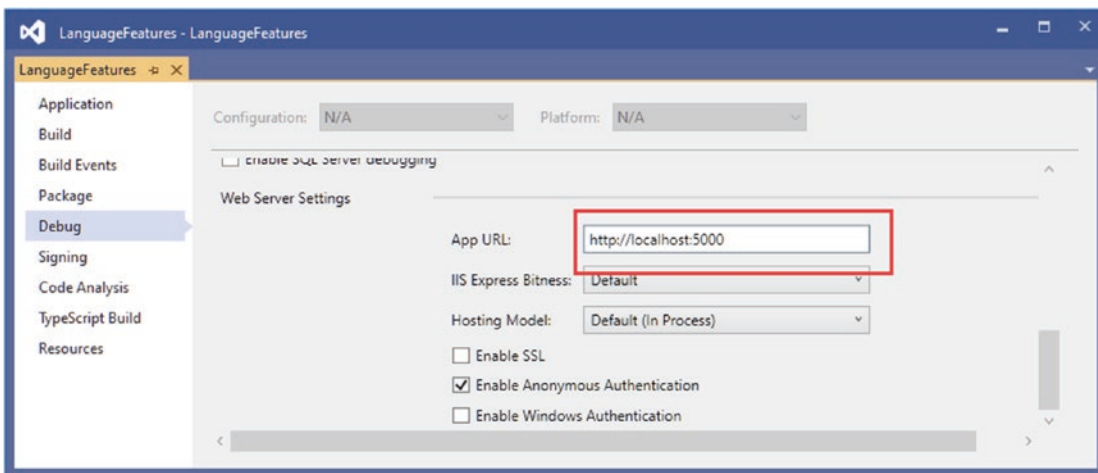


Figure 5-1. Setting the HTTP port

Running the Example Application

Start ASP.NET Core by selecting Start Without Debugging (Visual Studio) or Run Without Debugging (Visual Studio Code) from the Debug menu or by running the command shown in Listing 5-6 in the LanguageFeatures folder.

Listing 5-6. Running the Example Application

```
dotnet run
```

Request `http://localhost:5000`, and you will see the output shown in Figure 5-2.

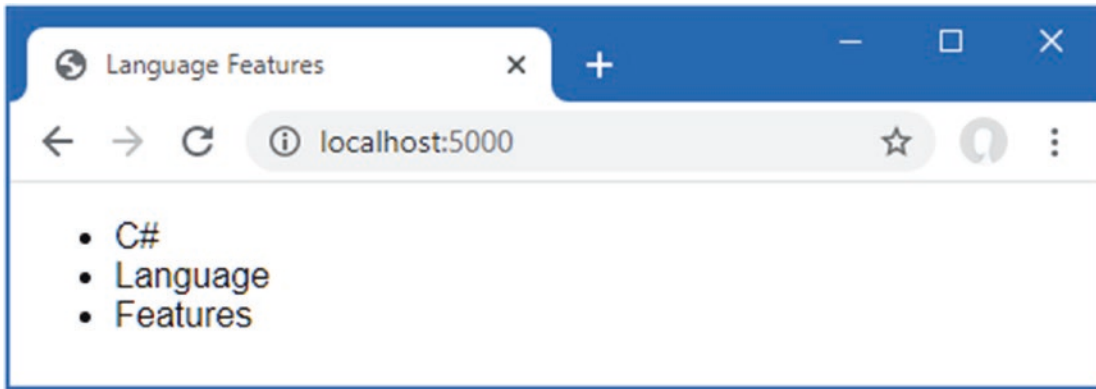


Figure 5-2. Running the example application

Since the output from all the examples in this chapter is text, I will show the messages displayed by the browser like this:

```
C#
Language
Features
```

Using the Null Conditional Operator

The null conditional operator allows for null values to be detected more elegantly. There can be a lot of checking for nulls in ASP.NET Core development as you work out whether a request contains a specific header or value or whether the model contains a specific data item. Traditionally, dealing with null values requires making an explicit check, which can become tedious and error-prone when both an object and its properties need to be inspected. The null conditional operator makes this process simpler and more concise, as shown in Listing 5-7.

Listing 5-7. Detecting null Values in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public IActionResult Index() {

            List<string> results = new List<string>();

            foreach (Product p in Product.GetProducts()) {
                string name = p?.Name;
                decimal? price = p?.Price;
                results.Add(string.Format("Name: {0}, Price: {1}", name, price));
            }
        }
    }
}
```

```

        return View(results);
    }
}

```

The static `GetProducts` method defined by the `Product` class returns an array of objects that I inspect in the `Index` action method to get a list of the `Name` and `Price` values. The problem is that both the object in the array and the value of the properties could be `null`, which means I can't just refer to `p.Name` or `p.Price` within the `foreach` loop without causing a `NullReferenceException`. To avoid this, I used the null conditional operator, like this:

```

...
string name = p?.Name;
decimal? price = p?.Price;
...

```

The null conditional operator is a single question mark (the `?` character). If `p` is `null`, then `name` will be set to `null` as well. If `p` is not `null`, then `name` will be set to the value of the `Person.Name` property. The `Price` property is subject to the same test. Notice that the variable you assign to when using the null conditional operator must be able to be assigned `null`, which is why the price variable is declared as a nullable decimal (`decimal?`).

Chaining the Null Conditional Operator

The null conditional operator can be chained to navigate through a hierarchy of objects, which is where it becomes an effective tool for simplifying code and allowing safe navigation. In Listing 5-8, I have added a property to the `Product` class that creates a more complex object hierarchy.

Listing 5-8. Adding a Property in the `Product.cs` File in the `Models` Folder

```

namespace LanguageFeatures.Models {
    public class Product {

        public string Name { get; set; }
        public decimal? Price { get; set; }
        public Product Related { get; set; }

        public static Product[] GetProducts() {

            Product kayak = new Product {
                Name = "Kayak", Price = 275M
            };
            Product lifejacket = new Product {
                Name = "Lifejacket", Price = 48.95M
            };

            kayak.Related = lifejacket;

            return new Product[] { kayak, lifejacket, null };
        }
    }
}

```

Each `Product` object has a `Related` property that can refer to another `Product` object. In the `GetProducts` method, I set the `Related` property for the `Product` object that represents a kayak. Listing 5-9 shows how I can chain the null conditional operator to navigate through the object properties without causing an exception.

Listing 5-9. Detecting Nested null Values in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public IActionResult Index() {

            List<string> results = new List<string>();

            foreach (Product p in Product.GetProducts()) {
                string name = p?.Name;
                decimal? price = p?.Price;
                string relatedName = p?.Related?.Name;
                results.Add(string.Format("Name: {0}, Price: {1}, Related: {2}",
                    name, price, relatedName));
            }

            return View(results);
        }
    }
}
```

The null conditional operator can be applied to each part of a chain of properties, like this:

```
...
string relatedName = p?.Related?.Name;
...
```

The result is that the relatedName variable will be null when p is null or when p.Related is null. Otherwise, the variable will be assigned the value of the p.Related.Name property. Restart ASP.NET Core and request `http://localhost:5000`, and you will see the following output in the browser window:

```
Name: Kayak, Price: 275, Related: Lifejacket
Name: Lifejacket, Price: 48.95, Related:
Name: , Price: , Related:
```

Combining the Conditional and Coalescing Operators

It can be useful to combine the null conditional operator (a single question mark) with the null coalescing operator (two question marks) to set a fallback value to prevent null values being used in the application, as shown in Listing 5-10.

Listing 5-10. Combining Null Operators in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {
```

```

public ActionResult Index() {
    List<string> results = new List<string>();

    foreach (Product p in Product.GetProducts()) {
        string name = p?.Name ?? "<No Name>";
        decimal? price = p?.Price ?? 0;
        string relatedName = p?.Related?.Name ?? "<None>";
        results.Add(string.Format("Name: {0}, Price: {1}, Related: {2}",
            name, price, relatedName));
    }

    return View(results);
}
}
}

```

The null conditional operator ensures that I don't get a `NullReferenceException` when navigating through the object properties, and the null coalescing operator ensures that I don't include null values in the results displayed in the browser. If you run the example, you will see the following results displayed in the browser window:

```

Name: Kayak, Price: 275, Related: Lifejacket
Name: Lifejacket, Price: 48.95, Related: <None>
Name: <No Name>, Price: 0, Related: <None>

```

NULLABLE AND NON-NULLABLE REFERENCE TYPES

Encountering unexpected null values is one of the most common causes of bugs. By default, C# treats null as a valid value for all types, which means that I can assign null to a string variable, like this:

```

...
string product = null;
...

```

It is the responsibility of the code that uses the variable to check for null values, which can be especially problematic when the same variable is in multiple places. It is easy to omit one of the checks or assume that a value won't be null, producing an error at runtime.

Nullable reference types shift responsibility for null checking to the code that assigns the value to a variable. When the nullable reference feature is enabled, regular reference types cannot be assigned null values, such as assigning null to a string, for example. Instead, nullable reference types must be used if null values are possible, like this:

```

...
string product = null; // compiler error - this is a non-nullable type
string? product = null; // no error - this is a nullable type
...

```

The `string?` type is nullable, while `string` is not, which means that the code that consumes a variable doesn't have to worry about null values unless it is dealing with a nullable type. To enable nullable reference types, an element must be added to the csproj file, like this:

```

...
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
    <Nullable>enable</Nullable>
  </PropertyGroup>

```

```
</Project>
...
```

If you are using Visual Studio, you can open the project file by right-clicking the project item in the Solution Explorer and selecting Edit Project File from the popup menu.

I like this feature, but it is not yet used widely enough for me to use it in this book, especially since it can make some complex topics even more difficult to follow. But, once the rest of .NET Core catches up with this feature, I expect it to be embraced in ASP.NET Core by default, and you can expect to see nullable reference types used in future editions of this book.

Using Automatically Implemented Properties

C# supports automatically implemented properties, and I used them when defining properties for the Person class in the previous section, like this:

```
namespace LanguageFeatures.Models {
    public class Product {

        public string Name { get; set; }
        public decimal? Price { get; set; }
        public Product Related { get; set; }

        public static Product[] GetProducts() {

            Product kayak = new Product {
                Name = "Kayak", Price = 275M
            };
            Product lifejacket = new Product {
                Name = "Lifejacket", Price = 48.95M
            };

            kayak.Related = lifejacket;

            return new Product[] { kayak, lifejacket, null };
        }
    }
}
```

This feature allows me to define properties without having to implement the get and set bodies. Using the auto-implemented property feature means I can define a property like this:

```
...
public string Name { get; set; }
...
is equivalent to the following code:
...
public string Name {
    get { return name; }
    set { name = value; }
}
...
```

This type of feature is known as *syntactic sugar*, which means that it makes C# more pleasant to use—in this case by eliminating redundant code that ends up being duplicated for every property—without substantially altering the way that the language behaves. The term *sugar* may seem pejorative, but any enhancements that make code easier to write and maintain can be beneficial, especially in large and complex projects.

Using Auto-implemented Property Initializers

Automatically implemented properties have been supported since C# 3.0. The latest version of C# supports initializers for automatically implemented properties, which allows an initial value to be set without having to use the constructor, as shown in Listing 5-11.

Listing 5-11. Using an Auto-implemented Property Initializer in the Product.cs File in the Models Folder

```
namespace LanguageFeatures.Models {
    public class Product {

        public string Name { get; set; }
        public string Category { get; set; } = "Watersports";
        public decimal? Price { get; set; }
        public Product Related { get; set; }

        public static Product[] GetProducts() {
            Product kayak = new Product {
                Name = "Kayak",
                Category = "Water Craft",
                Price = 275M
            };
            Product lifejacket = new Product {
                Name = "Lifejacket", Price = 48.95M
            };

            kayak.Related = lifejacket;

            return new Product[] { kayak, lifejacket, null };
        }
    }
}
```

Assigning a value to an auto-implemented property doesn't prevent the setter from being used to change the property later and just tidies up the code for simple types that ended up with a constructor that contained a list of property assignments to provide default values. In the example, the initializer assigns a value of `Watersports` to the `Category` property. The initial value can be changed, which I do when I create the `kayak` object and specify a value of `Water Craft` instead.

Creating Read-Only Automatically Implemented Properties

You can create a read-only property by using an initializer and omitting the `set` keyword from an auto-implemented property that has an initializer, as shown in Listing 5-12.

Listing 5-12. Creating a Read-Only Property in the Product.cs File in the Models Folder

```
namespace LanguageFeatures.Models {
    public class Product {

        public string Name { get; set; }
        public string Category { get; set; } = "Watersports";
        public decimal? Price { get; set; }
    }
}
```

```

public Product Related { get; set; }
public bool InStock { get; } = true;

public static Product[] GetProducts() {

    Product kayak = new Product {
        Name = "Kayak",
        Category = "Water Craft",
        Price = 275M
    };
    Product lifejacket = new Product {
        Name = "Lifejacket", Price = 48.95M
    };

    kayak.Related = lifejacket;

    return new Product[] { kayak, lifejacket, null };
}
}
}

```

The `InStock` property is initialized to `true` and cannot be changed; however, the value can be assigned to in the type's constructor, as shown in Listing 5-13.

Listing 5-13. Assigning a Value to a Read-Only Property in the `Product.cs` File in the `Models` Folder

```

namespace LanguageFeatures.Models {
    public class Product {

        public Product(bool stock = true) {
            InStock = stock;
        }

        public string Name { get; set; }
        public string Category { get; set; } = "Watersports";
        public decimal? Price { get; set; }
        public Product Related { get; set; }
        public bool InStock { get; }

        public static Product[] GetProducts() {
            Product kayak = new Product {
                Name = "Kayak",
                Category = "Water Craft",
                Price = 275M
            };

            Product lifejacket = new Product(false) {
                Name = "Lifejacket",
                Price = 48.95M
            };

            kayak.Related = lifejacket;

            return new Product[] { kayak, lifejacket, null };
        }
    }
}

```

The constructor allows the value for the read-only property to be specified as an argument and defaults to `true` if no value is provided. The property value cannot be changed once set by the constructor.

Using String Interpolation

The `string.Format` method is the traditional C# tool for composing strings that contain data values. Here is an example of this technique from the `Home` controller:

```
...
results.Add(string.Format("Name: {0}, Price: {1}, Related: {2}",
                name, price, relatedName));
...
```

C# also supports a different approach, known as *string interpolation*, that avoids the need to ensure that the `{0}` references in the string template match up with the variables specified as arguments. Instead, string interpolation uses the variable names directly, as shown in Listing 5-14.

Listing 5-14. Using String Interpolation in the `HomeController.cs` File in the `Controllers` Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public IActionResult Index() {

            List<string> results = new List<string>();

            foreach (Product p in Product.GetProducts()) {
                string name = p?.Name ?? "<No Name>";
                decimal? price = p?.Price ?? 0;
                string relatedName = p?.Related?.Name ?? "<None>";
                results.Add($"Name: {name}, Price: {price}, Related: {relatedName}");
            }

            return View(results);
        }
    }
}
```

Interpolated strings are prefixed with the `$` character and contain *holes*, which are references to values contained within the `{` and `}` characters. When the string is evaluated, the holes are filled in with the current values of the variables or constants that are specified.

■ **Tip** String interpolation supports all the format specifiers that are available with the `string.Format` method. The format specifics are included as part of the hole, so `$"Price: {price:C2}"` would format the `price` value as a currency value with two decimal digits.

Using Object and Collection Initializers

When I create an object in the static `GetProducts` method of the `Product` class, I use an *object initializer*, which allows me to create an object and specify its property values in a single step, like this:

```
...
Product kayak = new Product {
    Name = "Kayak",
    Category = "Water Craft",
    Price = 275M
};
...
```

This is another syntactic sugar feature that makes C# easier to use. Without this feature, I would have to call the `Product` constructor and then use the newly created object to set each of the properties, like this:

```
...
Product kayak = new Product();
kayak.Name = "Kayak";
kayak.Category = "Water Craft";
kayak.Price = 275M;
...
```

A related feature is the *collection initializer*, which allows the creation of a collection and its contents to be specified in a single step. Without an initializer, creating a string array, for example, requires the size of the array and the array elements to be specified separately, as shown in Listing 5-15.

Listing 5-15. Initializing an Object in the `HomeController.cs` File in the `Controllers` Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public IActionResult Index() {
            string[] names = new string[3];
            names[0] = "Bob";
            names[1] = "Joe";
            names[2] = "Alice";
            return View("Index", names);
        }
    }
}
```

Using a collection initializer allows the contents of the array to be specified as part of the construction, which implicitly provides the compiler with the size of the array, as shown in Listing 5-16.

Listing 5-16. Using a Collection Initializer in the `HomeController.cs` File in the `Controllers` Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {
```

```

    public IActionResult Index() {
        return View("Index", new string[] { "Bob", "Joe", "Alice" });
    }
}

```

The array elements are specified between the { and } characters, which allows for a more concise definition of the collection and makes it possible to define a collection inline within a method call. The code in Listing 5-16 has the same effect as the code in Listing 5-15. Restart ASP.NET Core and request `http://localhost:5000`, and you will see the following output in the browser window:

```

Bob
Joe
Alice

```

Using an Index Initializer

Recent versions of C# tidy up the way collections that use indexes, such as dictionaries, are initialized. Listing 5-17 shows the Index action rewritten to define a collection using the traditional C# approach to initializing a dictionary.

Listing 5-17. Initializing a Dictionary in the HomeController.cs File in the Controllers Folder

```

using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public IActionResult Index() {
            Dictionary<string, Product> products = new Dictionary<string, Product> {
                { "Kayak", new Product { Name = "Kayak", Price = 275M } },
                { "Lifejacket", new Product { Name = "Lifejacket", Price = 48.95M } }
            };
            return View("Index", products.Keys);
        }
    }
}

```

The syntax for initializing this type of collection relies too much on the { and } characters, especially when the collection values are created using object initializers. The latest versions of C# support a more natural approach to initializing indexed collections that is consistent with the way that values are retrieved or modified once the collection has been initialized, as shown in Listing 5-18.

Listing 5-18. Using Collection Initializer Syntax in the HomeController.cs File in the Controllers Folder

```

using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public IActionResult Index() {
            Dictionary<string, Product> products = new Dictionary<string, Product> {
                ["Kayak"] = new Product { Name = "Kayak", Price = 275M },
            };
            return View("Index", products.Keys);
        }
    }
}

```

```

        ["Lifejacket"] = new Product { Name = "Lifejacket", Price = 48.95M }
    };

    return View("Index", products.Keys);
}
}
}

```

The effect is the same—to create a dictionary whose keys are `Kayak` and `Lifejacket` and whose values are `Product` objects—but the elements are created using the index notation that is used for other collection operations. Restart ASP.NET Core and request `http://localhost:5000`, and you will see the following results in the browser:

```

Kayak
Lifejacket

```

Pattern Matching

One of the most useful recent additions to C# is support for pattern matching, which can be used to test that an object is of a specific type or has specific characteristics. This is another form of syntactic sugar, and it can dramatically simplify complex blocks of conditional statements. The `is` keyword is used to perform a type test, as demonstrated in Listing 5-19.

Listing 5-19. Performing a Type Test in the `HomeController.cs` File in the `Controllers` Folder

```

using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public IActionResult Index() {

            object[] data = new object[] { 275M, 29.95M,
                "apple", "orange", 100, 10 };
            decimal total = 0;
            for (int i = 0; i < data.Length; i++) {
                if (data[i] is decimal d) {
                    total += d;
                }
            }

            return View("Index", new string[] { $"Total: {total:C2}" });
        }
    }
}

```

The `is` keyword performs a type check and, if a value is of the specified type, will assign the value to a new variable, like this:

```

...
if (data[i] is decimal d) {
...

```

This expression evaluates as true if the value stored in `data[i]` is a decimal. The value of `data[i]` will be assigned to the variable `d`, which allows it to be used in subsequent statements without needing to perform any type conversions. The `is` keyword will match only the specified type, which means that only two of the values in the `data` array will be processed (the other items in the array are `string` and `int` values). If you run the application, you will see the following output in the browser window:

Total: \$304.95

Pattern Matching in switch Statements

Pattern matching can also be used in `switch` statements, which support the `when` keyword for restricting when a value is matched by a case statement, as shown in Listing 5-20.

Listing 5-20. Pattern Matching in the `HomeController.cs` File in the `Controllers` Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public IActionResult Index() {

            object[] data = new object[] { 275M, 29.95M,
                "apple", "orange", 100, 10 };
            decimal total = 0;
            for (int i = 0; i < data.Length; i++) {
                switch (data[i]) {
                    case decimal decimalValue:
                        total += decimalValue;
                        break;
                    case int intValue when intValue > 50:
                        total += intValue;
                        break;
                }
            }

            return View("Index", new string[] { $"Total: {total:C2}" });
        }
    }
}
```

To match any value of a specific type, use the type and variable name in the case statement, like this:

```
...
case decimal decimalValue:
...
```

This case statement matches any decimal value and assigns it to a new variable called `decimalValue`. To be more selective, the `when` keyword can be included, like this:

```
...
case int intValue when intValue > 50:
...
```

This case statement matches `int` values and assigns them to a variable called `intValue`, but only when the value is greater than 50. Restart ASP.NET Core and request `http://localhost:5000`, and you will see the following output in the browser window:

Total: \$404.95

Using Extension Methods

Extension methods are a convenient way of adding methods to classes that you cannot modify directly, typically because they are provided by Microsoft or a third-party package. Listing 5-21 shows the definition of the `ShoppingCart` class, which I added to the `Models` folder in a class file called `ShoppingCart.cs` file and which represents a collection of `Product` objects.

Listing 5-21. The Contents of the `ShoppingCart.cs` File in the `Models` Folder

```
using System.Collections.Generic;

namespace LanguageFeatures.Models {

    public class ShoppingCart {
        public IEnumerable<Product> Products { get; set; }
    }
}
```

This is a simple class that acts as a wrapper around a sequence of `Product` objects (I only need a basic class for this example). Suppose I need to be able to determine the total value of the `Product` objects in the `ShoppingCart` class, but I cannot modify the class because it comes from a third party, and I do not have the source code. I can use an extension method to add the functionality I need.

Add a class file named `MyExtensionMethods.cs` in the `Models` folder and use it to define the class shown in Listing 5-22.

Listing 5-22. The Contents of the `MyExtensionMethods.cs` File in the `Models` Folder

```
namespace LanguageFeatures.Models {

    public static class MyExtensionMethods {

        public static decimal TotalPrices(this ShoppingCart cartParam) {
            decimal total = 0;
            foreach (Product prod in cartParam.Products) {
                total += prod?.Price ?? 0;
            }
            return total;
        }
    }
}
```

Extension methods are defined in static classes within the same namespace as the class the extension methods applies to. In this case, the static `MyExtensionMethods` class is in the `LanguageFeatures.Models` namespace, which means that it can contain extension methods for classes in that namespace.

Extension methods are also static, and Listing 5-22 defines a single extension method named `TotalPrices`. The `this` keyword in front of the first parameter marks `TotalPrices` as an extension method. The first parameter tells .NET which class the extension method can be applied to—`ShoppingCart` in this case. I can refer to the instance of the `ShoppingCart` that the extension method has been applied to by using the `cartParam` parameter. This extension method enumerates the `Product` objects in the `ShoppingCart` and returns the sum of the `Product.Price` property values. Listing 5-23 shows how I apply the extension method in the `Home` controller's action method.

■ **Note** Extension methods do not let you break through the access rules that classes define for methods, fields, and properties. You can extend the functionality of a class by using an extension method but only using the class members that you had access to anyway.

Listing 5-23. Applying an Extension Method in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public IActionResult Index() {
            ShoppingCart cart
            = new ShoppingCart { Products = Product.GetProducts() };
            decimal cartTotal = cart.TotalPrices();
            return View("Index", new string[] { $"Total: {cartTotal:C2}" });
        }
    }
}
```

The key statement is this one:

```
...
decimal cartTotal = cart.TotalPrices();
...
```

I call the `TotalPrices` method on a `ShoppingCart` object as though it were part of the `ShoppingCart` class, even though it is an extension method defined by a different class altogether. .NET will find extension classes if they are in the scope of the current class, meaning that they are part of the same namespace or in a namespace that is the subject of a `using` statement. Restart ASP.NET Core and request `http://localhost:5000`, which will produce the following output in the browser window:

Total: \$323.95

Applying Extension Methods to an Interface

Extension methods can also be applied to an interface, which allows me to call the extension method on all the classes that implement the interface. Listing 5-24 shows the `ShoppingCart` class updated to implement the `IEnumerable<Product>` interface.

Listing 5-24. Implementing an Interface in the ShoppingCart.cs File in the Models Folder

```
using System.Collections;
using System.Collections.Generic;

namespace LanguageFeatures.Models {

    public class ShoppingCart : IEnumerable<Product> {
        public IEnumerable<Product> Products { get; set; }

        public IEnumerator<Product> GetEnumerator() {
            return Products.GetEnumerator();
        }
    }
}
```

```

    IEnumerator IEnumerable.GetEnumerator() {
        return GetEnumerator();
    }
}

```

I can now update the extension method so that it deals with `IEnumerable<Product>`, as shown in Listing 5-25.

Listing 5-25. Updating an Extension Method in the `MyExtensionMethods.cs` File in the Models Folder

```

using System.Collections.Generic;

namespace LanguageFeatures.Models {

    public static class MyExtensionMethods {

        public static decimal TotalPrices(this IEnumerable<Product> products) {
            decimal total = 0;
            foreach (Product prod in products) {
                total += prod?.Price ?? 0;
            }
            return total;
        }
    }
}

```

The first parameter type has changed to `IEnumerable<Product>`, which means that the `foreach` loop in the method body works directly on `Product` objects. The change to using the interface means that I can calculate the total value of the `Product` objects enumerated by any `IEnumerable<Product>`, which includes instances of `ShoppingCart` but also arrays of `Product` objects, as shown in Listing 5-26.

Listing 5-26. Applying an Extension Method in the `HomeController.cs` File in the Controllers Folder

```

using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public IActionResult Index() {

            ShoppingCart cart
                = new ShoppingCart { Products = Product.GetProducts() };

            Product[] productArray = {
                new Product {Name = "Kayak", Price = 275M},
                new Product {Name = "Lifejacket", Price = 48.95M}
            };

            decimal cartTotal = cart.TotalPrices();
            decimal arrayTotal = productArray.TotalPrices();

            return View("Index", new string[] {
                $"Cart Total: {cartTotal:C2}",
                $"Array Total: {arrayTotal:C2}" });
        }
    }
}

```

Restart ASP.NET Core and request `http://localhost:5000`, which will produce the following output in the browser, demonstrating that I get the same result from the extension method, irrespective of how the `Product` objects are collected:

```
Cart Total: $323.95
Array Total: $323.95
```

Creating Filtering Extension Methods

The last thing I want to show you about extension methods is that they can be used to filter collections of objects. An extension method that operates on an `IEnumerable<T>` and that also returns an `IEnumerable<T>` can use the `yield` keyword to apply selection criteria to items in the source data to produce a reduced set of results. Listing 5-27 demonstrates such a method, which I have added to the `MyExtensionMethods` class.

Listing 5-27. A Filtering Extension Method in the `MyExtensionMethods.cs` File in the `Models` Folder

```
using System.Collections.Generic;

namespace LanguageFeatures.Models {

    public static class MyExtensionMethods {

        public static decimal TotalPrices(this IEnumerable<Product> products) {
            decimal total = 0;
            foreach (Product prod in products) {
                total += prod?.Price ?? 0;
            }
            return total;
        }

        public static IEnumerable<Product> FilterByPrice(
            this IEnumerable<Product> productEnum, decimal minimumPrice) {

            foreach (Product prod in productEnum) {
                if ((prod?.Price ?? 0) >= minimumPrice) {
                    yield return prod;
                }
            }
        }
    }
}
```

This extension method, called `FilterByPrice`, takes an additional parameter that allows me to filter products so that `Product` objects whose `Price` property matches or exceeds the parameter are returned in the result. Listing 5-28 shows this method being used.

Listing 5-28. Using the Filtering Extension Method in the `HomeController.cs` File in the `Controllers` Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {
```



```

public ActionResult Index() {
    Product[] productArray = {
        new Product {Name = "Kayak", Price = 275M},
        new Product {Name = "Lifejacket", Price = 48.95M},
        new Product {Name = "Soccer ball", Price = 19.50M},
        new Product {Name = "Corner flag", Price = 34.95M}
    };

    decimal arrayTotal = productArray.FilterByPrice(20).TotalPrices();

    return View("Index", new string[] { $"Array Total: {arrayTotal:C2}" });
}
}
}

```

When I call the `FilterByPrice` method on the array of `Product` objects, only those that cost more than \$20 are received by the `TotalPrices` method and used to calculate the total. If you run the application, you will see the following output in the browser window:

Total: \$358.90

Using Lambda Expressions

Lambda expressions are a feature that causes a lot of confusion, not least because the feature they simplify is also confusing. To understand the problem that is being solved, consider the `FilterByPrice` extension method that I defined in the previous section. This method is written so that it can filter `Product` objects by price, which means I must create a second method I want to filter by name, as shown in Listing 5-29.

Listing 5-29. Adding a Filter Method in the `MyExtensionMethods.cs` File in the Models Folder

```

using System.Collections.Generic;

namespace LanguageFeatures.Models {

    public static class MyExtensionMethods {

        public static decimal TotalPrices(this IEnumerable<Product> products) {
            decimal total = 0;
            foreach (Product prod in products) {
                total += prod?.Price ?? 0;
            }
            return total;
        }

        public static IEnumerable<Product> FilterByPrice(
            this IEnumerable<Product> productEnum, decimal minimumPrice) {

            foreach (Product prod in productEnum) {
                if ((prod?.Price ?? 0) >= minimumPrice) {
                    yield return prod;
                }
            }
        }
    }
}

```

```

public static IEnumerable<Product> FilterByName(
    this IEnumerable<Product> productEnum, char firstLetter) {

    foreach (Product prod in productEnum) {
        if (prod?.Name?[0] == firstLetter) {
            yield return prod;
        }
    }
}

```

Listing 5-30 shows the use of both filter methods applied in the controller to create two different totals.

Listing 5-30. Using Two Filter Methods in the HomeController.cs File in the Controllers Folder

```

using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public IActionResult Index() {

            Product[] productArray = {
                new Product {Name = "Kayak", Price = 275M},
                new Product {Name = "Lifejacket", Price = 48.95M},
                new Product {Name = "Soccer ball", Price = 19.50M},
                new Product {Name = "Corner flag", Price = 34.95M}
            };

            decimal priceFilterTotal = productArray.FilterByPrice(20).TotalPrices();
            decimal nameFilterTotal = productArray.FilterByName('S').TotalPrices();

            return View("Index", new string[] {
                $"Price Total: {priceFilterTotal:C2}",
                $"Name Total: {nameFilterTotal:C2}" });
        }
    }
}

```

The first filter selects all the products with a price of \$20 or more, and the second filter selects products whose name starts with the letter S. You will see the following output in the browser window if you run the example application:

```

Price Total: $358.90
Name Total: $19.50

```

Defining Functions

I can repeat this process indefinitely to create filter methods for every property and every combination of properties that I am interested in. A more elegant approach is to separate the code that processes the enumeration from the selection criteria. C# makes this easy by allowing functions to be passed around as objects. Listing 5-31 shows a single extension method that filters an enumeration of Product objects but that delegates the decision about which ones are included in the results to a separate function.

Listing 5-31. Creating a General Filter Method in the MyExtensionMethods.cs File in the Models Folder

```

using System.Collections.Generic;
using System;

namespace LanguageFeatures.Models {

    public static class MyExtensionMethods {

        public static decimal TotalPrices(this IEnumerable<Product> products) {
            decimal total = 0;
            foreach (Product prod in products) {
                total += prod?.Price ?? 0;
            }
            return total;
        }

        public static IEnumerable<Product> Filter(
            this IEnumerable<Product> productEnum,
            Func<Product, bool> selector) {

            foreach (Product prod in productEnum) {
                if (selector(prod)) {
                    yield return prod;
                }
            }
        }
    }
}

```

The second argument to the `Filter` method is a function that accepts a `Product` object and that returns a `bool` value. The `Filter` method calls the function for each `Product` object and includes it in the result if the function returns `true`. To use the `Filter` method, I can specify a method or create a stand-alone function, as shown in [Listing 5-32](#).

Listing 5-32. Using a Function to Filter Objects in the HomeController.cs File in the Controllers Folder

```

using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;
using System;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        bool FilterByPrice(Product p) {
            return (p?.Price ?? 0) >= 20;
        }

        public IActionResult Index() {

            Product[] productArray = {
                new Product {Name = "Kayak", Price = 275M},
                new Product {Name = "Lifejacket", Price = 48.95M},
                new Product {Name = "Soccer ball", Price = 19.50M},
                new Product {Name = "Corner flag", Price = 34.95M}
            };
        }
    }
}

```

```

    Func<Product, bool> nameFilter = delegate (Product prod) {
        return prod?.Name?[0] == 'S';
    };

    decimal priceFilterTotal = productArray
        .Filter(FilterByPrice)
        .TotalPrices();
    decimal nameFilterTotal = productArray
        .Filter(nameFilter)
        .TotalPrices();

    return View("Index", new string[] {
        $"Price Total: {priceFilterTotal:C2}",
        $"Name Total: {nameFilterTotal:C2}" });
    }
}

```

Neither approach is ideal. Defining methods like `FilterByPrice` clutters up a class definition. Creating a `Func<Product, bool>` object avoids this problem but uses an awkward syntax that is hard to read and hard to maintain. It is this issue that lambda expressions address by allowing functions to be defined in a more elegant and expressive way, as shown in Listing 5-33.

Listing 5-33. Using a Lambda Expression in the `HomeController.cs` File in the `Controllers` Folder

```

using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;
using System;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public IActionResult Index() {

            Product[] productArray = {
                new Product {Name = "Kayak", Price = 275M},
                new Product {Name = "Lifejacket", Price = 48.95M},
                new Product {Name = "Soccer ball", Price = 19.50M},
                new Product {Name = "Corner flag", Price = 34.95M}
            };

            decimal priceFilterTotal = productArray
                .Filter(p => (p?.Price ?? 0) >= 20)
                .TotalPrices();
            decimal nameFilterTotal = productArray
                .Filter(p => p?.Name?[0] == 'S')
                .TotalPrices();

            return View("Index", new string[] {
                $"Price Total: {priceFilterTotal:C2}",
                $"Name Total: {nameFilterTotal:C2}" });
        }
    }
}

```

The lambda expressions are shown in bold. The parameters are expressed without specifying a type, which will be inferred automatically. The => characters are read aloud as “goes to” and link the parameter to the result of the lambda expression. In my examples, a Product parameter called p goes to a bool result, which will be true if the Price property is equal or greater than 20 in the first expression or if the Name property starts with S in the second expression. This code works in the same way as the separate method and the function delegate but is more concise and is—for most people—easier to read.

OTHER FORMS FOR LAMBDA EXPRESSIONS

I don't need to express the logic of my delegate in the lambda expression. I can as easily call a method, like this:

```
...
prod => EvaluateProduct(prod)
...
```

If I need a lambda expression for a delegate that has multiple parameters, I must wrap the parameters in parentheses, like this:

```
...
(prod, count) => prod.Price > 20 && count > 0
...
```

Finally, if I need logic in the lambda expression that requires more than one statement, I can do so by using braces ({}) and finishing with a return statement, like this:

```
...
(prod, count) => {
    // ...multiple code statements...
    return result;
}
...
```

You do not need to use lambda expressions in your code, but they are a neat way of expressing complex functions simply and in a manner that is readable and clear. I like them a lot, and you will see them used throughout this book.

Using Lambda Expression Methods and Properties

Lambda expressions can be used to implement constructors, methods, and properties. In ASP.NET Core development, you will often end up with methods that contain a single statement that selects the data to display and the view to render. In Listing 5-34, I have rewritten the Index action method so that it follows this common pattern.

Listing 5-34. Creating a Common Action Pattern in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;
using System;
using System.Linq;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public IActionResult Index() {
            return View(Product.GetProducts().Select(p => p?.Name));
        }
    }
}
```

The action method gets a collection of `Product` objects from the static `Product.GetProducts` method and uses LINQ to project the values of the `Name` properties, which are then used as the view model for the default view. If you run the application, you will see the following output displayed in the browser window:

```
Kayak
Lifejacket
```

There will be an empty list item in the browser window as well because the `GetProducts` method includes a `null` reference in its results, but that doesn't matter for this section of the chapter.

When a constructor or method body consists of a single statement, it can be rewritten as a lambda expression, as shown in Listing 5-35.

Listing 5-35. A Lambda Action Method in the `HomeController.cs` File in the `Controllers` Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;
using System;
using System.Linq;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public IActionResult Index() =>
            View(Product.GetProducts().Select(p => p?.Name));
    }
}
```

Lambda expressions for methods omit the `return` keyword and use `=>` (goes to) to associate the method signature (including its arguments) with its implementation. The `Index` method shown in Listing 5-35 works in the same way as the one shown in Listing 5-34 but is expressed more concisely. The same basic approach can also be used to define properties. Listing 5-36 shows the addition of a property that uses a lambda expression to the `Product` class.

Listing 5-36. A Lambda Property in the `Product.cs` File in the `Models` Folder

```
namespace LanguageFeatures.Models {
    public class Product {

        public Product(bool stock = true) {
            InStock = stock;
        }

        public string Name { get; set; }
        public string Category { get; set; } = "Watersports";
        public decimal? Price { get; set; }
        public Product Related { get; set; }
        public bool InStock { get; }
        public bool NameBeginsWithS => Name?[0] == 'S';

        public static Product[] GetProducts() {

            Product kayak = new Product {
                Name = "Kayak",
                Category = "Water Craft",
                Price = 275M
            };
        }
    }
}
```

```

        Product lifejacket = new Product(false) {
            Name = "Lifejacket",
            Price = 48.95M
        };

        kayak.Related = lifejacket;

        return new Product[] { kayak, lifejacket, null };
    }
}

```

Using Type Inference and Anonymous Types

The `var` keyword allows you to define a local variable without explicitly specifying the variable type, as demonstrated by Listing 5-37. This is called *type inference*, or *implicit typing*.

Listing 5-37. Using Type Inference in the HomeController.cs File in the Controllers Folder

```

using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;
using System;
using System.Linq;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public IActionResult Index() {
            var names = new [] { "Kayak", "Lifejacket", "Soccer ball" };
            return View(names);
        }
    }
}

```

It is not that the `names` variable does not have a type; instead, I am asking the compiler to infer the type from the code. The compiler examines the array declaration and works out that it is a string array. Running the example produces the following output:

```

Kayak
Lifejacket
Soccer ball

```

Using Anonymous Types

By combining object initializers and type inference, I can create simple view model objects that are useful for transferring data between a controller and a view without having to define a class or struct, as shown in Listing 5-38.

Listing 5-38. Creating an Anonymous Type in the HomeController.cs File in the Controllers Folder

```

using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;
using System;
using System.Linq;

```

```

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public IActionResult Index() {
            var products = new [] {
                new { Name = "Kayak", Price = 275M },
                new { Name = "Lifejacket", Price = 48.95M },
                new { Name = "Soccer ball", Price = 19.50M },
                new { Name = "Corner flag", Price = 34.95M }
            };

            return View(products.Select(p => p.Name));
        }
    }
}

```

Each of the objects in the `products` array is an anonymously typed object. This does not mean that it is dynamic in the sense that JavaScript variables are dynamic. It just means that the type definition will be created automatically by the compiler. Strong typing is still enforced. You can get and set only the properties that have been defined in the initializer, for example. Restart ASP.NET Core and request `http://localhost:5000`, and you will see the following output in the browser window:

```

Kayak
Lifejacket
Soccer ball
Corner flag

```

The C# compiler generates the class based on the name and type of the parameters in the initializer. Two anonymously typed objects that have the same property names and types will be assigned to the same automatically generated class. This means that all the objects in the `products` array will have the same type because they define the same properties.

■ **Tip** I have to use the `var` keyword to define the array of anonymously typed objects because the type isn't created until the code is compiled, so I don't know the name of the type to use. The elements in an array of anonymously typed objects must all define the same properties; otherwise, the compiler can't work out what the array type should be.

To demonstrate this, I have changed the output from the example in Listing 5-39 so that it shows the type name rather than the value of the `Name` property.

Listing 5-39. Displaying the Type Name in the `HomeController.cs` File in the `Controllers` Folder

```

using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;
using System;
using System.Linq;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public IActionResult Index() {
            var products = new [] {
                new { Name = "Kayak", Price = 275M },
                new { Name = "Lifejacket", Price = 48.95M },
            };

```



```

        new { Name = "Soccer ball", Price = 19.50M },
        new { Name = "Corner flag", Price = 34.95M }
    };

    return View(products.Select(p => p.GetType().Name));
}
}
}

```

All the objects in the array have been assigned the same type, which you can see if you run the example. The type name isn't user-friendly but isn't intended to be used directly, and you may see a different name than the one shown in the following output:

```

<>f__AnonymousType0`2
<>f__AnonymousType0`2
<>f__AnonymousType0`2
<>f__AnonymousType0`2

```

Using Default Implementations in Interfaces

C# 8.0 introduces the ability to define default implementations for properties and methods defined by interfaces. This may seem like an odd feature because interfaces are intended to be a description of features without specifying an implementation, but this addition to C# makes it possible to update interfaces without breaking the existing implementations of them.

Add a class file named `IProductSelection.cs` to the `Models` folder and use it to define the interface shown in Listing 5-40.

Listing 5-40. The Contents of the `IProductSelection.cs` File in the `Models` Folder

```

using System.Collections.Generic;

namespace LanguageFeatures.Models {
    public interface IProductSelection {

        IEnumerable<Product> Products { get; }
    }
}

```

Update the `ShoppingCart` class to implement the new interface, as shown in Listing 5-41.

Listing 5-41. Implementing an Interface in the `ShoppingCart.cs` File in the `Models` Folder

```

using System.Collections;
using System.Collections.Generic;

namespace LanguageFeatures.Models {

    public class ShoppingCart : IProductSelection {
        private List<Product> products = new List<Product>();

        public ShoppingCart(params Product[] prods) {
            products.AddRange(prods);
        }

        public IEnumerable<Product> Products { get => products; }
    }
}

```

Listing 5-42 updates the Home controller so that it uses the ShoppingCart class.

Listing 5-42. Using an Interface in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;
using System;
using System.Linq;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public IActionResult Index() {
            IProductSelection cart = new ShoppingCart(
                new Product { Name = "Kayak", Price = 275M },
                new Product { Name = "Lifejacket", Price = 48.95M },
                new Product { Name = "Soccer ball", Price = 19.50M },
                new Product { Name = "Corner flag", Price = 34.95M }
            );
            return View(cart.Products.Select(p => p.Name));
        }
    }
}
```

This is the familiar use of an interface, and if you restart ASP.NET Core and request `http://localhost:5000`, you will see the following output in the browser:

```
Kayak
Lifejacket
Soccer ball
Corner flag
```

If I want to add a new feature to the interface, I must locate and update all the classes that implement it, which can be difficult, especially if an interface is used by other development teams in their projects. This is where the default implementation feature can be used, allowing new features to be added to an interface, as shown in Listing 5-43.

Listing 5-43. Adding a Feature in the IProductSelection.cs File in the Models Folder

```
using System.Collections.Generic;
using System.Linq;

namespace LanguageFeatures.Models {
    public interface IProductSelection {

        IEnumerable<Product> Products { get; }

        IEnumerable<string> Names => Products.Select(p => p.Name);
    }
}
```

The listing defines a Names property and provides a default implementation, which means that consumers of the IProductSelection interface can use the Total property even if it isn't defined by implementation classes, as shown in Listing 5-44.

Listing 5-44. Using a Default Implementation in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;
using System;
using System.Linq;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public IActionResult Index() {

            IProductSelection cart = new ShoppingCart(
                new Product { Name = "Kayak", Price = 275M },
                new Product { Name = "Lifejacket", Price = 48.95M },
                new Product { Name = "Soccer ball", Price = 19.50M },
                new Product { Name = "Corner flag", Price = 34.95M }
            );
            return View(cart.Names);
        }
    }
}
```

The ShoppingCart class has not been modified, but the Index method is able to use the default implementation of the Names property. Restart ASP.NET Core and request `http://localhost:5000`, and you will see the following output in the browser:

```
Kayak
Lifejacket
Soccer ball
Corner flag
```

Using Asynchronous Methods

Asynchronous methods perform work in the background and notify you when they are complete, allowing your code to take care of other business while the background work is performed. Asynchronous methods are an important tool in removing bottlenecks from code and allow applications to take advantage of multiple processors and processor cores to perform work in parallel.

In ASP.NET Core, asynchronous methods can be used to improve the overall performance of an application by allowing the server more flexibility in the way that requests are scheduled and executed. Two C# keywords—`async` and `await`—are used to perform work asynchronously.

Working with Tasks Directly

C# and .NET have excellent support for asynchronous methods, but the code has tended to be verbose, and developers who are not used to parallel programming often get bogged down by the unusual syntax. To create an example, add a class file called `MyAsyncMethods.cs` to the `Models` folder and add the code shown in Listing 5-45.

Listing 5-45. The Contents of the MyAsyncMethods.cs File in the Models Folder

```
using System.Net.Http;
using System.Threading.Tasks;

namespace LanguageFeatures.Models {
```

```

public class MyAsyncMethods {

    public static Task<long?> GetPageLength() {
        HttpClient client = new HttpClient();
        var httpTask = client.GetAsync("http://apress.com");
        return httpTask.ContinueWith((Task<HttpResponseMessage> antecedent) => {
            return antecedent.Result.Content.Headers.ContentLength;
        });
    }
}

```

This method uses a `System.Net.Http.HttpClient` object to request the contents of the Apress home page and returns its length. .NET represents work that will be done asynchronously as a `Task`. `Task` objects are strongly typed based on the result that the background work produces. So, when I call the `HttpClient.GetAsync` method, what I get back is a `Task<HttpResponseMessage>`. This tells me that the request will be performed in the background and that the result of the request will be an `HttpResponseMessage` object.

■ **Tip** When I use words like *background*, I am skipping over a lot of detail to make just the key points that are important to the world of ASP.NET Core. The .NET support for asynchronous methods and parallel programming is excellent, and I encourage you to learn more about it if you want to create truly high-performing applications that can take advantage of multicore and multiprocessor hardware. You will see how ASP.NET Core makes it easy to create asynchronous web applications throughout this book as I introduce different features.

The part that most programmers get bogged down with is the *continuation*, which is the mechanism by which you specify what you want to happen when the task is complete. In the example, I have used the `ContinueWith` method to process the `HttpResponseMessage` object I get from the `HttpClient.GetAsync` method, which I do with a lambda expression that returns the value of a property that contains the length of the content I get from the Apress web server. Here is the continuation code:

```

...
return httpTask.ContinueWith((Task<HttpResponseMessage> antecedent) => {
    return antecedent.Result.Content.Headers.ContentLength;
});
...

```

Notice that I use the `return` keyword twice. This is the part that causes confusion. The first use of the `return` keyword specifies that I am returning a `Task<HttpResponseMessage>` object, which, when the task is complete, will return the length of the `ContentLength` header. The `ContentLength` header returns a `long?` result (a nullable long value), and this means that the result of my `GetPageLength` method is `Task<long?>`, like this:

```

...
public static Task<long?> GetPageLength() {
...

```

Do not worry if this does not make sense—you are not alone in your confusion. It is for this reason that Microsoft added keywords to C# to simplify asynchronous methods.

Applying the `async` and `await` Keywords

Microsoft introduced two keywords to C# that simplify using asynchronous methods like `HttpClient.GetAsync`. The keywords are `async` and `await`, and you can see how I have used them to simplify my example method in Listing 5-46.

Listing 5-46. Using the `async` and `await` Keywords in the `MyAsyncMethods.cs` File in the `Models` Folder

```
using System.Net.Http;
using System.Threading.Tasks;

namespace LanguageFeatures.Models {

    public class MyAsyncMethods {

        public async static Task<long?> GetPageLength() {
            HttpClient client = new HttpClient();
            var httpMessage = await client.GetAsync("http://apress.com");
            return httpMessage.Content.Headers.ContentLength;
        }
    }
}
```

I used the `await` keyword when calling the asynchronous method. This tells the C# compiler that I want to wait for the result of the `Task` that the `GetAsync` method returns and then carry on executing other statements in the same method.

Applying the `await` keyword means I can treat the result from the `GetAsync` method as though it were a regular method and just assign the `HttpResponseMessage` object that it returns to a variable. Even better, I can then use the `return` keyword in the normal way to produce a result from another method—in this case, the value of the `ContentLength` property. This is a much more natural technique, and it means I do not have to worry about the `ContinueWith` method and multiple uses of the `return` keyword.

When you use the `await` keyword, you must also add the `async` keyword to the method signature, as I have done in the example. The method result type does not change—my example `GetPageLength` method still returns a `Task<long?>`. This is because `await` and `async` are implemented using some clever compiler tricks, meaning that they allow a more natural syntax, but they do not change what is happening in the methods to which they are applied. Someone who is calling my `GetPageLength` method still has to deal with a `Task<long?>` result because there is still a background operation that produces a nullable `long`—although, of course, that programmer can also choose to use the `await` and `async` keywords as well.

This pattern follows through into the controller, which makes it easy to write asynchronous action methods, as shown in Listing 5-47.

■ **Note** You can also use the `async` and `await` keywords in lambda expressions, which I demonstrate in later chapters.

Listing 5-47. An Asynchronous Action Methods in the `HomeController.cs` File in the `Controllers` Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;
using System;
using System.Linq;
using System.Threading.Tasks;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public async Task<ViewResult> Index() {
            long? length = await MyAsyncMethods.GetPageLength();
            return View(new string[] { $"Length: {length}" });
        }
    }
}
```

I have changed the result of the Index action method to `Task<ViewResult>`, which declares that the action method will return a Task that will produce a `ViewResult` object when it completes, which will provide details of the view that should be rendered and the data that it requires. I have added the `async` keyword to the method's definition, which allows me to use the `await` keyword when calling the `MyAsyncMethods.GetPathLength` method. .NET takes care of dealing with the continuations, and the result is asynchronous code that is easy to write, easy to read, and easy to maintain. Restart ASP.NET Core and request `http://localhost:5000`, and you will see output similar to the following (although with a different length since the content of the Apress web site changes often):

Length: 101868

Using an Asynchronous Enumerable

An asynchronous enumerable describes a sequence of values that will be generated over time. To demonstrate the issue that this feature addresses, Listing 5-48 adds a method to the `MyAsyncMethods` class.

Listing 5-48. Adding a Method in the `MyAsyncMethods.cs` File in the Models Folder

```
using System.Net.Http;
using System.Threading.Tasks;
using System.Collections.Generic;

namespace LanguageFeatures.Models {

    public class MyAsyncMethods {

        public async static Task<long?> GetPageLength() {
            HttpClient client = new HttpClient();
            var httpMessage = await client.GetAsync("http://apress.com");
            return httpMessage.Content.Headers.ContentLength;
        }

        public static async Task<IEnumerable<long?>>
            GetPageLengths(List<string> output, params string[] urls) {
            List<long?> results = new List<long?>();
            HttpClient client = new HttpClient();
            foreach (string url in urls) {
                output.Add($"Started request for {url}");
                var httpMessage = await client.GetAsync($"http://{url}");
                results.Add(httpMessage.Content.Headers.ContentLength);
                output.Add($"Completed request for {url}");
            }
            return results;
        }
    }
}
```

The `GetPageLengths` method makes HTTP requests to a series of web sites and gets their length. The requests are performed asynchronously, but there is no way to feed the results back to the method's caller as they arrive. Instead, the method waits until all the requests are complete and then returns all of the results in one go. In addition to the URLs that will be requested, this method accepts a `List<string>` to which I add messages in order to highlight how the code works. Listing 5-49 updates the Index action method of the Home controller to use the new method.

Listing 5-49. Using the New Method in the HomeController.cs File in the Controllers Folder

```

using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;
using System;
using System.Linq;
using System.Threading.Tasks;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public async Task<ViewResult> Index() {
            List<string> output = new List<string>();
            foreach(long? len in await MyAsyncMethods.GetPageLengths(output,
                "apress.com", "microsoft.com", "amazon.com")) {
                output.Add($"Page length: { len}");
            }
            return View(output);
        }
    }
}

```

The action method enumerates the sequence produced by the `GetPageLengths` method and adds each result to the `List<string>` object, which produces an ordered sequence of messages showing the interaction between the `foreach` loop in the `Index` method that processes the results and the `foreach` loop in the `GetPageLengths` method that generates them. Restart ASP.NET Core and request `http://localhost:5000`, and you will see the following output in the browser (which may take several seconds to appear and may have different page lengths):

```

Started request for apress.com
Completed request for apress.com
Started request for microsoft.com
Completed request for microsoft.com
Started request for amazon.com
Completed request for amazon.com
Page length: 101868
Page length: 159158
Page length: 91879

```

You can see that the `Index` action method doesn't receive the results until all the HTTP requests have been completed. This is the problem that the asynchronous enumerable feature solves, as shown in Listing 5-50.

Listing 5-50. Using an Asynchronous Enumerable in the MyAsyncMethods.cs File in the Models Folder

```

using System.Net.Http;
using System.Threading.Tasks;
using System.Collections.Generic;

namespace LanguageFeatures.Models {

    public class MyAsyncMethods {

        public async static Task<long?> GetPageLength() {
            HttpClient client = new HttpClient();
            var httpMessage = await client.GetAsync("http://apress.com");
            return httpMessage.Content.Headers.ContentLength;
        }
    }
}

```

```

public static async IEnumerable<long?>
    GetPageLengths(List<string> output, params string[] urls) {
    HttpClient client = new HttpClient();
    foreach (string url in urls) {
        output.Add($"Started request for {url}");
        var httpMessage = await client.GetAsync($"http://{url}");
        output.Add($"Completed request for {url}");
        yield return httpMessage.Content.Headers.ContentLength;
    }
}

```

The methods result is `IAsyncEnumerable<long?>`, which denotes an asynchronous sequence of nullable long values. This result type has special support in .NET Core and works with standard `yield return` statements, which isn't otherwise possible because the result constraints for asynchronous methods conflict with the `yield` keyword. Listing 5-51 updates the controller to use the revised method.

Listing 5-51. Using an Asynchronous Enumerable in the HomeController.cs File in the Controllers Folder

```

using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;
using System;
using System.Linq;
using System.Threading.Tasks;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public async Task<ViewResult> Index() {
            List<string> output = new List<string>();
            await foreach(long? len in MyAsyncMethods.GetPageLengths(output,
                "apress.com", "microsoft.com", "amazon.com")) {
                output.Add($"Page length: { len}");
            }
            return View(output);
        }
    }
}

```

The difference is that the `await` keyword is applied before the `foreach` keyword and not before the call to the `async` method. Restart ASP.NET Core and request `http://localhost:5000`; once the HTTP requests are complete, you will see that the order of the response messages has changed, like this:

```

Started request for apress.com
Completed request for apress.com
Page length: 101868
Started request for microsoft.com
Completed request for microsoft.com
Page length: 159160
Started request for amazon.com
Completed request for amazon.com
Page length: 91674

```

The controller receives the next result in the sequence as it is produced. As I explain in Chapter 19, ASP.NET Core has special support for using `IAsyncEnumerable<T>` results in web services, allowing data values to be serialized as the values in the sequence are generated.

Getting Names

There are many tasks in web application development in which you need to refer to the name of an argument, variable, method, or class. Common examples include when you throw an exception or create a validation error when processing input from the user. The traditional approach has been to use a string value hard-coded with the name, as shown in Listing 5-52.

Listing 5-52. Hard-Coding a Name in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;
using System;
using System.Linq;
using System.Threading.Tasks;

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public IActionResult Index() {
            var products = new[] {
                new { Name = "Kayak", Price = 275M },
                new { Name = "Lifejacket", Price = 48.95M },
                new { Name = "Soccer ball", Price = 19.50M },
                new { Name = "Corner flag", Price = 34.95M }
            };
            return View(products.Select(p => $"Name: {p.Name}, Price: {p.Price}"));
        }
    }
}
```

The call to the LINQ `Select` method generates a sequence of strings, each of which contains a hard-coded reference to the `Name` and `Price` properties. Restart ASP.NET Core and request `http://localhost:5000`, and you will see the following output in the browser window:

```
Name: Kayak, Price: 275
Name: Lifejacket, Price: 48.95
Name: Soccer ball, Price: 19.50
Name: Corner flag, Price: 34.95
```

This approach is prone to errors, either because the name was mistyped or because the code was refactored and the name in the string isn't correctly updated. C# supports the `nameof` expression, in which the compiler takes responsibility for producing a name string, as shown in Listing 5-53.

Listing 5-53. Using `nameof` Expressions in the HomeController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using LanguageFeatures.Models;
using System;
using System.Linq;
using System.Threading.Tasks;
```

```

namespace LanguageFeatures.Controllers {
    public class HomeController : Controller {

        public IActionResult Index() {
            var products = new[] {
                new { Name = "Kayak", Price = 275M },
                new { Name = "Lifejacket", Price = 48.95M },
                new { Name = "Soccer ball", Price = 19.50M },
                new { Name = "Corner flag", Price = 34.95M }
            };
            return View(products.Select(p =>
                $"{nameof(p.Name)}: {p.Name}, {nameof(p.Price)}: {p.Price}"));
        }
    }
}

```

The compiler processes a reference such as `p.Name` so that only the last part is included in the string, producing the same output as in previous examples. There is IntelliSense support for `nameof` expressions, so you will be prompted to select references, and expressions will be correctly updated when you refactor code. Since the compiler is responsible for dealing with `nameof`, using an invalid reference causes a compiler error, which prevents incorrect or outdated references from escaping notice.

Summary

In this chapter, I gave you an overview of the key C# language features that an effective ASP.NET Core programmer needs to know. C# is a sufficiently flexible language that there are usually different ways to approach any problem, but these are the features that you will encounter most often during web application development and that you will see throughout the examples in this book. In the next chapter, I explain how to set up a unit test project for ASP.NET Core.

CHAPTER 6



Testing ASP.NET Core Applications

In this chapter, I demonstrate how to unit test ASP.NET Core applications. Unit testing is a form of testing in which individual components are isolated from the rest of the application so their behavior can be thoroughly validated. ASP.NET Core has been designed to make it easy to create unit tests, and there is support for a wide range of unit testing frameworks. I show you how to set up a unit test project and describe the process for writing and running tests. Table 6-1 summarizes the chapter.

DECIDING WHETHER TO UNIT TEST

Being able to easily perform unit testing is one of the benefits of using ASP.NET Core, but it isn't for everyone, and I have no intention of pretending otherwise.

I like unit testing, and I use it in my own projects, but not all of them and not as consistently as you might expect. I tend to focus on writing unit tests for features and functions that I know will be hard to write and likely to be the source of bugs in deployment. In these situations, unit testing helps structure my thoughts about how to best implement what I need. I find that just thinking about what I need to test helps produce ideas about potential problems, and that's before I start dealing with actual bugs and defects.

That said, unit testing is a tool and not a religion, and only you know how much testing you require. If you don't find unit testing useful or if you have a different methodology that suits you better, then don't feel you need to unit test just because it is fashionable. (However, if you *don't* have a better methodology and you are not testing at all, then you are probably letting users find your bugs, which is rarely ideal. You don't *have* to unit test, but you really should consider doing *some* testing of *some* kind.)

If you have not encountered unit testing before, then I encourage you to give it a try to see how it works. If you are not a fan of unit testing, then you can skip this chapter and move on to Chapter 7, where I start to build a more realistic ASP.NET Core application.

Table 6-1. Chapter Summary

Problem	Solution	Listing
Creating a unit test project	Use the <code>dotnet new</code> command with the project template for your preferred test framework	7
Creating an XUnit test	Create a class with methods decorated with the <code>Fact</code> attribute and use the <code>Assert</code> class to inspect the test results	9
Running unit tests	Use the Visual Studio or Visual Studio Code test runners or use the <code>dotnet test</code> command	11
Isolating a component for testing	Create mock implementations of the objects that the component under test requires	12-19

Preparing for This Chapter

To prepare for this chapter, I need to create a simple ASP.NET Core project. Open a new PowerShell command prompt using the Windows Start menu, navigate to a convenient location, and run the commands shown in Listing 6-1.

■ **Tip** You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/apress/pro-asp.net-core-3>. See Chapter 1 for how to get help if you have problems running the examples.

Listing 6-1. Creating the Example Project

```
dotnet new globaljson --sdk-version 3.1.101 --output Testing/SimpleApp
dotnet new web --no-https --output Testing/SimpleApp --framework netcoreapp3.1
dotnet new sln -o Testing

dotnet sln Testing add Testing/SimpleApp
```

These commands create a new project named SimpleApp using the web template, which contains the minimal configuration for ASP.NET Core applications. The project folder is contained within a solution folder also called Testing.

Opening the Project

If you are using Visual Studio, select File ► Open ► Project/Solution, select the Testing.sln file in the Testing folder, and click the Open button to open the solution file and the project it references. If you are using Visual Studio Code, select File ► Open Folder, navigate to the Testing folder, and click the Select Folder button.

Selecting the HTTP Port

If you are using Visual Studio, select Project ► SimpleApp Properties, select the Debug section, and change the HTTP port to 5000 in the App URL field, as shown in Figure 6-1. Select File ► Save All to save the new port. (This change is not required if you are using Visual Studio Code.)

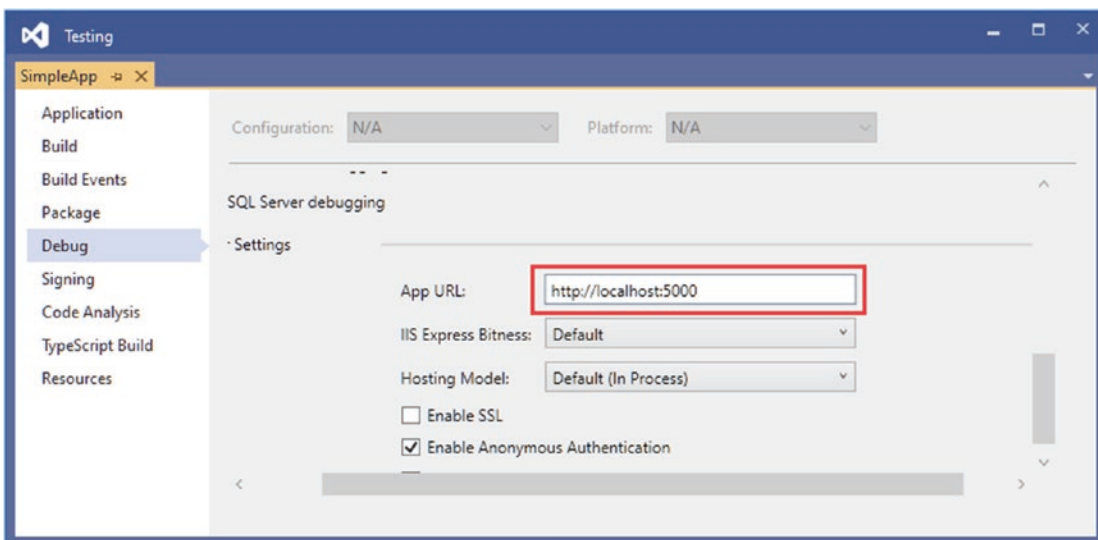


Figure 6-1. Setting the HTTP port

Enabling the MVC Framework

As I explained in Chapter 1, ASP.NET Core supports different application frameworks, but I am going to continue using the MVC Framework in this chapter. I introduce the other frameworks in the SportsStore application that I start to build in Chapter 7, but for the moment, the MVC Framework gives me a foundation for demonstrating how to perform unit testing that is familiar from earlier examples. Add the statements shown in Listing 6-2 to the Startup.cs file in the SimpleApp folder.

Listing 6-2. Enabling the MVC Framework in the Startup.cs File in the SimpleApp Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

namespace SimpleApp {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.AddControllersWithViews();
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
            if (env.IsDevelopment()) {
                app.UseDeveloperExceptionPage();
            }

            app.UseRouting();

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

Creating the Application Components

Now that the MVC Framework is set up, I can add the application components that I will use to demonstrate important C# language features.

Creating the Data Model

I started by creating a simple model class so that I can have some data to work with. I added a folder called Models and created a class file called Product.cs within it, which I used to define the class shown in Listing 6-3.

Listing 6-3. The Contents of the Product.cs File in the SimpleApp/Models Folder

```
namespace SimpleApp.Models {
    public class Product {

        public string Name { get; set; }
        public decimal? Price { get; set; }
    }
}
```

```

public static Product[] GetProducts() {
    Product kayak = new Product {
        Name = "Kayak", Price = 275M
    };

    Product lifejacket = new Product {
        Name = "Lifejacket", Price = 48.95M
    };

    return new Product[] { kayak, lifejacket };
}
}
}

```

The `Product` class defines `Name` and `Price` properties, and there is a static method called `GetProducts` that returns a `Products` array.

Creating the Controller and View

For the examples in this chapter, I use a simple controller class to demonstrate different language features. I created a `Controllers` folder and added to it a class file called `HomeController.cs`, the contents of which are shown in Listing 6-4.

Listing 6-4. The Contents of the `HomeController.cs` File in the `SimpleApp/Controllers` Folder

```

using Microsoft.AspNetCore.Mvc;
using SimpleApp.Models;

namespace SimpleApp.Controllers {
    public class HomeController : Controller {

        public IActionResult Index() {
            return View(Product.GetProducts());
        }
    }
}

```

The `Index` action method tells ASP.NET Core to render the default view and provides it with the `Product` objects obtained from the static `Product.GetProducts` method. To create the view for the action method, I added a `Views/Home` folder (by creating a `Views` folder and then adding a `Home` folder within it) and added a Razor View called `Index.cshtml`, with the contents shown in Listing 6-5.

Listing 6-5. The Contents of the `Index.cshtml` File in the `SimpleApp/Views/Home` Folder

```

@using SimpleApp.Models
@model IEnumerable<Product>
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Simple App</title>
</head>

```

```

<body>
  <ul>
    @foreach (Product p in Model) {
      <li>Name: @p.Name, Price: @p.Price</li>
    }
  </ul>
</body>
</html>

```

Running the Example Application

Start ASP.NET Core by selecting Start Without Debugging (Visual Studio) or Run Without Debugging (Visual Studio Code) from the Debug menu or by running the command shown in Listing 6-6 in the SimpleApp folder.

Listing 6-6. Running the Example Application

```
dotnet run
```

Request `http://localhost:5000`, and you will see the output shown in Figure 6-2.

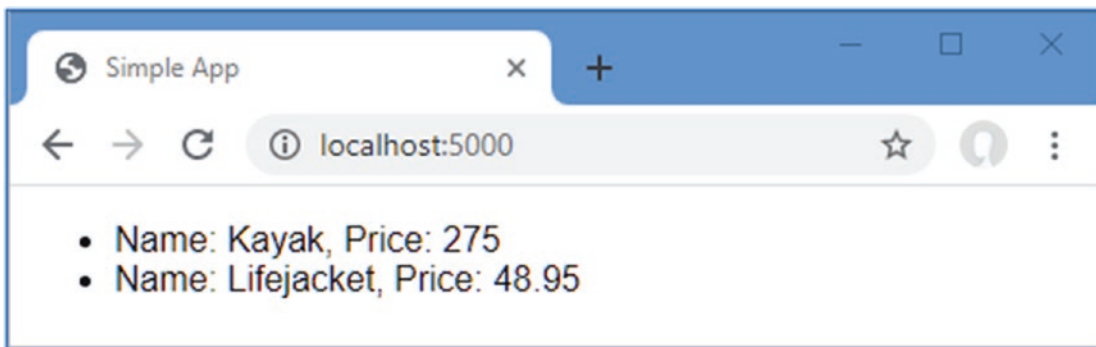


Figure 6-2. Running the example application

Creating a Unit Test Project

For ASP.NET Core applications, you generally create a separate Visual Studio project to hold the unit tests, each of which is defined as a method in a C# class. Using a separate project means you can deploy your application without also deploying the tests. The .NET Core SDK includes templates for unit test projects using three popular test tools, as described in Table 6-2.

Table 6-2. The Unit Test Project Tools

Name	Description
mstest	This template creates a project configured for the MS Test framework, which is produced by Microsoft.
nunit	This template creates a project configured for the NUnit framework.
xunit	This template creates a project configured for the XUnit framework.

These testing frameworks have largely the same feature set and differ only in how they are implemented and how they integrate into third-party testing environments. I recommend starting with XUnit if you do not have an established preference, largely because it is the test framework that I find easiest to work with.

The convention is to name the unit test project `<ApplicationName>.Tests`. Run the commands shown in Listing 6-7 in the Testing folder to create the XUnit test project named `SimpleApp.Tests`, add it to the solution file, and create a reference between projects so the unit tests can be applied to the classes defined in the `SimpleApp` project.

Listing 6-7. Creating the Unit Test Project

```
dotnet new xunit -o SimpleApp.Tests --framework netcoreapp3.1
dotnet sln add SimpleApp.Tests
dotnet add SimpleApp.Tests reference SimpleApp
```

If you are using Visual Studio, you will be prompted to reload the solution, which will cause the new unit test project to be displayed in the Solution Explorer, alongside the existing project. You may find that Visual Studio Code doesn't build the new project. If that happens, select Terminal ► Configure Default Build Task, select "build" from the list, and, if prompted, select .NET Core from the list of environments.

Removing the Default Test Class

The project template adds a C# class file to the test project, which will confuse the results of later examples. Either delete the `UnitTest1.cs` file from the `SimpleApp.Tests` folder using the Solution Explorer or File Explorer pane or run the command shown in Listing 6-8 in the Testing folder.

Listing 6-8. Removing the Default Test Class File

```
Remove-Item SimpleApp.Tests/UnitTest1.cs
```

Writing and Running Unit Tests

Now that all the preparation is complete, I can write some tests. To get started, I added a class file called `ProductTests.cs` to the `SimpleApp.Tests` project and used it to define the class shown in Listing 6-9. This is a simple class, but it contains everything required to get started with unit testing.

■ **Note** The `CanChangeProductPrice` method contains a deliberate error that I resolve later in this section.

Listing 6-9. The Contents of the `ProductTests.cs` File in the `SimpleApp.Tests` Folder

```
using SimpleApp.Models;
using Xunit;

namespace SimpleApp.Tests {
    public class ProductTests {
        [Fact]
        public void CanChangeProductName() {
            // Arrange
            var p = new Product { Name = "Test", Price = 100M };

            // Act
            p.Name = "New Name";

            //Assert
            Assert.Equal("New Name", p.Name);
        }
    }
}
```



```

[Fact]
public void CanChangeProductPrice() {

    // Arrange
    var p = new Product { Name = "Test", Price = 100M };

    // Act
    p.Price = 200M;

    //Assert
    Assert.Equal(100M, p.Price);
}
}
}

```

There are two unit tests in the `ProductTests` class, each of which tests a behavior of the `Product` model class from the `SimpleApp` project. A test project can contain many classes, each of which can contain many unit tests.

Conventionally, the name of the test methods describes what the test does, and the name of the class describes what is being tested. This makes it easier to structure the tests in a project and to understand what the results of all the tests are when they are run by Visual Studio. The name `ProductTests` indicates that the class contains tests for the `Product` class, and the method names indicate that they test the ability to change the name and price of a `Product` object.

The `Fact` attribute is applied to each method to indicate that it is a test. Within the method body, a unit test follows a pattern called *arrange, act, assert* (A/A/A). *Arrange* refers to setting up the conditions for the test, *act* refers to performing the test, and *assert* refers to verifying that the result was the one that was expected.

The *arrange* and *act* sections of these tests are regular C# code, but the *assert* section is handled by `xUnit.net`, which provides a class called `Assert`, whose methods are used to check that the outcome of an action is the one that is expected.

■ **Tip** The `Fact` attribute and the `Assert` class are defined in the `Xunit` namespace, for which there must be a `using` statement in every test class.

The methods of the `Assert` class are static and are used to perform different kinds of comparison between the expected and actual results. Table 6-3 shows the commonly used `Assert` methods.

Table 6-3. Commonly Used `xUnit.net` `Assert` Methods

Name	Description
<code>Equal(expected, result)</code>	This method asserts that the result is equal to the expected outcome. There are overloaded versions of this method for comparing different types and for comparing collections. There is also a version of this method that accepts an additional argument of an object that implements the <code>IEqualityComparer<T></code> interface for comparing objects.
<code>NotEqual(expected, result)</code>	This method asserts that the result is not equal to the expected outcome.
<code>True(result)</code>	This method asserts that the result is true.
<code>False(result)</code>	This method asserts that the result is false.
<code>IsType(expected, result)</code>	This method asserts that the result is of a specific type.
<code>IsNotType(expected, result)</code>	This method asserts that the result is not a specific type.
<code>IsNull(result)</code>	This method asserts that the result is null.
<code>IsNotNull(result)</code>	This method asserts that the result is not null.
<code>InRange(result, low, high)</code>	This method asserts that the result falls between low and high.
<code>NotInRange(result, low, high)</code>	This method asserts that the result falls outside low and high.
<code>Throws(exception, expression)</code>	This method asserts that the specified expression throws a specific exception type.

Each Assert method allows different types of comparison to be made and throws an exception if the result is not what was expected. The exception is used to indicate that a test has failed. In the tests in Listing 6-9, I used the Equal method to determine whether the value of a property has been changed correctly.

```
...
Assert.Equal("New Name", p.Name);
...
```

Running Tests with the Visual Studio Test Explorer

Visual Studio includes support for finding and running unit tests through the Test Explorer window, which is available through the Test ► Test Explorer menu and is shown in Figure 6-3.

■ **Tip** Build the solution if you don't see the unit tests in the Test Explorer window. Compilation triggers the process by which unit tests are discovered.

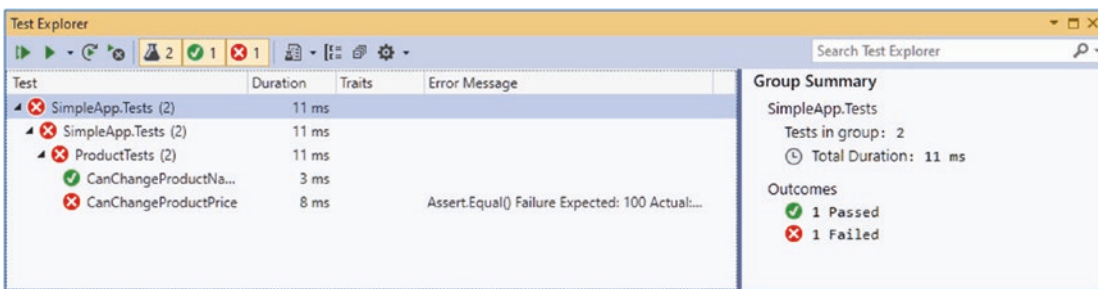


Figure 6-3. The Visual Studio Test Explorer

Run the tests by clicking the Run All Tests button in the Test Explorer window (it is the button that shows two arrows and is the first button in the row at the top of the window). As noted, the CanChangeProductPrice test contains an error that causes the test to fail, which is clearly indicated in the test results shown in the figure.

Running Tests with Visual Studio Code

Visual Studio Code detects tests and allows them to be run using the code lens feature, which displays details about code features in the editor. To run all the tests in the ProductTests class, click Run All Tests in the code editor when the unit test class is open, as shown in Figure 6-4.

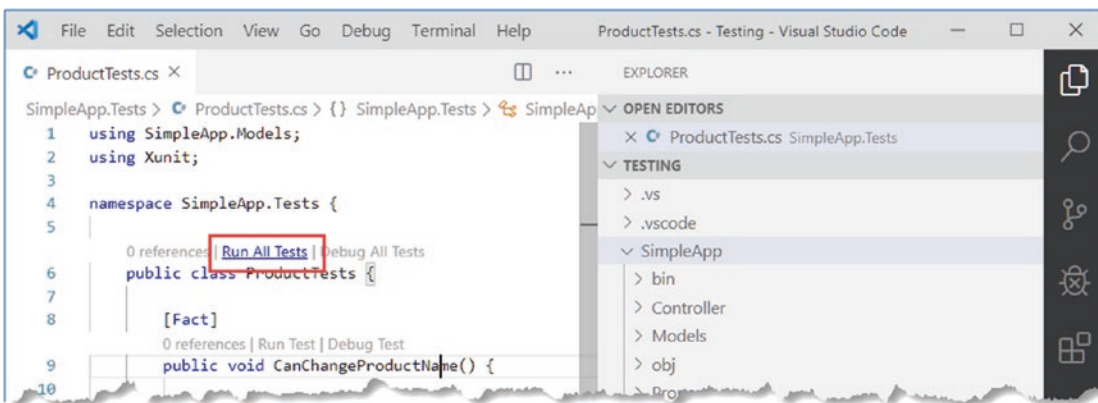


Figure 6-4. Running tests with the Visual Studio Code code lens feature

■ **Tip** Close and reopen the `Testing` folder in Visual Studio Code if you don't see the code lens test features.

Visual Studio Code runs the tests using the command-line tools that I describe in the following section, and the results are displayed as text in a terminal window.

Running Tests from the Command Line

To run the tests in the project, run the command shown in Listing 6-10 in the `Testing` folder.

Listing 6-10. Running Unit Tests

```
dotnet test
```

The tests are discovered and executed, producing the following results, which show the deliberate error that I introduced earlier:

```
Test run for C:\Users\adam\SimpleApp.Tests.dll(.NETCoreApp,Version=v3.1)
Microsoft (R) Test Execution Command Line Tool Version 16.3.0
Copyright (c) Microsoft Corporation. All rights reserved.
```

```
Starting test execution, please wait...
```

```
A total of 1 test files matched the specified pattern.
```

```
[xUnit.net 00:00:00.83] SimpleApp.Tests.ProductTests.CanChangeProductPrice [FAIL]
```

```
X SimpleApp.Tests.ProductTests.CanChangeProductPrice [6ms]
```

```
Error Message:
```

```
Assert.Equal() Failure
```

```
Expected: 100
```

```
Actual: 200
```

```
Stack Trace:
```

```
at SimpleApp.Tests.ProductTests.CanChangeProductPrice() in C:\Users\adam\Documents\Books\Pro ASP.NET Core
MVC 3\Source Code\Current\Testin
```

```
g\SimpleApp.Tests\ProductTests.cs:line 31
```

```
Test Run Failed.
```

```
Total tests: 2
```

```
Passed: 1
```

```
Failed: 1
```

```
Total time: 1.7201 Seconds
```

Correcting the Unit Test

The problem with the unit test is with the arguments to the `Assert.Equal` method, which compares the test result to the original `Price` property value rather than the value it has been changed to. Listing 6-11 corrects the problem.

■ **Tip** When a test fails, it is always a good idea to check the accuracy of the test before looking at the component it targets, especially if the test is new or has been recently modified.

Listing 6-11. Correcting a Test in the ProductTests.cs File in the SimpleApp.Tests Folder

```
using SimpleApp.Models;
using Xunit;

namespace SimpleApp.Tests {

    public class ProductTests {

        [Fact]
        public void CanChangeProductName() {

            // Arrange
            var p = new Product { Name = "Test", Price = 100M };

            // Act
            p.Name = "New Name";

            //Assert
            Assert.Equal("New Name", p.Name);
        }

        [Fact]
        public void CanChangeProductPrice() {

            // Arrange
            var p = new Product { Name = "Test", Price = 100M };

            // Act
            p.Price = 200M;

            //Assert
            Assert.Equal(200M, p.Price);
        }
    }
}
```

Run the tests again, and you will see they all pass. If you are using Visual Studio, you can click the Run Failed Tests button, which will execute only the tests that failed, as shown in Figure 6-5.

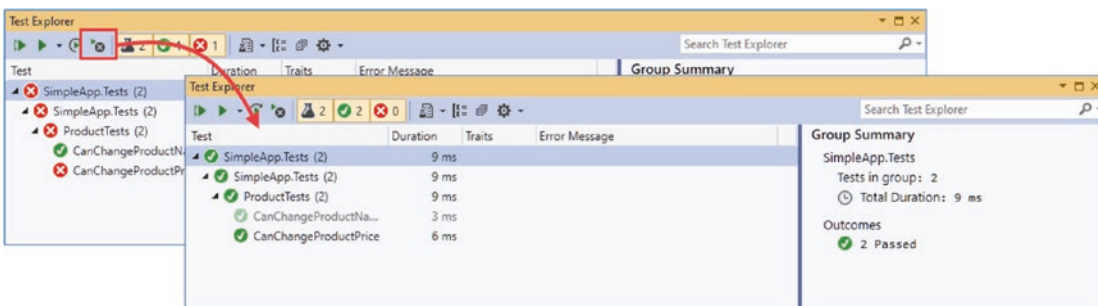


Figure 6-5. Running only failed tests

Isolating Components for Unit Testing

Writing unit tests for model classes like Product is easy. Not only is the Product class simple, but it is self-contained, which means that when I perform an action on a Product object, I can be confident that I am testing the functionality provided by the Product class.

The situation is more complicated with other components in an ASP.NET Core application because there are dependencies between them. The next set of tests that I define will operate on the controller, examining the sequence of `Product` objects that are passed between the controller and the view.

When comparing objects instantiated from custom classes, you will need to use the `xUnit.net.Assert.Equal` method that accepts an argument that implements the `IEqualityComparer<T>` interface so that the objects can be compared. My first step is to add a class file called `Comparer.cs` to the unit test project and use it to define the helper classes shown in Listing 6-12.

Listing 6-12. The Contents of the `Comparer.cs` File in the `SimpleApp.Tests` Folder

```
using System;
using System.Collections.Generic;

namespace SimpleApp.Tests {
    public class Comparer {
        public static Comparer<U> Get<U>(Func<U, U, bool> func) {
            return new Comparer<U>(func);
        }
    }

    public class Comparer<T> : Comparer, IEqualityComparer<T> {
        private Func<T, T, bool> comparisonFunction;

        public Comparer(Func<T, T, bool> func) {
            comparisonFunction = func;
        }

        public bool Equals(T x, T y) {
            return comparisonFunction(x, y);
        }

        public int GetHashCode(T obj) {
            return obj.GetHashCode();
        }
    }
}
```

These classes will allow me to create `IEqualityComparer<T>` objects using lambda expressions rather than having to define a new class for each type of comparison that I want to make. This isn't essential, but it will simplify the code in my unit test classes and make them easier to read and maintain.

Now that I can easily make comparisons, I can illustrate the problem of dependencies between components in the application. I added a new class called `HomeControllerTests.cs` to the `SimpleApp.Tests` folder and used it to define the unit test shown in Listing 6-13.

Listing 6-13. The `HomeControllerTests.cs` File in the `SimpleApp.Tests` Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using SimpleApp.Controllers;
using SimpleApp.Models;
using Xunit;

namespace SimpleApp.Tests {
    public class HomeControllerTests {

        [Fact]
        public void IndexActionModelIsComplete() {
            // Arrange
```

```

var controller = new HomeController();
Product[] products = new Product[] {
    new Product { Name = "Kayak", Price = 275M },
    new Product { Name = "Lifejacket", Price = 48.95M}
};

// Act
var model = (controller.Index() as ViewResult)?.ViewData.Model
    as IEnumerable<Product>;

// Assert
Assert.Equal(products, model,
    Comparer.Get<Product>((p1, p2) => p1.Name == p2.Name
        && p1.Price == p2.Price));
}
}
}

```

The unit test creates an array of `Product` objects and checks that they correspond to the ones the `Index` action method provides as the view model. (Ignore the `act` section of the test for the moment; I explain the `ViewResult` class in Chapters 21 and 22. For the moment, it is enough to know that I am getting the model data returned by the `Index` action method.)

The test passes, but it isn't a useful result because the `Product` data that I am testing is coming from the hardwired objects' `Product` class. I can't write a test to make sure that the controller behaves correctly when there are more than two `Product` objects, for example, or if the `Price` property of the first object has a decimal fraction. The overall effect is that I am testing the combined behavior of the `HomeController` and `Product` classes and only for the specific hardwired objects.

Unit tests are effective when they target small parts of an application, such as an individual method or class. What I need is the ability to isolate the `Home` controller from the rest of the application so that I can limit the scope of the test and rule out any impact caused by the repository.

Isolating a Component

The key to isolating components is to use C# interfaces. To separate the controller from the repository, I added a new class file called `IDataSource.cs` to the `Models` folder and used it to define the interface shown in Listing 6-14.

Listing 6-14. The Contents of the `IDataSource.cs` File in the `SimpleApp/Models` Folder

```

using System.Collections.Generic;

namespace SimpleApp.Models {
    public interface IDataSource {

        IEnumerable<Product> Products { get; }
    }
}

```

In Listing 6-15, I have removed the static method from the `Product` class and created a new class that implements the `IDataSource` interface.

Listing 6-15. Creating a Data Source in the `Product.cs` File in the `SimpleApp/Models` Folder

using System.Collections.Generic;

```

namespace SimpleApp.Models {

    public class Product {
        public string Name { get; set; }
        public decimal? Price { get; set; }
    }
}

```

```

public class ProductDataSource : IDataSource {
    public IEnumerable<Product> Products =>
        new Product[] {
            new Product { Name = "Kayak", Price = 275M },
            new Product { Name = "Lifejacket", Price = 48.95M }
        };
}
}

```

The next step is to modify the controller so that it uses the `ProductDataSource` class as the source for its data, as shown in Listing 6-16.

■ **Tip** ASP.NET Core supports a more elegant approach for solving this problem, known as *dependency injection*, which I describe in Chapter 14. Dependency injection often causes confusion, so I isolate components in a simpler and more manual way in this chapter.

Listing 6-16. Adding a Property in the `HomeController.cs` File in the `SimpleApp/Controllers` Folder

```

using Microsoft.AspNetCore.Mvc;
using SimpleApp.Models;

namespace SimpleApp.Controllers {
    public class HomeController : Controller {
        public IDataSource dataSource = new ProductDataSource();

        public IActionResult Index() {
            return View(dataSource.Products);
        }
    }
}

```

This may not seem like a significant change, but it allows me to change the data source the controller uses during testing, which is how I can isolate the controller. In Listing 6-17, I have updated the controller unit tests so they use a special version of the repository.

Listing 6-17. Isolating the Controller in the `HomeControllerTests.cs` File in the `SimpleApp.Tests` Folder

```

using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using SimpleApp.Controllers;
using SimpleApp.Models;
using Xunit;

namespace SimpleApp.Tests {
    public class HomeControllerTests {

        class FakeDataSource : IDataSource {
            public FakeDataSource(Product[] data) => Products = data;
            public IEnumerable<Product> Products { get; set; }
        }

        [Fact]
        public void IndexActionModelIsComplete() {
            // Arrange
            Product[] testData = new Product[] {
                new Product { Name = "P1", Price = 75.10M },
            };
        }
    }
}

```

```

        new Product { Name = "P2", Price = 120M },
        new Product { Name = "P3", Price = 110M }
    };
    IDataSource data = new FakeDataSource(testData);
    var controller = new HomeController();
    controller.dataSource = data;

    // Act
    var model = (controller.Index() as ViewResult)?.ViewData.Model
        as IEnumerable<Product>;

    // Assert
    Assert.Equal(data.Products, model,
        Comparer.Get<Product>((p1, p2) => p1.Name == p2.Name
            && p1.Price == p2.Price));
    }
}
}

```

I have defined a fake implementation of the `IDataSource` interface that lets me use any test data with the controller.

UNDERSTANDING TEST-DRIVEN DEVELOPMENT

I have followed the most commonly used unit testing style in this chapter, in which an application feature is written and then tested to make sure it works as required. This is popular because most developers think about application code first and testing comes second (this is certainly the category that I fall into).

This approach is that it tends to produce unit tests that focus only on the parts of the application code that were difficult to write or that needed some serious debugging, leaving some aspects of a feature only partially tested or untested altogether.

An alternative approach is *Test-Driven Development* (TDD). There are lots of variations on TDD, but the core idea is that you write the tests for a feature before implementing the feature itself. Writing the tests first makes you think more carefully about the specification you are implementing and how you will know that a feature has been implemented correctly. Rather than diving into the implementation detail, TDD makes you consider what the measures of success or failure will be in advance.

The tests that you write will all fail initially because your new feature will not be implemented. But as you add code to the application, your tests will gradually move from red to green, and all your tests will pass by the time that the feature is complete. TDD requires discipline, but it does produce a more comprehensive set of tests and can lead to more robust and reliable code.

Using a Mocking Package

It was easy to create a fake implementation for the `IDataSource` interface, but most classes for which fake implementations are required are more complex and cannot be handled as easily.

A better approach is to use a mocking package, which makes it easy to create fake—or mock—objects for tests. There are many mocking packages available, but the one I use (and have for years) is called Moq. To add Moq to the unit test project, run the command shown in Listing 6-18 in the `Testing` folder.

■ **Note** The Moq package is added to the unit testing project and not the project that contains the application to be tested.

Listing 6-18. Installing the Mocking Package

```
dotnet add SimpleApp.Tests package Moq --version 4.13.1
```


Creating a Mock Object

I can use the Moq framework to create a fake `IDataSource` object without having to define a custom test class, as shown in Listing 6-19.

Listing 6-19. Creating a Mock Object in the `HomeControllerTests.cs` File in the `SimpleApp.Tests` Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using SimpleApp.Controllers;
using SimpleApp.Models;
using Xunit;
using Moq;

namespace SimpleApp.Tests {
    public class HomeControllerTests {

        //class FakeDataSource : IDataSource {
        //    public FakeDataSource(params Product[] data) => Products = data;
        //    public IEnumerable<Product> Products { get; set; }
        //}

        [Fact]
        public void IndexActionModelIsComplete() {

            // Arrange
            Product[] testData = new Product[] {
                new Product { Name = "P1", Price = 75.10M },
                new Product { Name = "P2", Price = 120M },
                new Product { Name = "P3", Price = 110M }
            };
            var mock = new Mock<IDataSource>();
            mock.SetupGet(m => m.Products).Returns(testData);
            var controller = new HomeController();
            controller.dataSource = mock.Object;

            // Act
            var model = (controller.Index() as ViewResult)?.ViewData.Model
                as IEnumerable<Product>;

            // Assert
            Assert.Equal(testData, model,
                Comparer.Get<Product>((p1, p2) => p1.Name == p2.Name
                    && p1.Price == p2.Price));
            mock.VerifyGet(m => m.Products, Times.Once);
        }
    }
}
```

The use of Moq has allowed me to remove the fake implementation of the `IDataSource` interface and replace it with a few lines of code. I am not going to go into detail about the different features that Moq supports, but I will explain the way that I used Moq in the examples. (See <https://github.com/Moq/moq4> for examples and documentation for Moq. There are also examples in later chapters as I explain how to unit test different types of components.)

The first step is to create a new instance of the Mock object, specifying the interface that should be implemented, like this:

```
...
var mock = new Mock<IDataSource>();
...
```

The Mock object I created will fake the `IDataSource` interface. To create an implementation of the `Product` property, I use the `SetupGet` method, like this:

```
...
mock.SetupGet(m => m.Products).Returns(testData);
...
```

The `SetupGet` method is used to implement the getter for a property. The argument to this method is a lambda expression that specifies the property to be implemented, which is `Products` in this example. The `Returns` method is called on the result of the `SetupGet` method to specify the result that will be returned when the property value is read.

The Mock class defines an `Object` property, which returns the object that implements the specified interface with the behaviors that have been defined. I used the `Object` property to set the `dataSource` field defined by the `HomeController`, like this:

```
...
controller.dataSource = mock.Object;
...
```

The final Moq feature I used was to check that the `Products` property was called once, like this:

```
...
mock.VerifyGet(m => m.Products, Times.Once);
...
```

The `VerifyGet` method is one of the methods defined by the Mock class to inspect the state of the mock object when the test has completed. In this case, the `VerifyGet` method allows me to check the number of times that the `Products` property method has been read. The `Times.Once` value specifies that the `VerifyGet` method should throw an exception if the property has not been read exactly once, which will cause the test to fail. (The `Assert` methods usually used in tests work by throwing an exception when a test fails, which is why the `VerifyGet` method can be used to replace an `Assert` method when working with mock objects.)

The overall effect is the same as my fake interface implementation, but mocking is more flexible and more concise and can provide more insight into the behavior of the components under test.

Summary

This chapter focused on unit testing, which can be a powerful tool for improving the quality of code. Unit testing doesn't suit every developer, but it is worth experimenting with and can be useful even if used only for complex features or problem diagnosis. I described the use of the `xUnit.net` test framework, explained the importance of isolating components for testing, and demonstrated some tools and techniques for simplifying unit test code. In the next chapter, I start the development of a more realistic project, named `SportsStore`.

CHAPTER 7



SportsStore: A Real Application

In the previous chapters, I built quick and simple ASP.NET Core applications. I described ASP.NET Core patterns, the essential C# features, and the tools that good ASP.NET Core developers require. Now it is time to put everything together and build a simple but realistic e-commerce application.

My application, called SportsStore, will follow the classic approach taken by online stores everywhere. I will create an online product catalog that customers can browse by category and page, a shopping cart where users can add and remove products, and a checkout where customers can enter their shipping details. I will also create an administration area that includes create, read, update, and delete (CRUD) facilities for managing the catalog, and I will protect it so that only logged-in administrators can make changes.

My goal in this chapter and those that follow is to give you a sense of what real ASP.NET Core development is by creating as realistic an example as possible. I want to focus on ASP.NET Core, of course, so I have simplified the integration with external systems, such as the database, and omitted others entirely, such as payment processing.

You might find the going a little slow as I build up the levels of infrastructure I need, but the initial investment will result in maintainable, extensible, well-structured code with excellent support for unit testing.

UNIT TESTING

I include sections on unit testing different components in the SportsStore application throughout the development process, demonstrating how to isolate and test different ASP.NET Core components.

I know that unit testing is not embraced by everyone. If you do not want to unit test, that is fine with me. To that end, when I have something to say that is purely about testing, I put it in a sidebar like this one. If you are not interested in unit testing, you can skip right over these sections, and the SportsStore application will work just fine. You do not need to do any kind of unit testing to get the technology benefits of ASP.NET Core, although, of course, support for testing is a key reason for adopting ASP.NET Core in many projects.

Most of the features I use for the SportsStore application have their own chapters later in the book. Rather than duplicate everything here, I tell you just enough to make sense of the example application and point you to another chapter for in-depth information.

I will call out each step needed to build the application so that you can see how the ASP.NET Core features fit together. You should pay particular attention when I create views. You will get some odd results if you do not follow the examples closely.

Creating the Projects

I am going to start with a minimal ASP.NET Core project and add the features I require as they are needed. Open a new PowerShell command prompt from the Windows Start menu and run the commands shown in Listing 7-1 to get started.

Tip You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/apress/pro-asp.net-core-3>. See Chapter 1 for how to get help if you have problems running the examples.

Listing 7-1. Creating the SportsStore Project

```
dotnet new globaljson --sdk-version 3.1.101 --output SportsSln/SportsStore
dotnet new web --no-https --output SportsSln/SportsStore --framework netcoreapp3.1
dotnet new sln -o SportsSln

dotnet sln SportsSln add SportsSln/SportsStore
```

These commands create a SportsSln solution folder that contains a SportsStore project folder created with the web project template. The SportsSln folder also contains a solution file, to which the SportsStore project is added.

I am using different names for the solution and project folders to make the examples easier to follow, but if you create a project with Visual Studio, the default is to use the same name for both folders. There is no “right” approach, and you can use whatever names suit your project.

Creating the Unit Test Project

To create the unit test project, run the commands shown in Listing 7-2 in the same location you used for the commands shown in Listing 7-1.

Listing 7-2. Creating the Unit Test Project

```
dotnet new xunit -o SportsSln/SportsStore.Tests --framework netcoreapp3.1
dotnet sln SportsSln add SportsSln/SportsStore.Tests
dotnet add SportsSln/SportsStore.Tests reference SportsSln/SportsStore
```

I am going to use the Moq package to create mock objects. Run the command shown in Listing 7-3 to install the Moq package into the unit testing project. Run this command from the same location as the commands in Listings 7-1 and 7-2.

Listing 7-3. Installing the Moq Package

```
dotnet add SportsSln/SportsStore.Tests package Moq --version 4.13.1
```

Creating the Application Project Folders

The next step is to create folders that will contain the application’s components. Right-click the SportsStore item in the Visual Studio Solution Explorer or Visual Studio Code Explorer pane and select Add ► New Folder or New Folder to create the set of folders described in Table 7-1.

Table 7-1. The Folders Created in Listing 7-3

Name	Description
Models	This folder will contain the data model and the classes that provide access to the data in the application’s database.
Controllers	This folder will contain the controller classes that handle HTTP requests.
Views	This folder will contain all the Razor files, grouped into separate subfolders.
Views/Home	This folder will contain Razor files that are specific to the Home controller, which I create in the “Creating the Controller and View” section.
Views/Shared	This folder will contain Razor files that are common to all controllers.

Opening the Projects

If you are using Visual Studio Code, select File ► Open Folder, navigate to the SportsSln folder, and click the Select Folder button. Visual Studio Code will open the folder and discover the solution and project files. When prompted, as shown in Figure 7-1, click Yes to install the assets required to build the projects. Select SportsStore if Visual Studio Code prompts you to select the project to run.

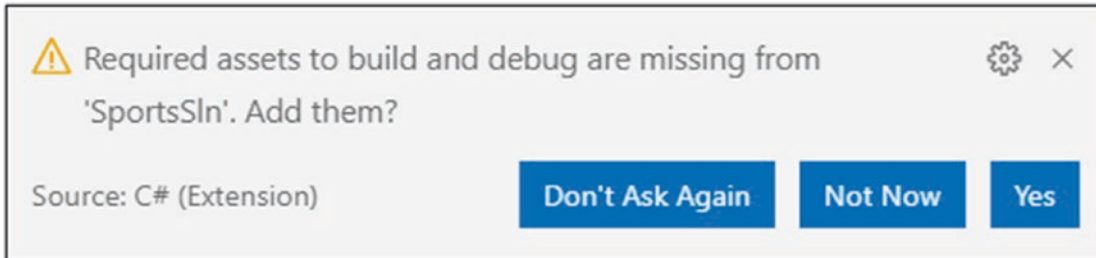


Figure 7-1. Adding assets in Visual Studio Code

If you are using Visual Studio, click the “Open a project or solution” button on the splash screen or select File ► Open ► Project/Solution. Select the SportsSln.sln file in the SportsSln folder and click the Open button to open the project. Once the projects have been opened, select Project ► SportsStore Properties, select the Debug section, and change the port for the URL in the App URL field to 5000, as shown in Figure 7-2. Select File ► Save All to save the new URL.

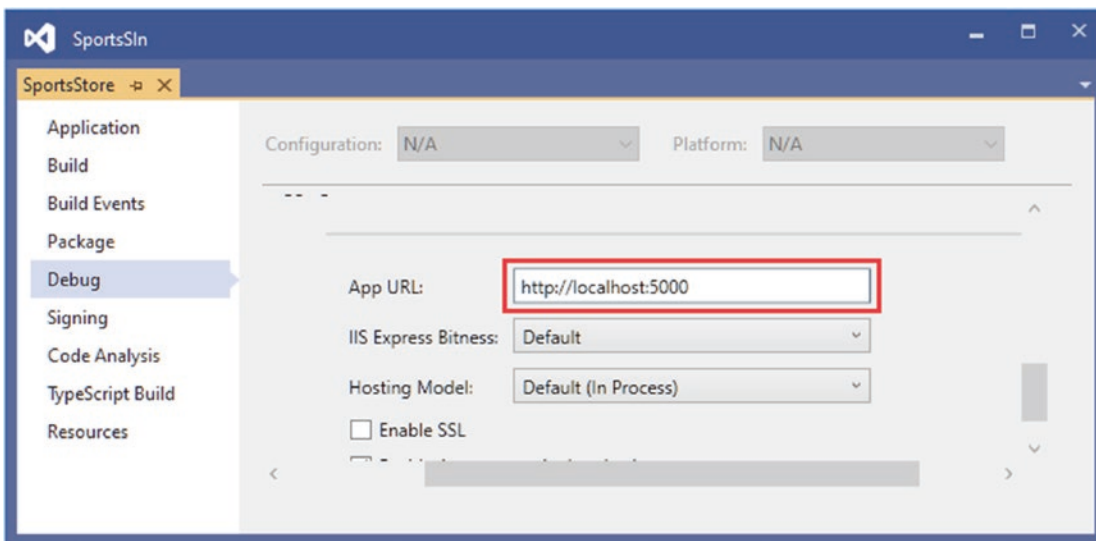


Figure 7-2. Changing the HTTP port in Visual Studio

Preparing the Application Services and the Request Pipeline

The Startup class is responsible for configuring the ASP.NET Core application. Apply the changes shown in Listing 7-4 to the Startup class in the SportsStore project to configure the basic application features.

■ **Note** The Startup class is an important ASP.NET Core feature. I describe it in detail in Chapter 12.

Listing 7-4 Configuring the Application in the Startup.cs File in the SportsStore Folder

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

namespace SportsStore {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.AddControllersWithViews();
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {

            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();

            app.UseRouting();
            app.UseEndpoints(endpoints => {
                endpoints.MapDefaultControllerRoute();
            });
        }
    }
}

```

The `ConfigureServices` method is used to set up objects, known as *services*, that can be used throughout the application and that are accessed through a feature called *dependency injection*, which I describe in Chapter 14. The `AddControllersWithViews` method called in the `ConfigureServices` method sets up the shared objects required by applications using the MVC Framework and the Razor view engine.

ASP.NET Core receives HTTP requests and passes them along a *request pipeline*, which is populated with middleware components registered in the `Configure` method. Each middleware component is able to inspect requests, modify them, generate a response, or modify the responses that other components have produced. The request pipeline is the heart of ASP.NET Core, and I describe it in detail in Chapter 12, where I also explain how to create custom middleware components. Table 7-2 describes the methods that are used to set up middleware components in Listing 7-4.

Table 7-2. The Middleware Methods Used in Listing 7-4

Name	Description
<code>UseDeveloperExceptionPage()</code>	This extension method displays details of exceptions that occur in the application, which is useful during the development process, as described in Chapter 16. It should not be enabled in deployed applications, and I disable this feature when I prepare the SportsStore application for deployment in Chapter 11.
<code>UseStatusCodePages()</code>	This extension method adds a simple message to HTTP responses that would not otherwise have a body, such as 404 - Not Found responses. This feature is described in Chapter 16.
<code>UseStaticFiles()</code>	This extension method enables support for serving static content from the <code>wwwroot</code> folder. I describe the support for static content in Chapter 15.

One especially important middleware component provides the endpoint routing feature, which matches HTTP requests to the application features - known as endpoints - able to produce responses for them, a process I describe in detail in Chapter 13. The endpoint routing feature is added to the request pipeline with the `UseRouting` and `UseEndpoints` methods. To register the MVC Framework as a source of endpoints, Listing 7-4 calls the `MapDefaultControllerRoute` method.

Configuring the Razor View Engine

The Razor view engine is responsible for processing view files, which have the `.cshtml` extension, to generate HTML responses. Some initial preparation is required to configure Razor to make it easier to create views for the application.

Add a Razor View Imports file named `_ViewImports.cshtml` in the `Views` folder with the content shown in Listing 7-5.

■ **Caution** Pay close attention to the contents of this file. It is easy to make a mistake that causes the application to generate incorrect HTML content.

Listing 7-5. The Contents of the `_ViewImports.cshtml` File in the `SportsStore/Views` Folder

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

The `@using` statement will allow me to use the types in the `SportsStore.Models` namespace in views without needing to refer to the namespace. The `@addTagHelper` statement enables the built-in tag helpers, which I use later to create HTML elements that reflect the configuration of the `SportsStore` application and which I describe in detail in Chapter 15.

Add a Razor View Start file named `_ViewStart.cshtml` to the `SportsStore/Views` folder with the content shown in Listing 7-6. (The file will already contain this expression if you create the file using the Visual Studio item template.)

Listing 7-6. The Contents of the `_ViewStart.cshtml` File in the `SportsStore/Views` Folder

```
@{
    Layout = "_Layout";
}
```

The view start file tells Razor to use a layout file in the HTML that it generates, reducing the amount of duplication in views. To create the view, add a Razor layout named `_Layout.cshtml` to the `Views/Shared` folder, with the content shown in Listing 7-7.

Listing 7-7. The Contents of the `_Layout.cshtml` File in the `SportsStore/Views/Shared` Folder

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>SportsStore</title>
</head>
<body>
    <div>
        @RenderBody()
    </div>
</body>
</html>
```

This file defines a simple HTML document into which the contents of other views will be inserted by the `@RenderBody` expression. I explain how Razor expressions work in detail in Chapter 21.

Creating the Controller and View

Add a class file named `HomeController.cs` in the `SportsStore/Controllers` folder and use it to define the class shown in Listing 7-8. This is a minimal controller that contains just enough functionality to produce a response.

Listing 7-8. The Contents of the `HomeController.cs` File in the `SportsStore/Controllers` Folder

```
using Microsoft.AspNetCore.Mvc;

namespace SportsStore.Controllers {
    public class HomeController: Controller {

        public IActionResult Index() => View();

    }
}
```

The `MapDefaultControllerRoute` method used in Listing 7-4 tells ASP.NET Core how to match URLs to controller classes. The configuration applied by that method declares that the `Index` action method defined by the `Home` controller will be used to handle requests.

The `Index` action method doesn't do anything useful yet and just returns the result of calling the `View` method, which is inherited from the `Controller` base class. This result tells ASP.NET Core to render the default view associated with the action method. To create the view, add a Razor View file named `Index.cshtml` to the `Views/Home` folder with the content shown in Listing 7-9.

Listing 7-9. The Contents of the `Index.cshtml` File in the `SportsStore/Views/Home` Folder

```
<h4>Welcome to SportsStore</h4>
```

Starting the Data Model

Almost all projects have a data model of some sort. Since this is an e-commerce application, the most obvious model I need is for a product. Add a class file named `Product.cs` to the `Models` folder and use it to define the class shown in Listing 7-10.

Listing 7-10. The Contents of the `Product.cs` File in the `SportsStore/Models` Folder

```
using System.ComponentModel.DataAnnotations.Schema;

namespace SportsStore.Models {
    public class Product {

        public long ProductID { get; set; }

        public string Name { get; set; }

        public string Description { get; set; }

        [Column(TypeName = "decimal(8, 2)")]
        public decimal Price { get; set; }

        public string Category { get; set; }
    }
}
```

The `Price` property has been decorated with the `Column` attribute to specify the SQL data type that will be used to store values for this property. Not all C# types map neatly onto SQL types, and this attribute ensures the database uses an appropriate type for the application data.

Checking and Running the Application

Before going any further, it is a good idea to make sure the application builds and runs as expected. Select Start Without Debugging or Run Without Debugging from the Debug menu or run the command shown in Listing 7-11 in the SportsStore folder.

Listing 7-11 Running the Example Application

```
dotnet run
```

Request `http://localhost:5000`, and you will see the response shown in Figure 7-3.

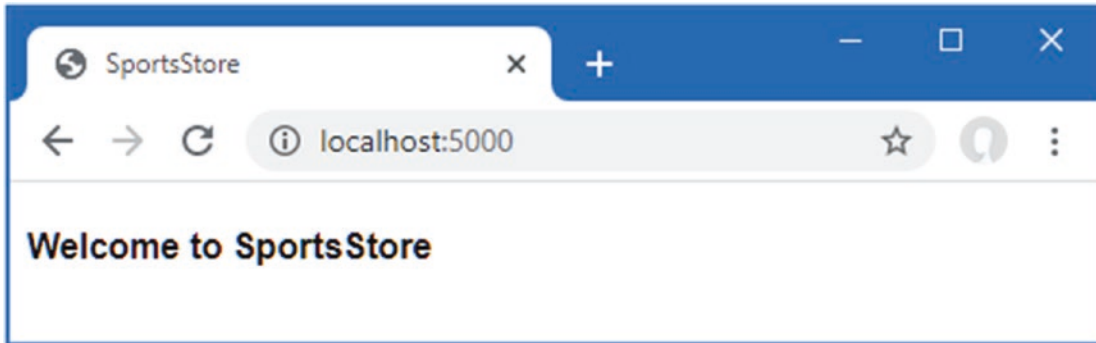


Figure 7-3. Running the example application

Adding Data to the Application

Now that the SportsStore contains some basic setup and can produce a simple response, it is time to add some data so that the application has something more useful to display. The SportsStore application will store its data in a SQL Server LocalDB database, which is accessed using Entity Framework Core. Entity Framework Core is the Microsoft object-to-relational mapping (ORM) framework, and it is the most widely used method of accessing databases in ASP.NET Core projects.

Caution If you did not install LocalDB when you prepared your development environment in Chapter 2, you must do so now. The SportsStore application will not work without its database.

Installing the Entity Framework Core Packages

The first step is to add Entity Framework Core to the project. Use a PowerShell command prompt to run the command shown in Listing 7-12 in the SportsStore folder.

Listing 7-12. Adding the Entity Framework Core Packages to the SportsStore Project

```
dotnet add package Microsoft.EntityFrameworkCore.Design --version 3.1.1
dotnet add package Microsoft.EntityFrameworkCore.SqlServer --version 3.1.1
```

These packages install Entity Framework Core and the support for using SQL Server. Entity Framework Core also requires a tools package, which includes the command-line tools required to prepare and create databases for ASP.NET Core applications. Run the commands shown in Listing 7-13 to remove any existing version of the tools package, if there is one, and install the version used in this book. (Since this package is installed globally, you can run these commands in any folder.)

Listing 7-13. Installing the Entity Framework Core Tool Package

```
dotnet tool uninstall --global dotnet-ef
dotnet tool install --global dotnet-ef --version 3.1.1
```

Defining the Connection String

Configuration settings, such as database connection strings, are stored in JSON configuration files. To describe the connection to the database that will be used for the SportsStore data, add the entries shown in Listing 7-14 to the `appsettings.json` file in the SportsStore folder.

The project also contains an `appsettings.Development.json` file that contains configuration settings that are used only in development. This file is displayed as nested within the `appsettings.json` file by Solution Explorer but is always visible in Visual Studio Code. I use only the `appsettings.json` file for the development of the SportsStore project, but I explain the relationship between the files and how they are both used in detail in Chapter 15.

■ **Tip** Connection strings must be expressed as a single unbroken line, which is fine in the code editor but doesn't fit on the printed page and is the cause of the awkward formatting in Listing 7-14. When you define the connection string in your own project, make sure that the value of the `SportsStoreConnection` item is on a single line.

Listing 7-14. Adding a Configuration Setting in the `appsettings.json` File in the SportsStore Folder

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "SportsStoreConnection": "Server=(localdb)\\MSSQLLocalDB;Database=SportsStore;MultipleActiveResultSets=true"
  }
}
```

This configuration string specifies a LocalDB database called SportsStore and enables the multiple active result set feature (MARS), which is required for some of the database queries that will be made by the SportsStore application using Entity Framework Core.

Pay close attention when you add the configuration setting. JSON data must be expressed exactly as shown in the listing, which means you must ensure you correctly quote the property names and values. You can download the configuration file from the GitHub repository if you have difficulty.

■ **Tip** Each database server requires its own connection string format. A helpful site for formulating connection strings is www.connectionstrings.com.

Creating the Database Context Class

Entity Framework Core provides access to the database through a context class. Add a class file named `StoreDbContext.cs` to the `Models` folder and use it to define the class shown in Listing 7-15.

Listing 7-15. The Contents of the StoreDbContext.cs File in the SportsStore/Models Folder

```
using Microsoft.EntityFrameworkCore;

namespace SportsStore.Models {
    public class StoreDbContext: DbContext {

        public StoreDbContext(DbContextOptions<StoreDbContext> options)
            : base(options) { }

        public DbSet<Product> Products { get; set; }
    }
}
```

The DbContext base class provides access to the Entity Framework Core's underlying functionality, and the Products property will provide access to the Product objects in the database. The StoreDbContext class is derived from DbContext and adds the properties that will be used to read and write the application's data. There is only one property for now, which will provide access to Product objects.

Configuring Entity Framework Core

Entity Framework Core must be configured so that it knows the type of database to which it will connect, which connection string describes that connection, and which context class will present the data in the database. Listing 7-16 shows the required changes to the Startup class.

Listing 7-16. Configuring Entity Framework Core in the Startup.cs File in the SportsStore Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using SportsStore.Models;

namespace SportsStore {
    public class Startup {

        public Startup(IConfiguration config) {
            Configuration = config;
        }

        private IConfiguration Configuration { get; set; }

        public void ConfigureServices(IServiceCollection services) {
            services.AddControllersWithViews();
            services.AddDbContext<StoreDbContext>(opts => {
                opts.UseSqlServer(
                    Configuration["ConnectionStrings:SportsStoreConnection"]);
            });
        }

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

```

        app.UseDeveloperExceptionPage();
        app.UseStatusCodePages();
        app.UseStaticFiles();

        app.UseRouting();
        app.UseEndpoints(endpoints => {
            endpoints.MapDefaultControllerRoute();
        });
    }
}

```

The `IConfiguration` interface provides access to the ASP.NET Core configuration system, which includes the contents of the `appsettings.json` file and which I describe in detail in Chapter 15. The constructor receives an `IConfiguration` object through its constructor and assigns it to the `Configuration` property, which is used to access the connection string.

Entity Framework Core is configured with the `AddDbContext` method, which registers the database context class and configures the relationship with the database. The `UseSqlServer` method declares that SQL Server is being used and the connection string is read via the `IConfiguration` object.

Creating a Repository

The next step is to create a repository interface and implementation class. The repository pattern is one of the most widely used, and it provides a consistent way to access the features presented by the database context class. Not everyone finds a repository useful, but my experience is that it can reduce duplication and ensures that operations on the database are performed consistently. Add a class file named `IStoreRepository.cs` to the `Models` folder and use it to define the interface shown in Listing 7-17.

Listing 7-17. The Contents of the `IStoreRepository.cs` File in the `SportsStore/Models` Folder

```

using System.Linq;

namespace SportsStore.Models {
    public interface IStoreRepository {

        IQueryable<Product> Products { get; }
    }
}

```

This interface uses `IQueryable<T>` to allow a caller to obtain a sequence of `Product` objects. The `IQueryable<T>` interface is derived from the more familiar `IEnumerable<T>` interface and represents a collection of objects that can be queried, such as those managed by a database.

A class that depends on the `IProductRepository` interface can obtain `Product` objects without needing to know the details of how they are stored or how the implementation class will deliver them.

UNDERSTANDING IENUMERABLE<T> AND IQUERYABLE<T> INTERFACES

The `IQueryable<T>` interface is useful because it allows a collection of objects to be queried efficiently. Later in this chapter, I add support for retrieving a subset of `Product` objects from a database, and using the `IQueryable<T>` interface allows me to ask the database for just the objects that I require using standard LINQ statements and without needing to know what database server stores the data or how it processes the query. Without the `IQueryable<T>` interface, I would have to retrieve all of the `Product` objects from the database and then discard the ones that I don't want, which becomes an expensive operation as the amount of data used by an application increases. It is for this reason that the `IQueryable<T>` interface is typically used instead of `IEnumerable<T>` in database repository interfaces and classes.

However, care must be taken with the `IQueryable<T>` interface because each time the collection of objects is enumerated, the query will be evaluated again, which means that a new query will be sent to the database. This can undermine the efficiency gains of using `IQueryable<T>`. In such situations, you can convert the `IQueryable<T>` interface to a more predictable form using the `ToList` or `ToArray` extension method.

To create an implementation of the repository interface, add a class file named `EFStoreRepository.cs` in the `Models` folder and use it to define the class shown in Listing 7-18.

Listing 7-18. The Contents of the `EFStoreRepository.cs` File in the `SportsStore/Models` Folder

```
using System.Linq;

namespace SportsStore.Models {
    public class EFStoreRepository : IStoreRepository {
        private StoreDbContext context;

        public EFStoreRepository(StoreDbContext ctx) {
            context = ctx;
        }

        public IQueryable<Product> Products => context.Products;
    }
}
```

I'll add additional functionality as I add features to the application, but for the moment, the repository implementation just maps the `Products` property defined by the `IStoreRepository` interface onto the `Products` property defined by the `StoreDbContext` class. The `Products` property in the context class returns a `DbSet<Product>` object, which implements the `IQueryable<T>` interface and makes it easy to implement the repository interface when using Entity Framework Core.

Earlier in the chapter, I explained that ASP.NET Core supports services that allow objects to be accessed throughout the application. One benefit of services is they allow classes to use interfaces without needing to know which implementation class is being used. I explain this in detail in Chapter 14, but for the `SportsStore` chapters, it means that application components can access objects that implement the `IStoreRepository` interface without knowing that it is the `EFStoreRepository` implementation class they are using. This makes it easy to change the implementation class the application uses without needing to make changes to the individual components. Add the statement shown in Listing 7-19 to the `Startup` class to create a service for the `IStoreRepository` interface that uses `EFStoreRepository` as the implementation class.

■ **Tip** Don't worry if this doesn't make sense right now. This topic is one of the most confusing aspects of working with ASP.NET Core, and it can take a while to understand.

Listing 7-19. Creating the Repository Service in the `Startup.cs` File in the `SportsStore` Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using SportsStore.Models;

namespace SportsStore {
    public class Startup {
```

```

public Startup(IConfiguration config) {
    Configuration = config;
}

private IConfiguration Configuration { get; set; }

public void ConfigureServices(IServiceCollection services) {
    services.AddControllersWithViews();

    services.AddDbContext<StoreDbContext>(opts => {
        opts.UseSqlServer(
            Configuration["ConnectionStrings:SportsStoreConnection"]);
    });
    services.AddScoped<IStoreRepository, EFStoreRepository>();
}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {

    app.UseDeveloperExceptionPage();
    app.UseStatusCodePages();
    app.UseStaticFiles();

    app.UseRouting();
    app.UseEndpoints(endpoints => {
        endpoints.MapDefaultControllerRoute();
    });
}
}
}

```

The `AddScoped` method creates a service where each HTTP request gets its own repository object, which is the way that Entity Framework Core is typically used.

Creating the Database Migration

Entity Framework Core is able to generate the schema for the database using the data model classes through a feature called *migrations*. When you prepare a migration, Entity Framework Core creates a C# class that contains the SQL commands required to prepare the database. If you need to modify your model classes, then you can create a new migration that contains the SQL commands required to reflect the changes. In this way, you don't have to worry about manually writing and testing SQL commands and can just focus on the C# model classes in the application.

Entity Framework Core commands are performed from the command line. Open a PowerShell command prompt and run the command shown in Listing 7-20 in the `SportsStore` folder to create the migration class that will prepare the database for its first use.

Listing 7-20. Creating the Database Migration

```
dotnet ef migrations add Initial
```

When this command has finished, the `SportsStore` project will contain a `Migrations` folder. This is where Entity Framework Core stores its migration classes. One of the file names will be a timestamp followed by `_Initial.cs`, and this is the class that will be used to create the initial schema for the database. If you examine the contents of this file, you can see how the `Product` model class has been used to create the schema.

WHAT ABOUT THE ADD-MIGRATION AND UPDATE-DATABASE COMMANDS?

If you are an experienced Entity Framework developer, you may be used to using the `Add-Migration` command to create a database migration and to using the `Update-Database` command to apply it to a database.

With the introduction of .NET Core, Entity Framework Core has added commands that are integrated into the `dotnet` command-line tool, using the commands added by the `Microsoft.EntityFrameworkCore.Tools.DotNet` package. These are the commands that I have used because they are consistent with other .NET commands and they can be used in any command prompt or PowerShell window, unlike the `Add-Migration` and `Update-Database` commands, which work only in a specific Visual Studio window.

Creating Seed Data

To populate the database and provide some sample data, I added a class file called `SeedData.cs` to the `Models` folder and defined the class shown in Listing 7-21.

Listing 7-21. The Contents of the `SeedData.cs` File in the `SportsStore/Models` Folder

```
using System.Linq;
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.EntityFrameworkCore;

namespace SportsStore.Models {

    public static class SeedData {

        public static void EnsurePopulated(IApplicationBuilder app) {
            StoreDbContext context = app.ApplicationServices
                .CreateScope().ServiceProvider.GetRequiredService<StoreDbContext>();

            if (context.Database.GetPendingMigrations().Any()) {
                context.Database.Migrate();
            }

            if (!context.Products.Any()) {
                context.Products.AddRange(
                    new Product {
                        Name = "Kayak", Description = "A boat for one person",
                        Category = "Watersports", Price = 275
                    },
                    new Product {
                        Name = "Lifejacket",
                        Description = "Protective and fashionable",
                        Category = "Watersports", Price = 48.95m
                    },
                    new Product {
                        Name = "Soccer Ball",
                        Description = "FIFA-approved size and weight",
                        Category = "Soccer", Price = 19.50m
                    },
                    new Product {
                        Name = "Corner Flags",
                        Description = "Give your playing field a professional touch",
                        Category = "Soccer", Price = 34.95m
                    }
                );
            }
        }
    }
}
```

```

        new Product {
            Name = "Stadium",
            Description = "Flat-packed 35,000-seat stadium",
            Category = "Soccer", Price = 79500
        },
        new Product {
            Name = "Thinking Cap",
            Description = "Improve brain efficiency by 75%",
            Category = "Chess", Price = 16
        },
        new Product {
            Name = "Unsteady Chair",
            Description = "Secretly give your opponent a disadvantage",
            Category = "Chess", Price = 29.95m
        },
        new Product {
            Name = "Human Chess Board",
            Description = "A fun game for the family",
            Category = "Chess", Price = 75
        },
        new Product {
            Name = "Bling-Bling King",
            Description = "Gold-plated, diamond-studded King",
            Category = "Chess", Price = 1200
        }
    );
    context.SaveChanges();
}
}
}
}
}

```

The static `EnsurePopulated` method receives an `IApplicationBuilder` argument, which is the interface used in the `Configure` method of the `Startup` class to register middleware components to handle HTTP requests. `IApplicationBuilder` also provides access to the application's services, including the Entity Framework Core database context service.

The `EnsurePopulated` method obtains a `StoreDbContext` object through the `IApplicationBuilder` interface and calls the `Database.Migrate` method if there are any pending migrations, which means that the database will be created and prepared so that it can store `Product` objects. Next, the number of `Product` objects in the database is checked. If there are no objects in the database, then the database is populated using a collection of `Product` objects using the `AddRange` method and then written to the database using the `SaveChanges` method.

The final change is to seed the database when the application starts, which I have done by adding a call to the `EnsurePopulated` method from the `Startup` class, as shown in Listing 7-22.

Listing 7-22. Seeding the Database in the `Startup.cs` File in the `SportsStore` Folder

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using SportsStore.Models;

```



```

namespace SportsStore {
    public class Startup {

        public Startup(IConfiguration config) {
            Configuration = config;
        }

        private IConfiguration Configuration { get; set; }

        public void ConfigureServices(IServiceCollection services) {
            services.AddControllersWithViews();

            services.AddDbContext<StoreDbContext>(opts => {
                opts.UseSqlServer(
                    Configuration["ConnectionStrings:SportsStoreConnection"]);
            });
            services.AddScoped<IStoreRepository, EFStoreRepository>();
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {

            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();

            app.UseRouting();
            app.UseEndpoints(endpoints => {
                endpoints.MapDefaultControllerRoute();
            });
            SeedData.EnsurePopulated(app);
        }
    }
}

```

RESETTING THE DATABASE

If you need to reset the database, then run this command in the `SportsStore` folder:

```

...
dotnet ef database drop --force --context StoreDbContext
...

```

Start ASP.NET Core, and the database will be re-created and seeded with data.

Displaying a List of Products

As you have seen, the initial preparation work for an ASP.NET Core project can take some time. But the good news is that once the foundation is in place, the pace improves, and features are added more rapidly. In this section, I am going to create a controller and an action method that can display details of the products in the repository.

USING THE VISUAL STUDIO SCAFFOLDING

As I noted in Chapter 4, Visual Studio supports scaffolding to add items to a project.

I don't use the scaffolding in this book. The code and markup that the scaffolding generates are so generic as to be all but useless, and the scenarios that are supported are narrow and don't address common development problems. My goal in this book is not only to make sure you know how to create ASP.NET Core applications but also to explain how everything works behind the scenes, and that is harder to do when responsibility for creating components is handed to the scaffolding.

If you are using Visual Scer in the Solution Explorer, selecting Add ► New Item from the pop-up menu, and then choosing an item template from the Add New Item window.

You may find your development style to be different from mine, and you may find that you prefer working with the scaffolding in your own projects. That's perfectly reasonable, although I recommend you take the time to understand what the scaffolding does so you know where to look if you don't get the results you expect.

Preparing the Controller

Add the statements shown in Listing 7-23 to prepare the controller to display the list of products.

Listing 7-23. Preparing the Controller in the HomeController.cs File in the SportsStore/Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;

namespace SportsStore.Controllers {
    public class HomeController : Controller {
        private IStoreRepository repository;

        public HomeController(IStoreRepository repo) {
            repository = repo;
        }

        public IActionResult Index() => View(repository.Products);
    }
}
```

When ASP.NET Core needs to create a new instance of the HomeController class to handle an HTTP request, it will inspect the constructor and see that it requires an object that implements the IStoreRepository interface. To determine what implementation class should be used, ASP.NET Core consults the configuration in the Startup class, which tells it that EFStoreRepository should be used and that a new instance should be created for every request. ASP.NET Core creates a new EFStoreRepository object and uses it to invoke the HomeController constructor to create the controller object that will process the HTTP request.

This is known as *dependency injection*, and its approach allows the HomeController object to access the application's repository through the IStoreRepository interface without knowing which implementation class has been configured. I could reconfigure the service to use a different implementation class—one that doesn't use Entity Framework Core, for example—and dependency injection means that the controller will continue to work without changes.

■ **Note** Some developers don't like dependency injection and believe it makes applications more complicated. That's not my view, but if you are new to dependency injection, then I recommend you wait until you have read Chapter 14 before you make up your mind.

UNIT TEST: REPOSITORY ACCESS

I can unit test that the controller is accessing the repository correctly by creating a mock repository, injecting it into the constructor of the HomeController class, and then calling the Index method to get the response that contains the list of products. I then compare the Product objects I get to what I would expect from the test data in the mock implementation. See Chapter 6 for details of how to set up unit tests. Here is the unit test I created for this purpose, in a class file called HomeControllerTests.cs that I added to the SportsStore.Tests project:

```
using System.Collections.Generic;
using System.Linq;
using Microsoft.AspNetCore.Mvc;
```

```

using Moq;
using SportsStore.Controllers;
using SportsStore.Models;
using Xunit;

namespace SportsStore.Tests {
    public class ProductControllerTests {
        [Fact]
        public void Can_Use_Repository() {
            // Arrange
            Mock<IStoreRepository> mock = new Mock<IStoreRepository>();
            mock.Setup(m => m.Products).Returns((new Product[] {
                new Product {ProductID = 1, Name = "P1"},
                new Product {ProductID = 2, Name = "P2"}
            }).AsQueryable<Product>());

            HomeController controller = new HomeController(mock.Object);

            // Act
            IEnumerable<Product> result =
                (controller.Index() as ViewResult).ViewData.Model
                as IEnumerable<Product>;

            // Assert
            Product[] prodArray = result.ToArray();
            Assert.True(prodArray.Length == 2);
            Assert.Equal("P1", prodArray[0].Name);
            Assert.Equal("P2", prodArray[1].Name);
        }
    }
}

```

It is a little awkward to get the data returned from the action method. The result is a `ViewResult` object, and I have to cast the value of its `ViewData.Model` property to the expected data type. I explain the different result types that can be returned by action methods and how to work with them in Part 2.

Updating the View

The `Index` action method in Listing 7-23 passes the collection of `Product` objects from the repository to the `View` method, which means these objects will be the view model that Razor uses when it generates HTML content from the view. Make the changes to the view shown in Listing 7-24 to generate content using the `Product` view model objects.

Listing 7-24. Using the Product Data in the `Index.cshtml` File in the `SportsStore/Views/Home` Folder

```

@model IQueryable<Product>

@foreach (var p in Model) {
    <div>
        <h3>@p.Name</h3>
        @p.Description
        <h4>@p.Price.ToString("c")</h4>
    </div>
}

```

The `@model` expression at the top of the file specifies that the view expects to receive a sequence of `Product` objects from the action method as its model data. I use an `@foreach` expression to work through the sequence and generate a simple set of HTML elements for each `Product` object that is received.

The view doesn't know where the `Product` objects came from, how they were obtained, or whether they represent all the products known to the application. Instead, the view deals only with how details of each `Product` are displayed using HTML elements.

■ **Tip** I converted the `Price` property to a string using the `ToString("c")` method, which renders numerical values as currency according to the culture settings that are in effect on your server. For example, if the server is set up as `en-US`, then `(1002.3)`. `ToString("c")` will return `$1,002.30`, but if the server is set to `en-GB`, then the same method will return `£1,002.30`.

Running the Application

Start ASP.NET Core and request `http://localhost:5000` to see the list of products, which is shown in Figure 7-4. This is the typical pattern of development for ASP.NET Core. An initial investment of time setting everything up is necessary, and then the basic features of the application snap together quickly.

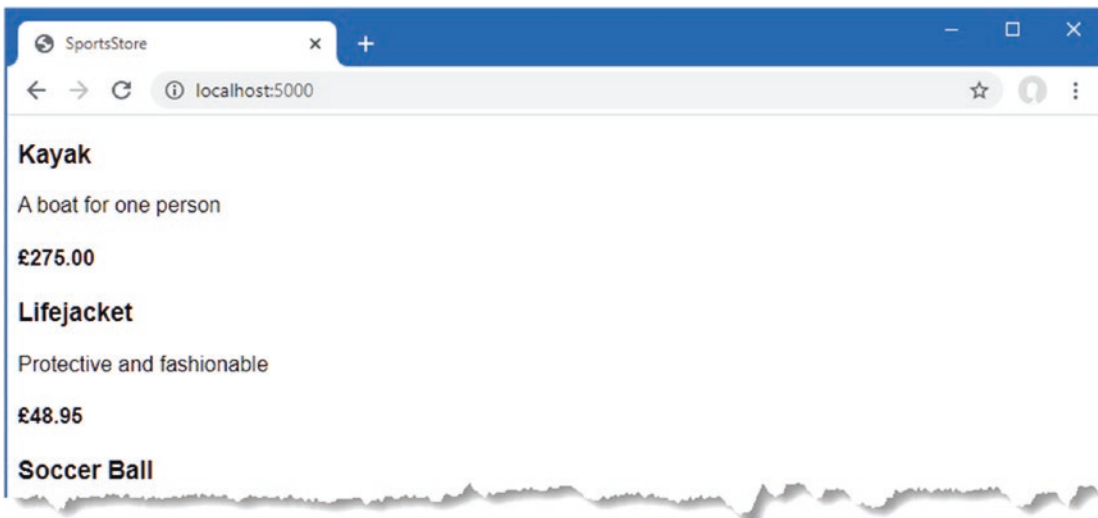


Figure 7-4. Displaying a list of products

Adding Pagination

You can see from Figure 7-4 that the `Index.cshtml` view displays the products in the database on a single page. In this section, I will add support for pagination so that the view displays a smaller number of products on a page, and the user can move from page to page to view the overall catalog. To do this, I am going to add a parameter to the `Index` method in the `Home` controller, as shown in Listing 7-25.

Listing 7-25. Adding Pagination in the `HomeController.cs` File in the `SportsStore/Controllers` Folder

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;
using System.Linq;

namespace SportsStore.Controllers {
    public class HomeController : Controller {
        private IStoreRepository repository;
        public int PageSize = 4;
    }
}
```

```

public HomeController(IStoreRepository repo) {
    repository = repo;
}

public IActionResult Index(int productPage = 1)
    => View(repository.Products
        .OrderBy(p => p.ProductID)
        .Skip((productPage - 1) * PageSize)
        .Take(PageSize));
}
}

```

The `PageSize` field specifies that I want four products per page. I have added an optional parameter to the `Index` method, which means that if the method is called without a parameter, the call is treated as though I had supplied the value specified in the parameter definition, with the effect that the action method displays the first page of products when it is invoked without an argument. Within the body of the action method, I get the `Product` objects, order them by the primary key, skip over the products that occur before the start of the current page, and take the number of products specified by the `PageSize` field.

UNIT TEST: PAGINATION

I can unit test the pagination feature by mocking the repository, requesting a specific page from the controller, and making sure I get the expected subset of the data. Here is the unit test I created for this purpose and added to the `HomeControllerTests.cs` file in the `SportsStore.Tests` project:

```

using System.Collections.Generic;
using System.Linq;
using Microsoft.AspNetCore.Mvc;
using Moq;
using SportsStore.Controllers;
using SportsStore.Models;
using Xunit;

namespace SportsStore.Tests {

    public class ProductControllerTests {

        [Fact]
        public void Can_Use_Repository() {
            // ...statements omitted for brevity...
        }

        [Fact]
        public void Can_Paginate() {
            // Arrange
            Mock<IStoreRepository> mock = new Mock<IStoreRepository>();
            mock.Setup(m => m.Products).Returns((new Product[] {
                new Product {ProductID = 1, Name = "P1"},
                new Product {ProductID = 2, Name = "P2"},
                new Product {ProductID = 3, Name = "P3"},
                new Product {ProductID = 4, Name = "P4"},
                new Product {ProductID = 5, Name = "P5"}
            }).AsQueryable<Product>());

            HomeController controller = new HomeController(mock.Object);
            controller.PageSize = 3;

            // Act
            IEnumerable<Product> result =
                (controller.Index(2) as ViewResult).ViewData.Model
                as IEnumerable<Product>;
        }
    }
}

```

```

        // Assert
        Product[] prodArray = result.ToArray();
        Assert.True(prodArray.Length == 2);
        Assert.Equal("P4", prodArray[0].Name);
        Assert.Equal("P5", prodArray[1].Name);
    }
}
}

```

You can see the new test follows the pattern of the existing one, relying on Moq to provide a known set of data with which to work.

Displaying Page Links

Restart ASP.NET Core and request `http://localhost:5000`, and you will see that there are now four items shown on the page, as shown in Figure 7-5. If you want to view another page, you can append query string parameters to the end of the URL, like this:

```
http://localhost:5000/?productPage=2
```

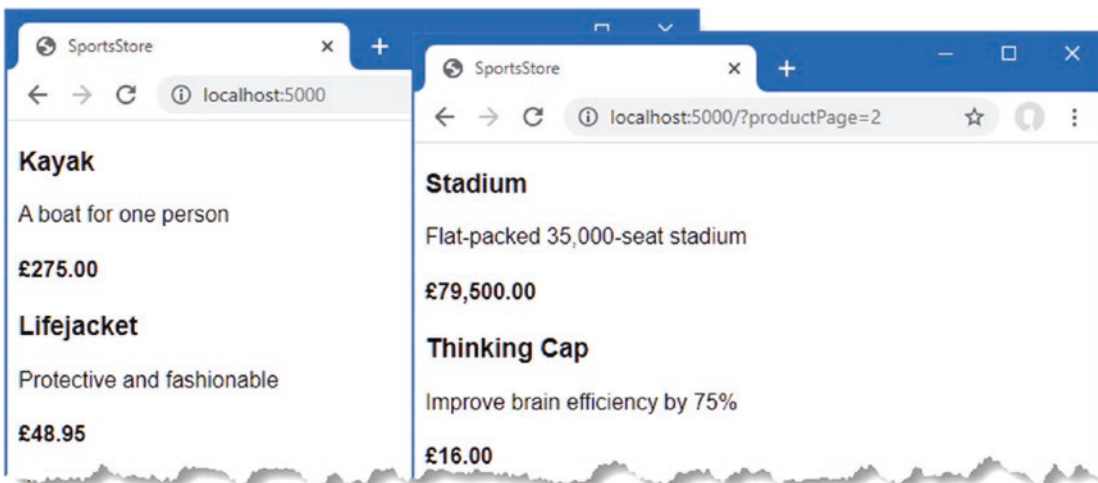


Figure 7-5. Paging through data

Using these query strings, you can navigate through the catalog of products. There is no way for customers to figure out that these query string parameters exist, and even if there were, customers are not going to want to navigate this way. Instead, I need to render some page links at the bottom of each list of products so that customers can navigate between pages. To do this, I am going to create a *tag helper*, which generates the HTML markup for the links I require.

Adding the View Model

To support the tag helper, I am going to pass information to the view about the number of pages available, the current page, and the total number of products in the repository. The easiest way to do this is to create a view model class, which is used specifically to pass data between a controller and a view. Create a `Models/ViewModels` folder in the `SportsStore` project, add to it a class file named `PagingInfo.cs`, and define the class shown in Listing 7-26.

Listing 7-26. The Contents of the PagingInfo.cs File in the SportsStore/Models/ViewModels Folder

```
using System;

namespace SportsStore.Models.ViewModels {

    public class PagingInfo {
        public int TotalItems { get; set; }
        public int ItemsPerPage { get; set; }
        public int CurrentPage { get; set; }

        public int TotalPages =>
            (int)Math.Ceiling((decimal)TotalItems / ItemsPerPage);
    }
}
```

Adding the Tag Helper Class

Now that I have a view model, it is time to create a tag helper class. Create a folder named `Infrastructure` in the `SportsStore` project and add to it a class file called `PageLinkTagHelper.cs`, with the code shown in Listing 7-27. Tag helpers are a big part of ASP.NET Core development, and I explain how they work and how to use and create them in Chapters 25–27.

■ **Tip** The `Infrastructure` folder is where I put classes that deliver the plumbing for an application but that are not related to the application’s main functionality. You don’t have to follow this convention in your own projects.

Listing 7-27. The Contents of the PageLinkTagHelper.cs File in the SportsStore/Infrastructure Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.AspNetCore.Mvc.Routing;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using Microsoft.AspNetCore.Razor.TagHelpers;
using SportsStore.Models.ViewModels;

namespace SportsStore.Infrastructure {

    [HtmlTargetElement("div", Attributes = "page-model")]
    public class PageLinkTagHelper : TagHelper {
        private IUrlHelperFactory urlHelperFactory;

        public PageLinkTagHelper(IUrlHelperFactory helperFactory) {
            urlHelperFactory = helperFactory;
        }

        [ViewContext]
        [HtmlAttributeNotBound]
        public ViewContext ViewContext { get; set; }

        public PagingInfo PageModel { get; set; }

        public string PageAction { get; set; }

        public override void Process(TagHelperContext context,
            TagHelperOutput output) {
            IUrlHelper urlHelper = urlHelperFactory.GetUrlHelper(ViewContext);
```

```

        TagBuilder result = new TagBuilder("div");
        for (int i = 1; i <= PageModel.TotalPages; i++) {
            TagBuilder tag = new TagBuilder("a");
            tag.Attributes["href"] = urlHelper.Action(PageAction,
                new { productPage = i });
            tag.InnerHtml.Append(i.ToString());
            result.InnerHtml.AppendHtml(tag);
        }
        output.Content.AppendHtml(result.InnerHtml);
    }
}
}

```

This tag helper populates a div element with a elements that correspond to pages of products. I am not going to go into detail about tag helpers now; it is enough to know that they are one of the most useful ways that you can introduce C# logic into your views. The code for a tag helper can look tortured because C# and HTML don't mix easily. But using tag helpers is preferable to including blocks of C# code in a view because a tag helper can be easily unit tested.

Most ASP.NET Core components, such as controllers and views, are discovered automatically, but tag helpers have to be registered. In Listing 7-28, I have added a statement to the `_ViewImports.cshtml` file in the Views folder that tells ASP.NET Core to look for tag helper classes in the SportsStore project. I also added an `@using` expression so that I can refer to the view model classes in views without having to qualify their names with the namespace.

Listing 7-28. Registering a Tag Helper in the `_ViewImports.cshtml` File in the SportsStore/Views Folder

```

@using SportsStore.Models
@using SportsStore.Models.ViewModels
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper *, SportsStore

```

UNIT TEST: CREATING PAGE LINKS

To test the `PageLinkTagHelper` tag helper class, I call the `Process` method with test data and provide a `TagHelperOutput` object that I inspect to see the HTML that is generated, as follows, which I defined in a new `PageLinkTagHelperTests.cs` file in the `SportsStore.Tests` project:

```

using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Routing;
using Microsoft.AspNetCore.Razor.TagHelpers;
using Moq;
using SportsStore.Infrastructure;
using SportsStore.Models.ViewModels;
using Xunit;

namespace SportsStore.Tests {
    public class PageLinkTagHelperTests {
        [Fact]
        public void Can_Generate_Page_Links() {
            // Arrange
            var urlHelper = new Mock<IUrlHelper>();
            urlHelper.SetupSequence(x => x.Action(It.IsAny<UrlActionContext>()))
                .Returns("Test/Page1")
                .Returns("Test/Page2")
                .Returns("Test/Page3");
        }
    }
}

```



```

var urlHelperFactory = new Mock<IUrlHelperFactory>();
urlHelperFactory.Setup(f =>
    f.GetUrlHelper(It.IsAny<ActionContext>()))
    .Returns(urlHelper.Object);

PageLinkTagHelper helper =
    new PageLinkTagHelper(urlHelperFactory.Object) {
        PageModel = new PagingInfo {
            CurrentPage = 2,
            TotalItems = 28,
            ItemsPerPage = 10
        },
        PageAction = "Test"
    };

TagHelperContext ctx = new TagHelperContext(
    new TagHelperAttributeList(),
    new Dictionary<object, object>(), "");

var content = new Mock<TagHelperContent>();
TagHelperOutput output = new TagHelperOutput("div",
    new TagHelperAttributeList(),
    (cache, encoder) => Task.FromResult(content.Object));

// Act
helper.Process(ctx, output);

// Assert
Assert.Equal(@"<a href=""Test/Page1"">1</a>"
    + @"<a href=""Test/Page2"">2</a>"
    + @"<a href=""Test/Page3"">3</a>",
    output.Content.GetContent());
}
}
}

```

The complexity in this test is in creating the objects that are required to create and use a tag helper. Tag helpers use `IUrlHelperFactory` objects to generate URLs that target different parts of the application, and I have used Moq to create an implementation of this interface and the related `IUrlHelper` interface that provides test data.

The core part of the test verifies the tag helper output by using a literal string value that contains double quotes. C# is perfectly capable of working with such strings, as long as the string is prefixed with `@` and uses two sets of double quotes (`""`) in place of one set of double quotes. You must remember not to break the literal string into separate lines unless the string you are comparing to is similarly broken. For example, the literal I use in the test method has wrapped onto several lines because the width of a printed page is narrow. I have not added a newline character; if I did, the test would fail.

Adding the View Model Data

I am not quite ready to use the tag helper because I have yet to provide an instance of the `PagingInfo` view model class to the view. To do this, I added a class file called `ProductsListViewModel.cs` to the `Models/ViewModels` folder of the `SportsStore` project with the content shown in Listing 7-29.

Listing 7-29. The Contents of the `ProductsListViewModel.cs` File in the `SportsStore/Models/ViewModels` Folder

```

using System.Collections.Generic;
using SportsStore.Models;

namespace SportsStore.Models.ViewModels {

```

```

public class ProductsListViewModel {
    public IEnumerable<Product> Products { get; set; }
    public PagingInfo PagingInfo { get; set; }
}
}

```

I can update the Index action method in the HomeController class to use the ProductsListViewModel class to provide the view with details of the products to display on the page and with details of the pagination, as shown in Listing 7-30.

Listing 7-30. Updating the Action Method in the HomeController.cs File in the SportsStore/Controllers Folder

```

using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;
using System.Linq;
using SportsStore.Models.ViewModels;

namespace SportsStore.Controllers {
    public class HomeController : Controller {
        private IStoreRepository repository;
        public int PageSize = 4;

        public HomeController(IStoreRepository repo) {
            repository = repo;
        }

        public IActionResult Index(int productPage = 1)
        => View(new ProductsListViewModel {
            Products = repository.Products
            .OrderBy(p => p.ProductID)
            .Skip((productPage - 1) * PageSize)
            .Take(PageSize),
            PagingInfo = new PagingInfo {
                CurrentPage = productPage,
                ItemsPerPage = PageSize,
                TotalItems = repository.Products.Count()
            };
        });
    }
}

```

These changes pass a ProductsListViewModel object as the model data to the view.

UNIT TEST: PAGE MODEL VIEW DATA

I need to ensure that the controller sends the correct pagination data to the view. Here is the unit test I added to the HomeControllerTests class in the test project to make sure:

```

...
[Fact]
public void Can_Send_Pagination_View_Model() {
    // Arrange
    Mock<IStoreRepository> mock = new Mock<IStoreRepository>();
    mock.Setup(m => m.Products).Returns((new Product[] {
        new Product {ProductID = 1, Name = "P1"},
        new Product {ProductID = 2, Name = "P2"},
        new Product {ProductID = 3, Name = "P3"},
    }

```

```

        new Product {ProductID = 4, Name = "P4"},
        new Product {ProductID = 5, Name = "P5"}
    }).AsQueryable<Product>());
// Arrange
HomeController controller =
    new HomeController(mock.Object) { PageSize = 3 };
// Act
ProductsListViewModel result =
    controller.Index(2).ViewData.Model as ProductsListViewModel;
// Assert
PagingInfo pageInfo = result.PagingInfo;
Assert.Equal(2, pageInfo.CurrentPage);
Assert.Equal(3, pageInfo.ItemsPerPage);
Assert.Equal(5, pageInfo.TotalItems);
Assert.Equal(2, pageInfo.TotalPages);
}
...

```

I also need to modify the earlier unit tests to reflect the new result from the `Index` action method. Here are the revised tests:

```

...
[Fact]
public void Can_Use_Repository() {
    // Arrange
    Mock<IStoreRepository> mock = new Mock<IStoreRepository>();
    mock.Setup(m => m.Products).Returns((new Product[] {
        new Product {ProductID = 1, Name = "P1"},
        new Product {ProductID = 2, Name = "P2"}
    }).AsQueryable<Product>());

    HomeController controller = new HomeController(mock.Object);

    // Act
    ProductsListViewModel result =
        controller.Index().ViewData.Model as ProductsListViewModel;

    // Assert
    Product[] prodArray = result.Products.ToArray();
    Assert.True(prodArray.Length == 2);
    Assert.Equal("P1", prodArray[0].Name);
    Assert.Equal("P2", prodArray[1].Name);
}

[Fact]
public void Can_Paginate() {
    // Arrange
    Mock<IStoreRepository> mock = new Mock<IStoreRepository>();
    mock.Setup(m => m.Products).Returns((new Product[] {
        new Product {ProductID = 1, Name = "P1"},
        new Product {ProductID = 2, Name = "P2"},
        new Product {ProductID = 3, Name = "P3"},
        new Product {ProductID = 4, Name = "P4"},
        new Product {ProductID = 5, Name = "P5"}
    }).AsQueryable<Product>());

    HomeController controller = new HomeController(mock.Object);
    controller.PageSize = 3;

    // Act
    ProductsListViewModel result =
        controller.Index(2).ViewData.Model as ProductsListViewModel;

```

```

// Assert
Product[] prodArray = result.Products.ToArray();
Assert.True(prodArray.Length == 2);
Assert.Equal("P4", prodArray[0].Name);
Assert.Equal("P5", prodArray[1].Name);
}
...

```

I would usually create a common setup method, given the degree of duplication between these two test methods. However, since I am delivering the unit tests in individual sidebars like this one, I am going to keep everything separate so you can see each test on its own.

The view is currently expecting a sequence of `Product` objects, so I need to update the `Index.cshtml` file, as shown in Listing 7-31, to deal with the new view model type.

Listing 7-31. Updating the `Index.cshtml` File in the `SportsStore/Views/Home` Folder

```

@model ProductsListViewModel

@foreach (var p in Model.Products) {
    <div>
        <h3>@p.Name</h3>
        @p.Description
        <h4>@p.Price.ToString("c")</h4>
    </div>
}

```

I have changed the `@model` directive to tell Razor that I am now working with a different data type. I updated the `foreach` loop so that the data source is the `Products` property of the model data.

Displaying the Page Links

I have everything in place to add the page links to the `Index` view. I created the view model that contains the paging information, updated the controller so that it passes this information to the view, and changed the `@model` directive to match the new model view type. All that remains is to add an HTML element that the tag helper will process to create the page links, as shown in Listing 7-32.

Listing 7-32. Adding the Pagination Links in the `Index.cshtml` File in the `SportsStore/Views/Home` Folder

```

@model ProductsListViewModel

@foreach (var p in Model.Products) {
    <div>
        <h3>@p.Name</h3>
        @p.Description
        <h4>@p.Price.ToString("c")</h4>
    </div>
}

<div page-model="@Model.PagingInfo" page-action="Index"></div>

```

Restart ASP.NET Core and request `http://localhost:5000`, and you will see the new page links, as shown in Figure 7-6. The style is still basic, which I will fix later in the chapter. What is important for the moment is that the links take the user from page to page in the catalog and allow for exploration of the products for sale. When Razor finds the `page-model` attribute on the `div` element, it asks the `PageLinkTagHelper` class to transform the element, which produces the set of links shown in the figure.

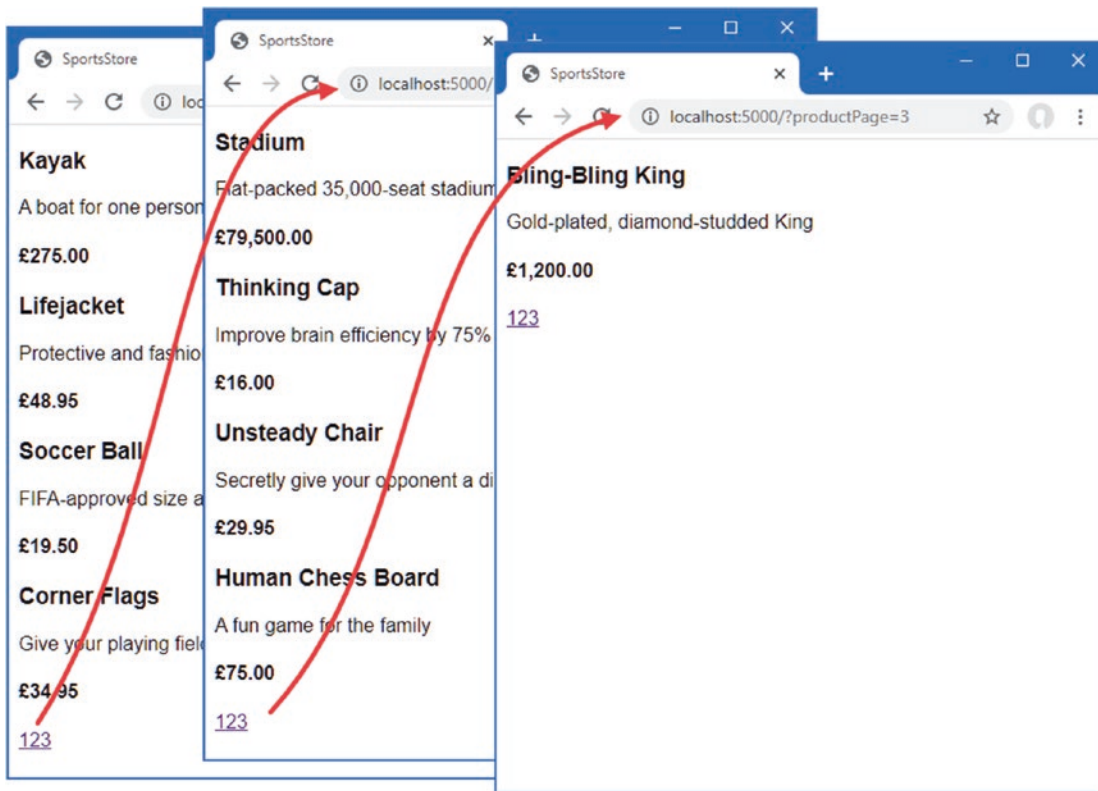


Figure 7-6. Displaying page navigation links

Improving the URLs

I have the page links working, but they still use the query string to pass page information to the server, like this:

```
http://localhost/?productPage=2
```

I can create URLs that are more appealing by creating a scheme that follows the pattern of *composable URLs*. A composable URL is one that makes sense to the user, like this one:

```
http://localhost/Page2
```

The ASP.NET Core routing feature makes it easy to change the URL scheme in an application. All I need to do is add a new route in the Startup class, as shown in Listing 7-33.

Listing 7-33. Adding a New Route in the Startup.cs File in the SportsStore Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
```

```

using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using SportsStore.Models;

namespace SportsStore {
    public class Startup {

        public Startup(IConfiguration config) {
            Configuration = config;
        }

        private IConfiguration Configuration { get; set; }

        public void ConfigureServices(IServiceCollection services) {
            services.AddControllersWithViews();

            services.AddDbContext<StoreDbContext>(opts => {
                opts.UseSqlServer(
                    Configuration["ConnectionStrings:SportsStoreConnection"]);
            });
            services.AddScoped<IStoreRepository, EFStoreRepository>();
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {

            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();

            app.UseRouting();
            app.UseEndpoints(endpoints => {
                endpoints.MapControllerRoute("pagination",
                    "Products/Page{productPage}",
                    new { Controller = "Home", action = "Index" });
                endpoints.MapDefaultControllerRoute();
            });

            SeedData.EnsurePopulated(app);
        }
    }
}

```

It is important that you add the new route before the call to the `MapDefaultControllerRoute` method. As you will learn in Chapter 13, the routing system processes routes in the order they are listed, and I need the new route to take precedence over the existing one.

This is the only alteration required to change the URL scheme for product pagination. ASP.NET Core and the routing function are tightly integrated, so the application automatically reflects a change like this in the URLs used by the application, including those generated by tag helpers like the one I use to generate the page navigation links.

Restart ASP.NET Core, request `http://localhost:5000`, and click one of the pagination links. The browser will navigate to a URL that uses the new URL scheme, as shown in Figure 7-7.

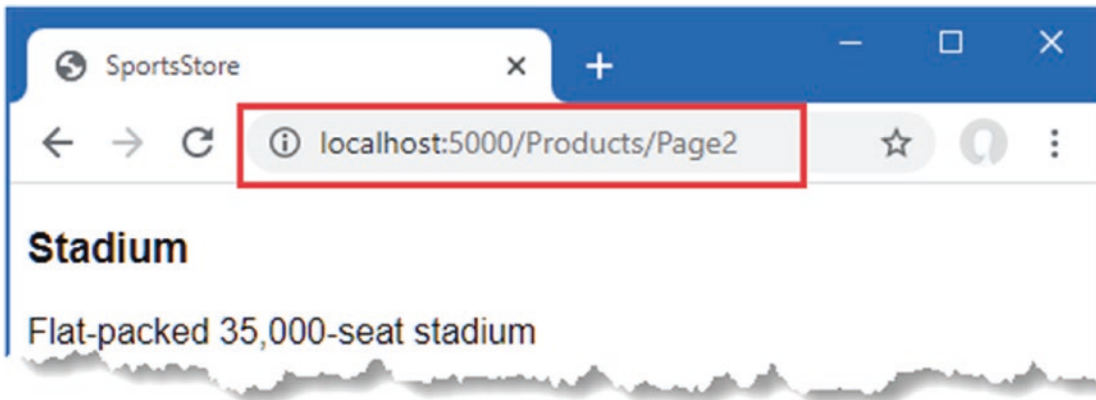


Figure 7-7. The new URL scheme displayed in the browser

Styling the Content

I have built a great deal of infrastructure, and the basic features of the application are starting to come together, but I have not paid any attention to appearance. Even though this book is not about design or CSS, the SportsStore application design is so miserably plain that it undermines its technical strengths. In this section, I will put some of that right. I am going to implement a classic two-column layout with a header, as shown in Figure 7-8.



Figure 7-8. The design goal for the SportsStore application

Installing the Bootstrap Package

I am going to use the Bootstrap package to provide the CSS styles I will apply to the application. As explained in Chapter 4, client-side packages are installed using LibMan. If you did not install the LibMan package when following the examples in Chapter 4, use a PowerShell command prompt to run the commands shown in Listing 7-34, which remove any existing LibMan package and install the version required for this book.

Listing 7-34. Installing the LibMan Tool Package

```
dotnet tool uninstall --global Microsoft.Web.LibraryManager.Cli
dotnet tool install --global Microsoft.Web.LibraryManager.Cli --version 2.0.96
```

Once you have installed LibMan, run the commands shown in Listing 7-35 in the SportsStore folder to initialize the example project and install the Bootstrap package.

Listing 7-35. Initializing the Example Project

```
libman init -p cdnjs
libman install twitter-bootstrap@4.3.1 -d wwwroot/lib/twitter-bootstrap
```

Applying Bootstrap Styles

Razor layouts provide common content so that it doesn't have to be repeated in multiple views. Add the elements shown in Listing 7-36 to the `_Layout.cshtml` file in the `Views/Shared` folder to include the Bootstrap CSS stylesheet in the content sent to the browser and define a common header that will be used throughout the SportsStore application.

Listing 7-36. Applying Bootstrap CSS to the `_Layout.cshtml` File in the `SportsStore/Views/Shared` Folder

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>SportsStore</title>
  <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
  <div class="bg-dark text-white p-2">
    <span class="navbar-brand m-1-2">SPORTS STORE</span>
  </div>
  <div class="row m-1 p-1">
    <div id="categories" class="col-3">
      Put something useful here later
    </div>
    <div class="col-9">
      @RenderBody()
    </div>
  </div>
</body>
</html>
```

Adding the Bootstrap CSS stylesheet to the layout means that I can use the styles it defines in any of the views that rely on the layout. Listing 7-37 shows the styling I applied to the `Index.cshtml` file.

Listing 7-37. Styling Content in the `Index.cshtml` File in the `SportsStore/Views/Home` Folder

```
@model ProductsListViewModel

@foreach (var p in Model.Products) {
  <div class="card card-outline-primary m-1 p-1">
    <div class="bg-faded p-1">
      <h4>
        @p.Name
      </h4>
    </div>
  </div>
}
```



```

        <span class="badge badge-pill badge-primary" style="float:right">
            <small>@p.Price.ToString("c")</small>
        </span>
    </h4>
</div>
<div class="card-text p-1">@p.Description</div>
</div>
}

<div page-model="@Model.PagingInfo" page-action="Index" page-classes-enabled="true"
    page-class="btn" page-class-normal="btn-outline-dark"
    page-class-selected="btn-primary" class="btn-group pull-right m-1">
</div>

```

I need to style the buttons generated by the `PageLinkTagHelper` class, but I don't want to hardwire the Bootstrap classes into the C# code because it makes it harder to reuse the tag helper elsewhere in the application or change the appearance of the buttons. Instead, I have defined custom attributes on the `div` element that specify the classes that I require, and these correspond to properties I added to the tag helper class, which are then used to style the `a` elements that are produced, as shown in Listing 7-38.

Listing 7-38. Adding Classes to Elements in the `PageLinkTagHelper.cs` File in the `SportsStore/Infrastructure` Folder

```

using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.AspNetCore.Mvc.Routing;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using Microsoft.AspNetCore.Razor.TagHelpers;
using SportsStore.Models.ViewModels;

namespace SportsStore.Infrastructure {

    [HtmlTargetElement("div", Attributes = "page-model")]
    public class PageLinkTagHelper : TagHelper {
        private IUrlHelperFactory urlHelperFactory;

        public PageLinkTagHelper(IUrlHelperFactory helperFactory) {
            urlHelperFactory = helperFactory;
        }

        [ViewContext]
        [HtmlAttributeNotBound]
        public ViewContext ViewContext { get; set; }

        public PagingInfo PageModel { get; set; }

        public string PageAction { get; set; }

        public bool PageClassesEnabled { get; set; } = false;
        public string PageClass { get; set; }
        public string PageClassNormal { get; set; }
        public string PageClassSelected { get; set; }

        public override void Process(TagHelperContext context,
            TagHelperOutput output) {
            IUrlHelper urlHelper = urlHelperFactory.GetUrlHelper(ViewContext);
            TagBuilder result = new TagBuilder("div");
            for (int i = 1; i <= PageModel.TotalPages; i++) {
                TagBuilder tag = new TagBuilder("a");
            }
        }
    }
}

```

```

tag.Attributes["href"] = urlHelper.Action(PageAction,
    new { productPage = i });
if (PageClassesEnabled) {
    tag.AddCssClass(PageClass);
    tag.AddCssClass(i == PageModel.CurrentPage
        ? PageClassSelected : PageClassNormal);
}
tag.InnerHtml.Append(i.ToString());
result.InnerHtml.AppendHtml(tag);
}
output.Content.AppendHtml(result.InnerHtml);
}
}
}

```

The values of the attributes are automatically used to set the tag helper property values, with the mapping between the HTML attribute name format (`page-class-normal`) and the C# property name format (`PageClassNormal`) taken into account. This allows tag helpers to respond differently based on the attributes of an HTML element, creating a more flexible way to generate content in an ASP.NET Core application.

Restart ASP.NET Core and request `http://localhost:5000`, and you will see the appearance of the application has been improved—at least a little, anyway—as illustrated by Figure 7-9.

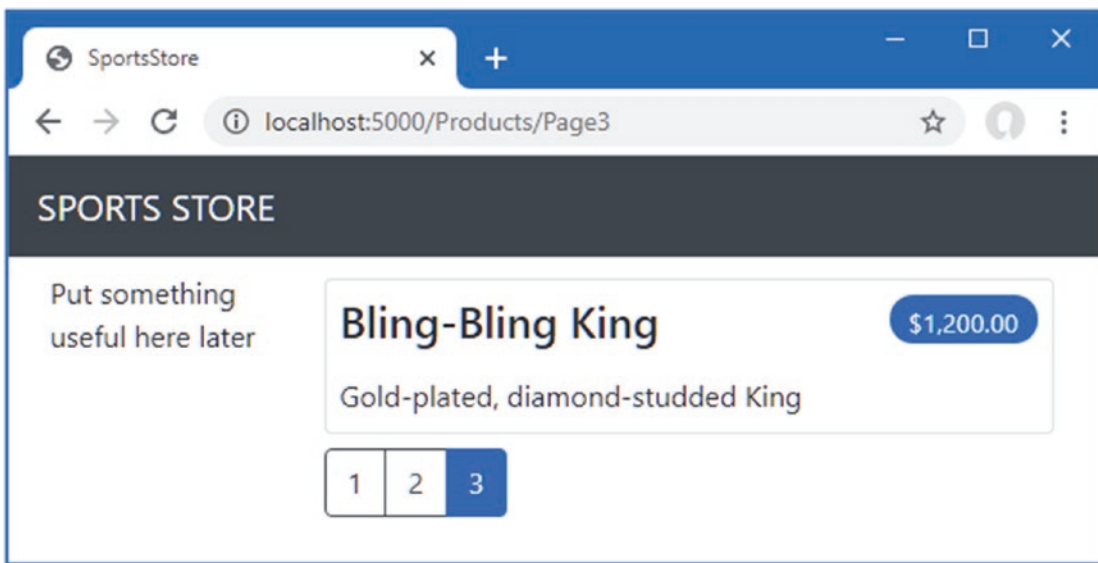


Figure 7-9. Applying styles to the SportsStore application

Creating a Partial View

As a finishing flourish for this chapter, I am going to refactor the application to simplify the `Index.cshtml` view. I am going to create a *partial view*, which is a fragment of content that you can embed into another view, rather like a template. I describe partial views in detail in Chapter 22, and they help reduce duplication when you need the same content to appear in different places in an application. Rather than copy and paste the same Razor markup into multiple views, you can define it once in a partial view. To create the partial view, I added a Razor View called `ProductSummary.cshtml` to the `Views/Shared` folder and added the markup shown in Listing 7-39.

Listing 7-39. The Contents of the ProductSummary.cshtml File in the SportsStore/Views/Shared Folder

```
@model Product

<div class="card card-outline-primary m-1 p-1">
  <div class="bg-faded p-1">
    <h4>
      @Model.Name
      <span class="badge badge-pill badge-primary" style="float:right">
        <small>@Model.Price.ToString("c")</small>
      </span>
    </h4>
  </div>
  <div class="card-text p-1">@Model.Description</div>
</div>
```

Now I need to update the Index.cshtml file in the Views/Home folder so that it uses the partial view, as shown in Listing 7-40.

Listing 7-40. Using a Partial View in the Index.cshtml File in the SportsStore/Views/Home Folder

```
@model ProductsListViewModel

@foreach (var p in Model.Products) {
  <partial name="ProductSummary" model="p" />
}

<div page-model="@Model.PagingInfo" page-action="Index" page-classes-enabled="true"
  page-class="btn" page-class-normal="btn-outline-dark"
  page-class-selected="btn-primary" class="btn-group pull-right m-1">
</div>
```

I have taken the markup that was previously in the @foreach expression in the Index.cshtml view and moved it to the new partial view. I call the partial view using a partial element, using the name and model attributes to specify the name of the partial view and its view model. Using a partial view allows the same markup to be inserted into any view that needs to display a summary of a product.

Restart ASP.NET Core and request `http://localhost:5000`, and you will see that introducing the partial view doesn't change the appearance of the application; it just changes where Razor finds the content that is used to generate the response sent to the browser.

Summary

In this chapter, I built the core infrastructure for the SportsStore application. It does not have many features that you could demonstrate to a client at this point, but behind the scenes, there are the beginnings of a domain model with a product repository backed by SQL Server and Entity Framework Core. There is a single controller, `HomeController`, that can produce paginated lists of products, and I have set up a clean and friendly URL scheme.

If this chapter felt like a lot of setup for little benefit, then the next chapter will balance the equation. Now that the fundamental structure is in place, we can forge ahead and add all the customer-facing features: navigation by category and the start of a shopping cart.

CHAPTER 8



SportsStore: Navigation and Cart

In this chapter, I continue to build out the SportsStore example app. I add support for navigating around the application and start building a shopping cart.

■ **Tip** You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/apress/pro-asp.net-core-3>. See Chapter 1 for how to get help if you have problems running the examples.

Adding Navigation Controls

The SportsStore application will be more useful if customers can navigate products by category. I will do this in three phases.

- Enhance the Index action method in the HomeController class so that it can filter the Product objects in the repository
- Revisit and enhance the URL scheme
- Create a category list that will go into the sidebar of the site, highlighting the current category and linking to others

Filtering the Product List

I am going to start by enhancing the view model class, ProductsListViewModel, which I added to the SportsStore project in the previous chapter. I need to communicate the current category to the view to render the sidebar, and this is as good a place to start as any. Listing 8-1 shows the changes I made to the ProductsListViewModel.cs file in the Models/ViewModels folder.

Listing 8-1. Modifying the ProductsListViewModel.cs File in the SportsStore/Models/ViewModels Folder

```
using System.Collections.Generic;
using SportsStore.Models;

namespace SportsStore.Models.ViewModels {

    public class ProductsListViewModel {
        public IEnumerable<Product> Products { get; set; }
        public PagingInfo PagingInfo { get; set; }
        public string CurrentCategory { get; set; }
    }
}
```

I added a property called CurrentCategory. The next step is to update the Home controller so that the Index action method will filter Product objects by category and use the property I added to the view model to indicate which category has been selected, as shown in Listing 8-2.

Listing 8-2. Adding Category Support in the HomeController.cs File in the SportsStore/Controllers Folder

```

using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;
using System.Linq;
using SportsStore.Models.ViewModels;

namespace SportsStore.Controllers {
    public class HomeController : Controller {
        private IStoreRepository repository;
        public int PageSize = 4;

        public HomeController(IStoreRepository repo) {
            repository = repo;
        }

        public IActionResult Index(string category, int productPage = 1)
        => View(new ProductsListViewModel {
            Products = repository.Products
                .Where(p => category == null || p.Category == category)
                .OrderBy(p => p.ProductID)
                .Skip((productPage - 1) * PageSize)
                .Take(PageSize),
            PagingInfo = new PagingInfo {
                CurrentPage = productPage,
                ItemsPerPage = PageSize,
                TotalItems = repository.Products.Count()
            },
            CurrentCategory = category
        });
    }
}

```

I made three changes to the action method. First, I added a parameter called `category`. This `category` parameter is used by the second change in the listing, which is an enhancement to the LINQ query: if `category` is not `null`, only those `Product` objects with a matching `Category` property are selected. The last change is to set the value of the `CurrentCategory` property I added to the `ProductsListViewModel` class. However, these changes mean that the value of `PagingInfo.TotalItems` is incorrectly calculated because it doesn't take the `category` filter into account. I will fix this in a while.

UNIT TEST: UPDATING EXISTING UNIT TESTS

I changed the signature of the `Index` action method, which will prevent some of the existing unit test methods from compiling. To address this, I need to pass `null` as the first parameter to the `Index` method in those unit tests that work with the controller. For example, in the `Can_Use_Repository` test in the `HomeControllerTests.cs` file, the action section of the unit test becomes as follows:

```

...
[Fact]
public void Can_Use_Repository() {
    // Arrange
    Mock<IStoreRepository> mock = new Mock<IStoreRepository>();
    mock.Setup(m => m.Products).Returns((new Product[] {
        new Product {ProductID = 1, Name = "P1"},
        new Product {ProductID = 2, Name = "P2"}
    })).AsQueryable<Product>();

    HomeController controller = new HomeController(mock.Object);
}

```

```

// Act
ProductsListViewModel result =
    controller.Index(null).ViewData.Model as ProductsListViewModel;

// Assert
Product[] prodArray = result.Products.ToArray();
Assert.True(prodArray.Length == 2);
Assert.Equal("P1", prodArray[0].Name);
Assert.Equal("P2", prodArray[1].Name);
}
...

```

By using null for the category argument, I receive all the Product objects that the controller gets from the repository, which is the same situation I had before adding the new parameter. I need to make the same change to the Can_Paginate and Can_Send_Pagination_View_Model tests as well.

```

...
[Fact]
public void Can_Paginate() {
    // Arrange
    Mock<IStoreRepository> mock = new Mock<IStoreRepository>();
    mock.Setup(m => m.Products).Returns((new Product[] {
        new Product {ProductID = 1, Name = "P1"},
        new Product {ProductID = 2, Name = "P2"},
        new Product {ProductID = 3, Name = "P3"},
        new Product {ProductID = 4, Name = "P4"},
        new Product {ProductID = 5, Name = "P5"}
    }).AsQueryable<Product>());

    HomeController controller = new HomeController(mock.Object);
    controller.PageSize = 3;

    // Act
    ProductsListViewModel result =
        controller.Index(null, 2).ViewData.Model as ProductsListViewModel;

    // Assert
    Product[] prodArray = result.Products.ToArray();

    Assert.True(prodArray.Length == 2);
    Assert.Equal("P4", prodArray[0].Name);
    Assert.Equal("P5", prodArray[1].Name);
}

[Fact]
public void Can_Send_Pagination_View_Model() {
    // Arrange
    Mock<IStoreRepository> mock = new Mock<IStoreRepository>();
    mock.Setup(m => m.Products).Returns((new Product[] {
        new Product {ProductID = 1, Name = "P1"},
        new Product {ProductID = 2, Name = "P2"},
        new Product {ProductID = 3, Name = "P3"},
        new Product {ProductID = 4, Name = "P4"},
        new Product {ProductID = 5, Name = "P5"}
    }).AsQueryable<Product>());

    // Arrange
    HomeController controller =
        new HomeController(mock.Object) { PageSize = 3 };
}

```

```

// Act
ProductsListViewModel result =
    controller.Index(null, 2).ViewData.Model as ProductsListViewModel;

// Assert
PagingInfo pageInfo = result.PagingInfo;
Assert.Equal(2, pageInfo.CurrentPage);
Assert.Equal(3, pageInfo.ItemsPerPage);
Assert.Equal(5, pageInfo.TotalItems);
Assert.Equal(2, pageInfo.TotalPages);
}
...

```

Keeping your unit tests synchronized with your code changes quickly becomes second nature when you get into the testing mindset.

To see the effect of the category filtering, start ASP.NET Core and select a category using the following URL, taking care to use an uppercase S for Soccer:

```
http://localhost:5000/?category=Soccer
```

You will see only the products in the Soccer category, as shown in Figure 8-1.

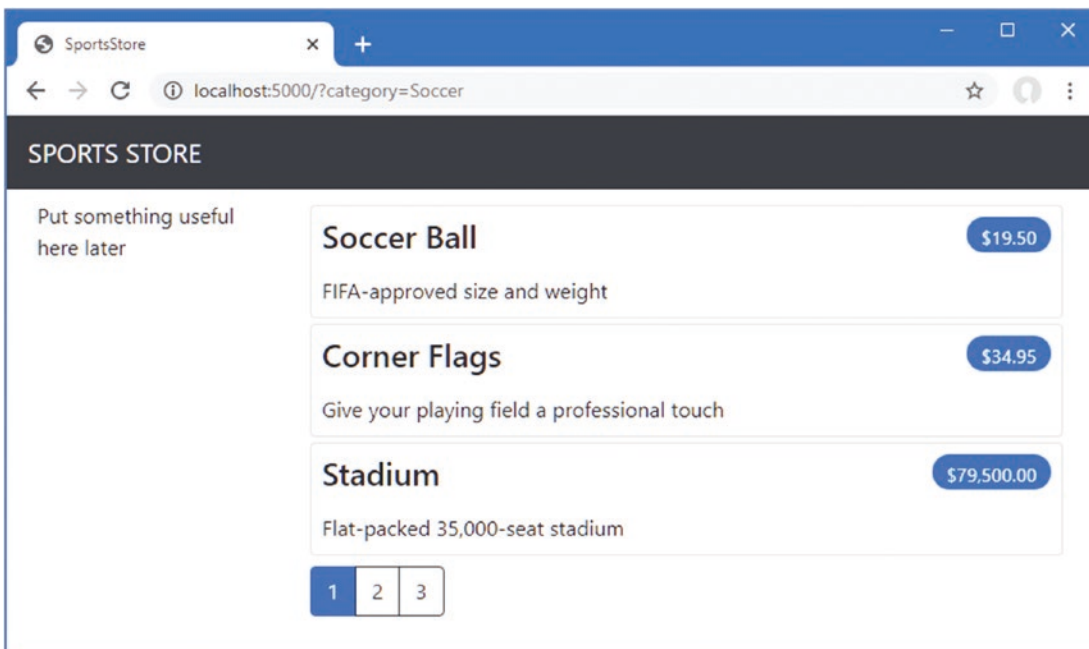


Figure 8-1. Using the query string to filter by category

Obviously, users won't want to navigate to categories using URLs, but you can see how small changes can have a big impact once the basic structure of an ASP.NET Core application is in place.

UNIT TEST: CATEGORY FILTERING

I need a unit test to properly test the category filtering function to ensure that the filter can correctly generate products in a specified category. Here is the test method I added to the `ProductControllerTests` class:

```
...
[Fact]
public void Can_Filter_Products() {
    // Arrange
    // - create the mock repository
    Mock<IStoreRepository> mock = new Mock<IStoreRepository>();
    mock.Setup(m => m.Products).Returns((new Product[] {
        new Product {ProductID = 1, Name = "P1", Category = "Cat1"},
        new Product {ProductID = 2, Name = "P2", Category = "Cat2"},
        new Product {ProductID = 3, Name = "P3", Category = "Cat1"},
        new Product {ProductID = 4, Name = "P4", Category = "Cat2"},
        new Product {ProductID = 5, Name = "P5", Category = "Cat3"}
    }).AsQueryable<Product>());

    // Arrange - create a controller and make the page size 3 items
    HomeController controller = new HomeController(mock.Object);
    controller.PageSize = 3;

    // Action
    Product[] result =
        (controller.Index("Cat2", 1).ViewData.Model as ProductsListViewModel)
            .Products.ToArray();

    // Assert
    Assert.Equal(2, result.Length);
    Assert.True(result[0].Name == "P2" && result[0].Category == "Cat2");
    Assert.True(result[1].Name == "P4" && result[1].Category == "Cat2");
}
...
```

This test creates a mock repository containing `Product` objects that belong to a range of categories. One specific category is requested using the action method, and the results are checked to ensure that the results are the right objects in the right order.

Refining the URL Scheme

No one wants to see or use ugly URLs such as `/?category=Soccer`. To address this, I am going to change the routing configuration in the `Configure` method of the `Startup` class to create a more useful set of URLs, as shown in Listing 8-3.

■ **Caution** It is important to add the new routes in Listing 8-3 in the order they are shown. Routes are applied in the order in which they are defined, and you will get some odd effects if you change the order.

Listing 8-3. Changing the Routing Schema in the `Startup.cs` File in the `SportsStore` Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
```



```

using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using SportsStore.Models;

namespace SportsStore {
    public class Startup {

        public Startup(IConfiguration config) {
            Configuration = config;
        }

        private IConfiguration Configuration { get; set; }

        public void ConfigureServices(IServiceCollection services) {
            services.AddControllersWithViews();

            services.AddDbContext<StoreDbContext>(opts => {
                opts.UseSqlServer(
                    Configuration["ConnectionStrings:SportsStoreConnection"]);
            });
            services.AddScoped<IStoreRepository, EFStoreRepository>();
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();

            app.UseRouting();
            app.UseEndpoints(endpoints => {
                endpoints.MapControllerRoute("catpage",
                    "{category}/Page{productPage:int}",
                    new { Controller = "Home", action = "Index" });

                endpoints.MapControllerRoute("page", "Page{productPage:int}",
                    new { Controller = "Home", action = "Index", productPage = 1 });

                endpoints.MapControllerRoute("category", "{category}",
                    new { Controller = "Home", action = "Index", productPage = 1 });

                endpoints.MapControllerRoute("pagination",
                    "Products/Page{productPage}",
                    new { Controller = "Home", action = "Index", productPage = 1 });
                endpoints.MapDefaultControllerRoute();
            });

            SeedData.EnsurePopulated(app);
        }
    }
}

```

Table 8-1 describes the URL scheme that these routes represent. I explain the routing system in detail in Chapter 13.

Table 8-1. Route Summary

URL	Leads To
/	Lists the first page of products from all categories
/Page2	Lists the specified page (in this case, page 2), showing items from all categories
/Soccer	Shows the first page of items from a specific category (in this case, the Soccer category)
/Soccer/Page2	Shows the specified page (in this case, page 2) of items from the specified category (in this case, Soccer)

The ASP.NET Core routing system handles *incoming* requests from clients, but it also generates *outgoing* URLs that conform to the URL scheme and that can be embedded in web pages. By using the routing system both to handle incoming requests and to generate outgoing URLs, I can ensure that all the URLs in the application are consistent.

The `IUrlHelper` interface provides access to URL-generating functionality. I used this interface and the `Action` method it defines in the tag helper I created in the previous chapter. Now that I want to start generating more complex URLs, I need a way to receive additional information from the view without having to add extra properties to the tag helper class. Fortunately, tag helpers have a nice feature that allows properties with a common prefix to be received all together in a single collection, as shown in Listing 8-4.

Listing 8-4. Prefixed Values in the `PageLinkTagHelper.cs` File in the `SportsStore/Infrastructure` Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.AspNetCore.Mvc.Routing;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using Microsoft.AspNetCore.Razor.TagHelpers;
using SportsStore.Models.ViewModels;
using System.Collections.Generic;

namespace SportsStore.Infrastructure {

    [HtmlTargetElement("div", Attributes = "page-model")]
    public class PageLinkTagHelper : TagHelper {
        private IUrlHelperFactory urlHelperFactory;

        public PageLinkTagHelper(IUrlHelperFactory helperFactory) {
            urlHelperFactory = helperFactory;
        }

        [ViewContext]
        [HtmlAttributeNotBound]
        public ViewContext ViewContext { get; set; }

        public PagingInfo PageModel { get; set; }

        public string PageAction { get; set; }

        [HtmlAttributeName(DictionaryAttributePrefix = "page-url-")]
        public Dictionary<string, object> PageUrlValues { get; set; }
        = new Dictionary<string, object>();

        public bool PageClassesEnabled { get; set; } = false;
        public string PageClass { get; set; }
        public string PageClassNormal { get; set; }
        public string PageClassSelected { get; set; }

        public override void Process(TagHelperContext context,
            TagHelperOutput output) {
            IUrlHelper urlHelper = urlHelperFactory.GetUrlHelper(ViewContext);
```

```

    TagBuilder result = new TagBuilder("div");
    for (int i = 1; i <= PageModel.TotalPages; i++) {
        TagBuilder tag = new TagBuilder("a");
        PageUrlValues["productPage"] = i;
        tag.Attributes["href"] = urlHelper.Action(PageAction, PageUrlValues);
        if (PageClassesEnabled) {
            tag.AddCssClass(PageClass);
            tag.AddCssClass(i == PageModel.CurrentPage
                ? PageClassSelected : PageClassNormal);
        }
        tag.InnerHtml.Append(i.ToString());
        result.InnerHtml.AppendHtml(tag);
    }
    output.Content.AppendHtml(result.InnerHtml);
}
}
}

```

Decorating a tag helper property with the `HtmlAttributeName` attribute allows me to specify a prefix for attribute names on the element, which in this case will be `page-url-`. The value of any attribute whose name begins with this prefix will be added to the dictionary that is assigned to the `PageUrlValues` property, which is then passed to the `IUrlHelper.Action` method to generate the URL for the `href` attribute of the `a` elements that the tag helper produces.

In Listing 8-5, I have added a new attribute to the `div` element that is processed by the tag helper, specifying the category that will be used to generate the URL. I have added only one new attribute to the view, but any attribute with the same prefix would be added to the dictionary.

Listing 8-5. Adding a New Attribute in the `Index.cshtml` File in the `SportsStore/Views/Home` Folder

```

@model ProductsListViewModel

@foreach (var p in Model.Products) {
    <partial name="ProductSummary" model="p" />
}

<div page-model="@Model.PagingInfo" page-action="Index" page-classes-enabled="true"
    page-class="btn" page-class-normal="btn-outline-dark"
    page-class-selected="btn-primary" page-url-category="@Model.CurrentCategory"
    class="btn-group pull-right m-1">
</div>

```

Prior to this change, the links generated for the pagination links looked like this:

<http://localhost:5000/Page1>

If the user clicked a page link like this, the category filter would be lost, and the application would present a page containing products from all categories. By adding the current category, taken from the view model, I generate URLs like this instead:

<http://localhost:5000/Chess/Page1>

When the user clicks this kind of link, the current category will be passed to the `Index` action method, and the filtering will be preserved. To see the effect of this change, start ASP.NET Core and request `http://localhost:5000/Chess`, which will display just the products in the `Chess` category, as shown in Figure 8-2.

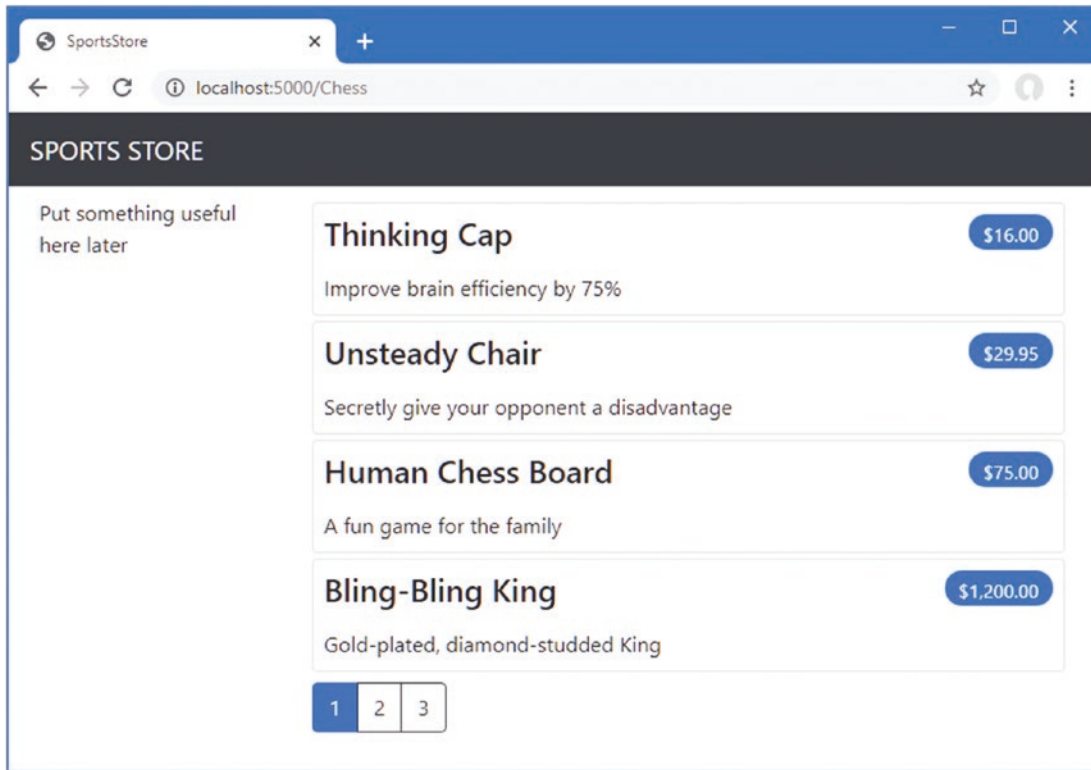


Figure 8-2. Filtering data by category

Building a Category Navigation Menu

I need to provide users with a way to select a category that does not involve typing in URLs. This means presenting a list of the available categories and indicating which, if any, is currently selected.

ASP.NET Core has the concept of *view components*, which are perfect for creating items such as reusable navigation controls. A view component is a C# class that provides a small amount of reusable application logic with the ability to select and display Razor partial views. I describe view components in detail in Chapter 24.

In this case, I will create a view component that renders the navigation menu and integrate it into the application by invoking the component from the shared layout. This approach gives me a regular C# class that can contain whatever application logic I need and that can be unit tested like any other class.

Creating the Navigation View Component

I created a folder called `Components`, which is the conventional home of view components, in the `SportsStore` project and added to it a class file named `NavigationMenuViewComponent.cs`, which I used to define the class shown in Listing 8-6.

Listing 8-6. The Contents of the `NavigationMenuViewComponent.cs` File in the `SportsStore/Components` Folder

```
using Microsoft.AspNetCore.Mvc;

namespace SportsStore.Components {
    public class NavigationMenuViewComponent : ViewComponent {
        public string Invoke() {
            return "Hello from the Nav View Component";
        }
    }
}
```

The view component's `Invoke` method is called when the component is used in a Razor view, and the result of the `Invoke` method is inserted into the HTML sent to the browser. I have started with a simple view component that returns a string, but I'll replace this with HTML shortly.

I want the category list to appear on all pages, so I am going to use the view component in the shared layout, rather than in a specific view. Within a view, view components are applied using a tag helper, as shown in Listing 8-7.

Listing 8-7. Using a View Component in the `_Layout.cshtml` File in the `SportsStore/Views/Shared` Folder

```
<!DOCTYPE html>
<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <title>SportsStore</title>
  <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
  <div class="bg-dark text-white p-2">
    <span class="navbar-brand ml-2">SPORTS STORE</span>
  </div>
  <div class="row m-1 p-1">
    <div id="categories" class="col-3">
      <vc:navigation-menu />
    </div>
    <div class="col-9">
      @RenderBody()
    </div>
  </div>
</body>
</html>
```

I removed the placeholder text and replaced it with the `vc:navigation-menu` element, which inserts the view component. The element omits the `ViewComponent` part of the class name and hyphenates it, such that `vc:navigation-menu` specifies the `NavigationMenuViewComponent` class.

Restart ASP.NET Core and request `http://localhost:5000`, and you will see that the output from the `Invoke` method is included in the HTML sent to the browser, as shown in Figure 8-3.

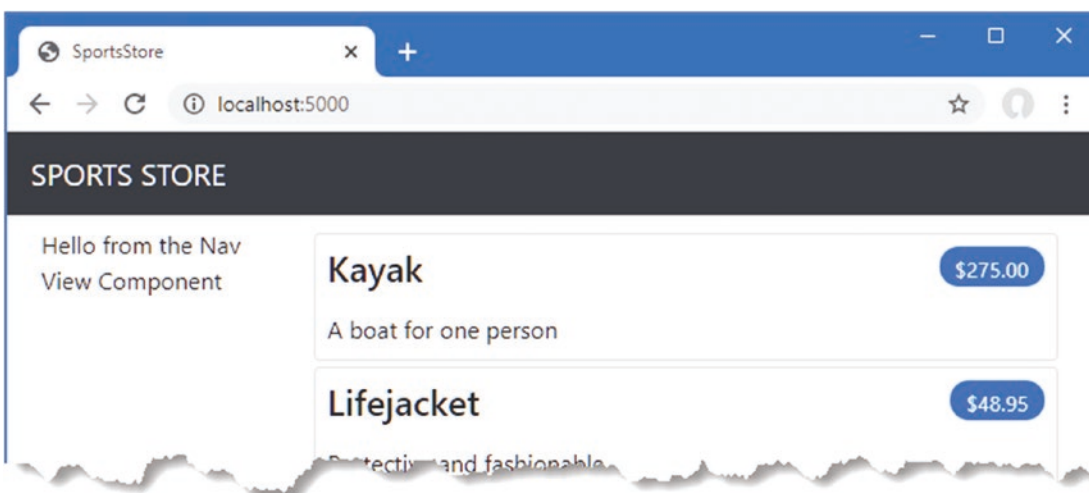


Figure 8-3. Using a view component

Generating Category Lists

I can now return to the navigation view component and generate a real set of categories. I could build the HTML for the categories programmatically, as I did for the page tag helper, but one of the benefits of working with view components is they can render Razor partial views. That means I can use the view component to generate the list of components and then use the more expressive Razor syntax to render the HTML that will display them. The first step is to update the view component, as shown in Listing 8-8.

Listing 8-8. Adding Categories in the NavigationMenuViewComponent.cs File in the SportsStore/Components Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Linq;
using SportsStore.Models;

namespace SportsStore.Components {

    public class NavigationMenuViewComponent : ViewComponent {
        private IStoreRepository repository;

        public NavigationMenuViewComponent(IStoreRepository repo) {
            repository = repo;
        }

        public IViewComponentResult Invoke() {
            return View(repository.Products
                .Select(x => x.Category)
                .Distinct()
                .OrderBy(x => x));
        }
    }
}
```

The constructor defined in Listing 8-8 defines an `IStoreRepository` parameter. When ASP.NET Core needs to create an instance of the view component class, it will note the need to provide a value for this parameter and inspect the configuration in the `Startup` class to determine which implementation object should be used. This is the same dependency injection feature that I used in the controller in Chapter 7, and it has the same effect, which is to allow the view component to access data without knowing which repository implementation will be used, a feature I describe in detail in Chapter 14.

In the `Invoke` method, I use LINQ to select and order the set of categories in the repository and pass them as the argument to the `View` method, which renders the default Razor partial view, details of which are returned from the method using an `IViewComponentResult` object, a process I describe in more detail in Chapter 24.

UNIT TEST: GENERATING THE CATEGORY LIST

The unit test for my ability to produce a category list is relatively simple. The goal is to create a list that is sorted in alphabetical order and contains no duplicates, and the simplest way to do this is to supply some test data that *does* have duplicate categories and that is *not* in order, pass this to the tag helper class, and assert that the data has been properly cleaned up. Here is the unit test, which I defined in a new class file called `NavigationMenuViewComponentTests.cs` in the `SportsStore.Tests` project:

```
using System.Collections.Generic;
using System.Linq;
using Microsoft.AspNetCore.Components;
using Microsoft.AspNetCore.Mvc.Rendering;
using Microsoft.AspNetCore.Mvc.ViewComponents;
using Moq;
using SportsStore.Components;
using SportsStore.Models;
using Xunit;
```

```

namespace SportsStore.Tests {
    public class NavigationMenuViewComponentTests {
        [Fact]
        public void Can_Select_Categories() {
            // Arrange
            Mock<IStoreRepository> mock = new Mock<IStoreRepository>();
            mock.Setup(m => m.Products).Returns((new Product[] {
                new Product {ProductID = 1, Name = "P1", Category = "Apples"},
                new Product {ProductID = 2, Name = "P2", Category = "Apples"},
                new Product {ProductID = 3, Name = "P3", Category = "Plums"},
                new Product {ProductID = 4, Name = "P4", Category = "Oranges"},
            }).AsQueryable<Product>());

            NavigationMenuViewComponent target =
                new NavigationMenuViewComponent(mock.Object);

            // Act = get the set of categories
            string[] results = ((IEnumerable<string>)(target.Invoke()
                as ViewViewComponentResult).ViewData.Model).ToArray();

            // Assert
            Assert.True(Enumerable.SequenceEqual(new string[] { "Apples",
                "Oranges", "Plums" }, results));
        }
    }
}

```

I created a mock repository implementation that contains repeating categories and categories that are not in order. I assert that the duplicates are removed and that alphabetical ordering is imposed.

Creating the View

Razor uses different conventions for locating with views that are selected by view components. Both the default name of the view and the locations that are searched for the view are different from those used for controllers. To that end, I created the Views/Shared/Components/NavigationMenu folder in the SportsStore project and added to it a Razor view named Default.cshtml, to which I added the content shown in Listing 8-9.

Listing 8-9. The Contents of the Default.cshtml File in the SportsStore/Views/Shared/Components/NavigationMenu Folder

```

@model IEnumerable<string>

<a class="btn btn-block btn-outline-secondary" asp-action="Index"
    asp-controller="Home" asp-route-category="">
    Home
</a>

@foreach (string category in Model) {
    <a class="btn btn-block btn-outline-secondary"
        asp-action="Index" asp-controller="Home"
        asp-route-category="@category"
        asp-route-productPage="1">
        @category
    </a>
}

```

This view uses one of the built-in tag helpers, which I describe in Chapters 25–27, to create anchor elements whose href attribute contains a URL that selects a different product category.

Restart ASP.NET Core and request `http://localhost:5000` to see the category navigation buttons. If you click a button, the list of items is updated to show only items from the selected category, as shown in Figure 8-4.

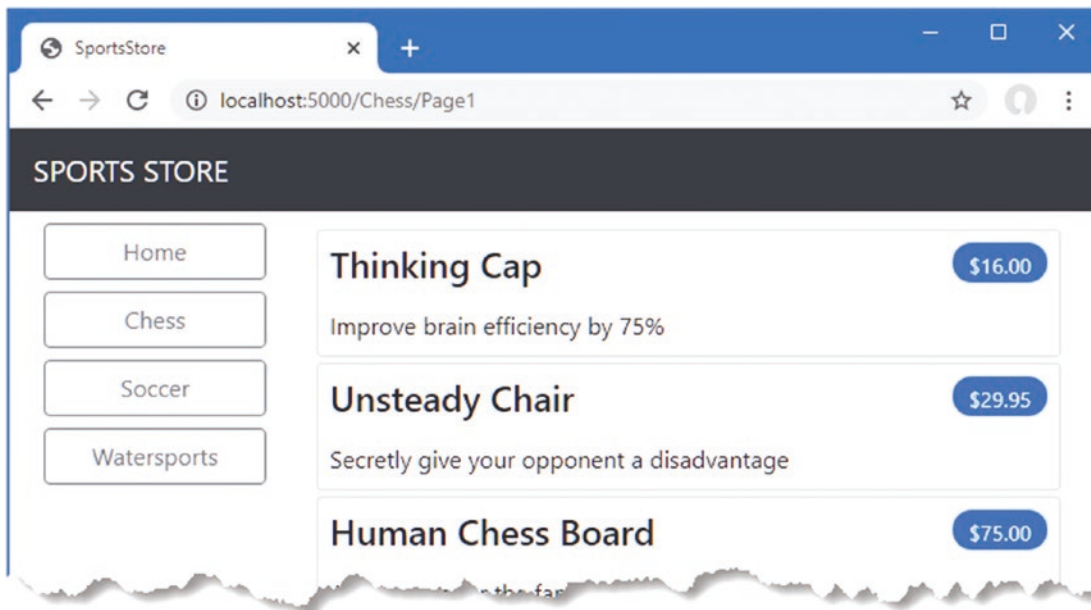


Figure 8-4. Generating category links with a view component

Highlighting the Current Category

There is no feedback to the user to indicate which category has been selected. It might be possible to infer the category from the items in the list, but some clear visual feedback seems like a good idea. ASP.NET Core components such as controllers and view components can receive information about the current request by asking for a context object. Most of the time, you can rely on the base classes that you use to create components to take care of getting the context object for you, such as when you use the `Controller` base class to create controllers.

The `ViewComponent` base class is no exception and provides access to context objects through a set of properties. One of the properties is called `RouteData`, which provides information about how the request URL was handled by the routing system.

In Listing 8-10, I use the `RouteData` property to access the request data in order to get the value for the currently selected category. I could pass the category to the view by creating another view model class (and that's what I would do in a real project), but for variety, I am going to use the view bag feature, which allows unstructured data to be passed to a view alongside the view model object. I describe how this feature works in detail in Chapter 22.

Listing 8-10. Passing the Selected Category in the `NavigationMenuViewComponent.cs` File in the `SportsStore/Components` Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Linq;
using SportsStore.Models;

namespace SportsStore.Components {

    public class NavigationMenuViewComponent : ViewComponent {
        private IStoreRepository repository;

        public NavigationMenuViewComponent(IStoreRepository repo) {
            repository = repo;
        }
    }
}
```



```

public IActionResult Invoke() {
    ViewBag.SelectedCategory = RouteData?.Values["category"];
    return View(repository.Products
        .Select(x => x.Category)
        .Distinct()
        .OrderBy(x => x));
}
}
}

```

Inside the `Invoke` method, I have dynamically assigned a `SelectedCategory` property to the `ViewBag` object and set its value to be the current category, which is obtained through the context object returned by the `RouteData` property. The `ViewBag` is a dynamic object that allows me to define new properties simply by assigning values to them.

UNIT TEST: REPORTING THE SELECTED CATEGORY

I can test that the view component correctly adds details of the selected category by reading the value of the `ViewBag` property in a unit test, which is available through the `ViewViewComponentResult` class. Here is the test, which I added to the `NavigationMenuViewComponentTests` class:

```

...
[Fact]
public void Indicates_Selected_Category() {
    // Arrange
    string categoryToSelect = "Apples";
    Mock<IStoreRepository> mock = new Mock<IStoreRepository>();
    mock.Setup(m => m.Products).Returns((new Product[] {
        new Product {ProductID = 1, Name = "P1", Category = "Apples"},
        new Product {ProductID = 4, Name = "P2", Category = "Oranges"},
    })).AsQueryable<Product>());

    NavigationMenuViewComponent target =
        new NavigationMenuViewComponent(mock.Object);
    target.ViewComponentContext = new ViewComponentContext {
        ViewContext = new ViewContext {
            RouteData = new Microsoft.AspNetCore.Routing.RouteData()
        }
    };
    target.RouteData.Values["category"] = categoryToSelect;

    // Action
    string result = (string)(target.Invoke() as
        ViewViewComponentResult).ViewData["SelectedCategory"];

    // Assert
    Assert.Equal(categoryToSelect, result);
}
...

```

This unit test provides the view component with routing data through the `ViewComponentContext` property, which is how view components receive all their context data. The `ViewComponentContext` property provides access to view-specific context data through its `ViewContext` property, which in turn provides access to the routing information through its `RouteData` property. Most of the code in the unit test goes into creating the context objects that will provide the selected category in the same way that it would be presented when the application is running and the context data is provided by ASP.NET Core MVC.

Now that I am providing information about which category is selected, I can update the view selected by the view component and vary the CSS classes used to style the links so that the one representing the current category is distinct. Listing 8-11 shows the change I made to the `Default.cshtml` file.

Listing 8-11. Highlighting in the Default.cshtml File in the SportsStore/Views/Shared/Components/NavigationMenu Folder

```
@model IEnumerable<string>

<a class="btn btn-block btn-outline-secondary"asp-action="Index"
  asp-controller="Home" asp-route-category="">
  Home
</a>

@foreach (string category in Model) {
  <a class="btn btn-block
    @(category == ViewBag.SelectedCategory
      ? "btn-primary": "btn-outline-secondary")"
    asp-action="Index" asp-controller="Home"
    asp-route-category="@category"
    asp-route-productPage="1">
    @category
  </a>
}
```

I have used a Razor expression within the class attribute to apply the `btn-primary` class to the element that represents the selected category and the `btn-secondary` class otherwise. These classes apply different Bootstrap styles and make the active button obvious, which you can see by restarting ASP.NET Core, requesting `http://localhost:5000`, and clicking one of the category buttons, as shown in Figure 8-5.

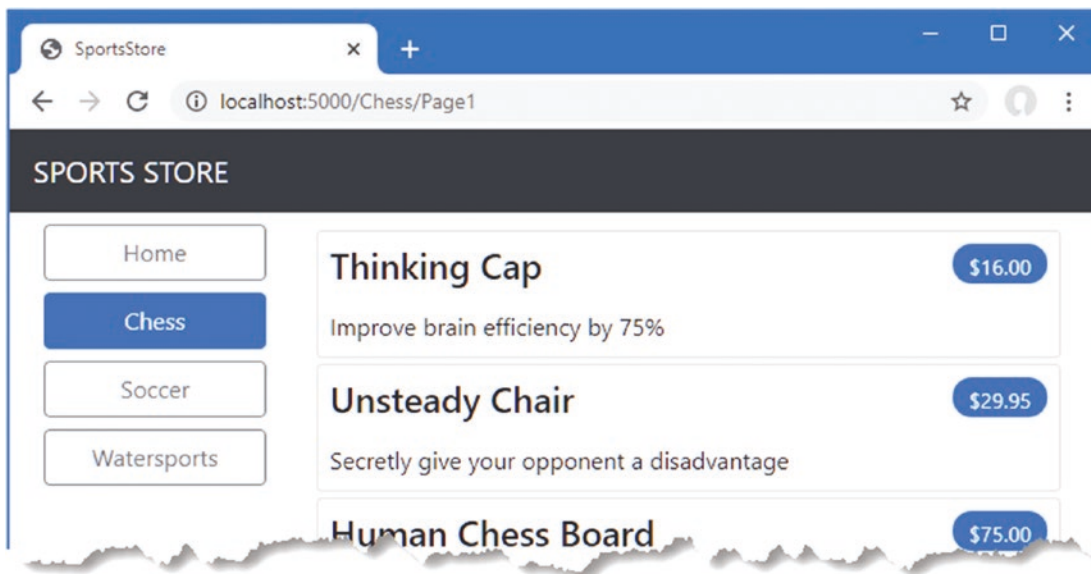


Figure 8-5. Highlighting the selected category

Correcting the Page Count

I need to correct the page links so that they work correctly when a category is selected. Currently, the number of page links is determined by the total number of products in the repository and not the number of products in the selected category. This means that the customer can click the link for page 2 of the Chess category and end up with an empty page because there are not enough chess products to fill two pages. You can see the problem in Figure 8-6.

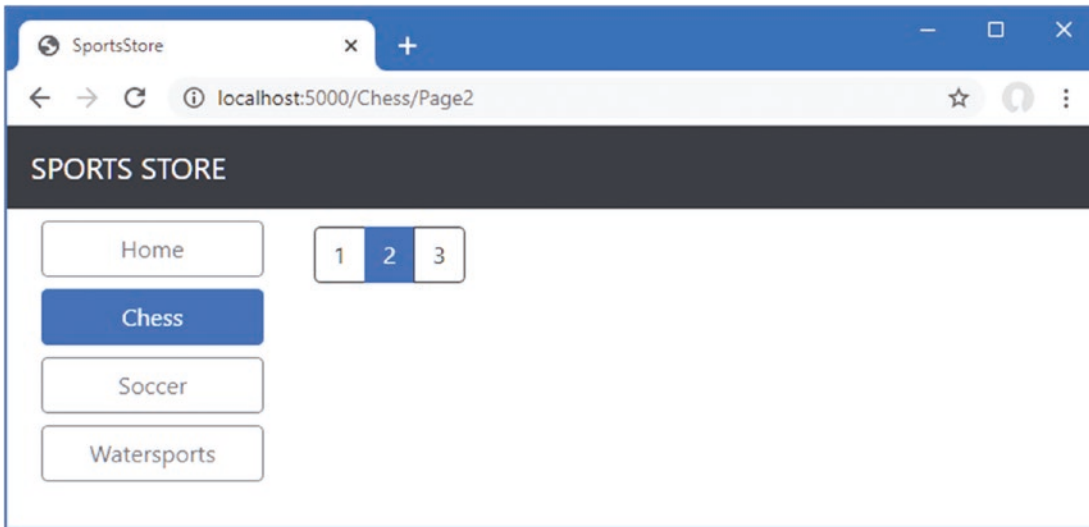


Figure 8-6. Displaying the wrong page links when a category is selected

I can fix this by updating the Index action method in the Home controller so that the pagination information takes the categories into account, as shown in Listing 8-12.

Listing 8-12. Creating Category Pagination Data in the HomeController.cs File in the SportsStore/Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;
using System.Linq;
using SportsStore.Models.ViewModels;

namespace SportsStore.Controllers {
    public class HomeController : Controller {
        private IStoreRepository repository;
        public int PageSize = 4;

        public HomeController(IStoreRepository repo) {
            repository = repo;
        }

        public IActionResult Index(string category, int productPage = 1)
            => View(new ProductsListViewModel {
                Products = repository.Products
                    .Where(p => category == null || p.Category == category)
                    .OrderBy(p => p.ProductID)
                    .Skip((productPage - 1) * PageSize)
                    .Take(PageSize),
                PagingInfo = new PagingInfo {
                    CurrentPage = productPage,
                    ItemsPerPage = PageSize,
                    TotalItems = category == null ?
                        repository.Products.Count() :
                        repository.Products.Where(e =>
                            e.Category == category).Count()
                },
                CurrentCategory = category
            });
    }
}
```

If a category has been selected, I return the number of items in that category; if not, I return the total number of products. Restart ASP.NET Core and request `http://localhost:5000` to see the changes when a category is selected, as shown in Figure 8-7.

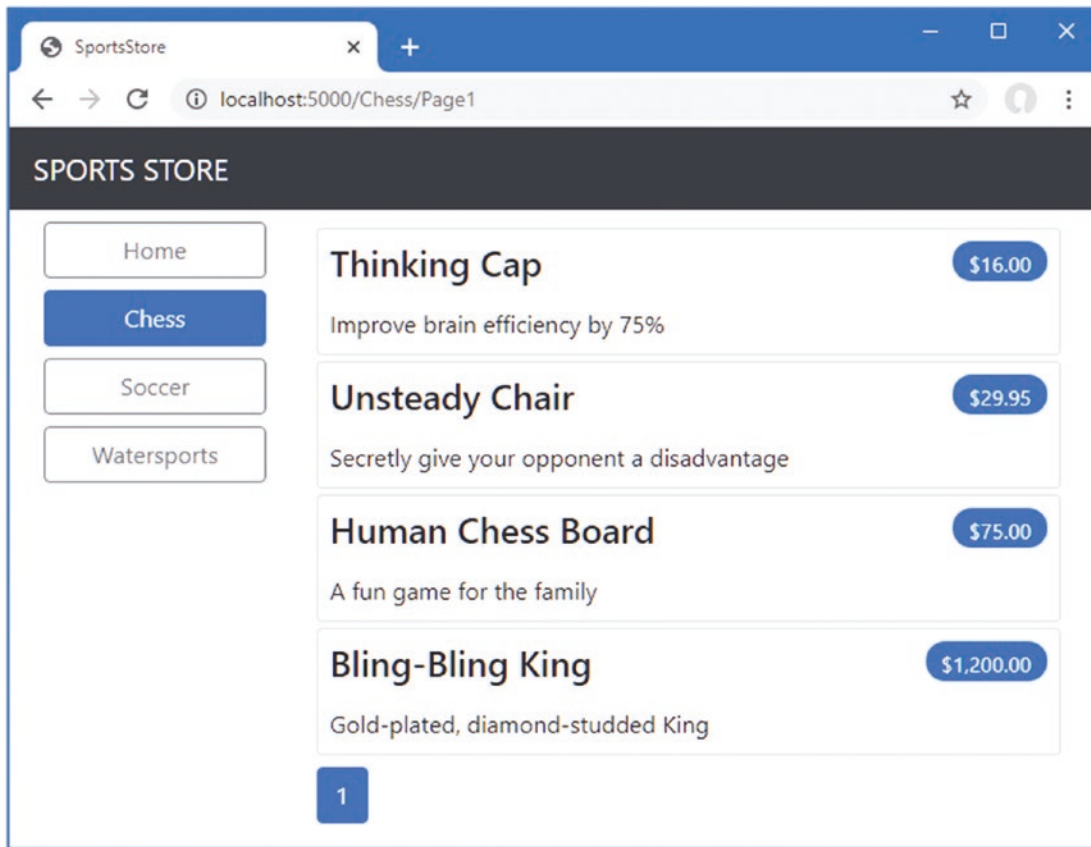


Figure 8-7. *Displaying category-specific page counts*

UNIT TEST: CATEGORY-SPECIFIC PRODUCT COUNTS

Testing that I am able to generate the current product count for different categories is simple. I create a mock repository that contains known data in a range of categories and then call the `List` action method requesting each category in turn. Here is the unit test method that I added to the `HomeControllerTests` class (you will need to import the `System` namespace for this test):

```
...
[Fact]
public void Generate_Category_Specific_Product_Count() {
    // Arrange
    Mock<IStoreRepository> mock = new Mock<IStoreRepository>();
    mock.Setup(m => m.Products).Returns((new Product[] {
        new Product {ProductID = 1, Name = "P1", Category = "Cat1"},
        new Product {ProductID = 2, Name = "P2", Category = "Cat2"},
        new Product {ProductID = 3, Name = "P3", Category = "Cat1"},
        new Product {ProductID = 4, Name = "P4", Category = "Cat2"},
        new Product {ProductID = 5, Name = "P5", Category = "Cat3"}
    })).AsQueryable<Product>());

    HomeController target = new HomeController(mock.Object);
    target.PageSize = 3;
}
```

```

Func<ViewResult, ProductsListViewModel> GetModel = result =>
    result?.ViewData?.Model as ProductsListViewModel;

// Action
int? res1 = GetModel(target.Index("Cat1"))?.PagingInfo.TotalItems;
int? res2 = GetModel(target.Index("Cat2"))?.PagingInfo.TotalItems;
int? res3 = GetModel(target.Index("Cat3"))?.PagingInfo.TotalItems;
int? resAll = GetModel(target.Index(null))?.PagingInfo.TotalItems;

// Assert
Assert.Equal(2, res1);
Assert.Equal(2, res2);
Assert.Equal(1, res3);
Assert.Equal(5, resAll);
}
...

```

Notice that I also call the `Index` method, specifying no category, to make sure I get the correct total count as well.

Building the Shopping Cart

The application is progressing nicely, but I cannot sell any products until I implement a shopping cart. In this section, I will create the shopping cart experience shown in Figure 8-8. This will be familiar to anyone who has ever made a purchase online.



Figure 8-8. The basic shopping cart flow

An Add To Cart button will be displayed alongside each of the products in the catalog. Clicking this button will show a summary of the products the customer has selected so far, including the total cost. At this point, the user can click the Continue Shopping button to return to the product catalog or click the Checkout Now button to complete the order and finish the shopping session.

Configuring Razor Pages

So far, I have used the MVC Framework to define the SportsStore project features. For variety, I am going to use Razor Pages—another application framework supported by ASP.NET Core—to implement the shopping cart. Listing 8-13 configures the `Startup` class to enable Razor Pages in the SportsStore application.

Listing 8-13. Enabling Razor Pages in the `Startup.cs` File in the SportsStore Folder

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

```

```

using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using SportsStore.Models;

namespace SportsStore {
    public class Startup {

        public Startup(IConfiguration config) {
            Configuration = config;
        }

        private IConfiguration Configuration { get; set; }

        public void ConfigureServices(IServiceCollection services) {
            services.AddControllersWithViews();
            services.AddDbContext<StoreDbContext>(opts => {
                opts.UseSqlServer(
                    Configuration["ConnectionStrings:SportsStoreConnection"]);
            });
            services.AddScoped<IStoreRepository, EFStoreRepository>();
            services.AddRazorPages();
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();

            app.UseRouting();
            app.UseEndpoints(endpoints => {
                endpoints.MapControllerRoute("catpage",
                    "{category}/Page{productPage:int}",
                    new { Controller = "Home", action = "Index" });

                endpoints.MapControllerRoute("page", "Page{productPage:int}",
                    new { Controller = "Home", action = "Index", productPage = 1 });

                endpoints.MapControllerRoute("category", "{category}",
                    new { Controller = "Home", action = "Index", productPage = 1 });

                endpoints.MapControllerRoute("pagination",
                    "Products/Page{productPage}",
                    new { Controller = "Home", action = "Index", productPage = 1 });
                endpoints.MapDefaultControllerRoute();
                endpoints.MapRazorPages();
            });

            SeedData.EnsurePopulated(app);
        }
    }
}

```

The `AddRazorPages` method sets up the services used by Razor Pages, and the `MapRazorPages` method registers Razor Pages as endpoints that the URL routing system can use to handle requests.

Add a folder named `Pages`, which is the conventional location for Razor Pages, to the `SportsStore` project. Add a Razor View Imports file named `_ViewImports.cshtml` to the `Pages` folder with the content shown in Listing 8-14. These expressions set the namespace that the Razor Pages will belong to and allow the `SportsStore` classes to be used in Razor Pages without needing to specify their namespace.

Listing 8-14. The Contents of the `_ViewImports.cshtml` File in the `SportsStore/Pages` Folder

```
@namespace SportsStore.Pages
@using Microsoft.AspNetCore.Mvc.RazorPages
@using SportsStore.Models
@using SportsStore.Infrastructure
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

Next, add a Razor View Start file named `_ViewStart.cshtml` to the `Pages` folder, with the content shown in Listing 8-15. Razor Pages have their own configuration files, and this one specifies that the Razor Pages in the `SportsStore` project will use a layout file named `_CartLayout` by default.

Listing 8-15. The Contents of the `_ViewStart.cshtml` File in the `SportsStore/Pages` Folder

```
@{
    Layout = "_CartLayout";
}
```

Finally, to provide the layout the Razor Pages will use, add a Razor View named `_CartLayout.cshtml` to the `Pages` folder with the content shown in Listing 8-16.

Listing 8-16. The Contents of the `_CartLayout.cshtml` File in the `SportsStore/Pages` Folder

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>SportsStore</title>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>
<body>
    <div class="bg-dark text-white p-2">
        <span class="navbar-brand ml-2">SPORTS STORE</span>
    </div>
    <div class="m-1 p-1">
        @RenderBody()
    </div>
</body>
</html>
```

Creating a Razor Page

If you are using Visual Studio, use the Razor Page template item and set the item name to `Cart.cshtml`. This will create a `Cart.cshtml` file and a `Cart.cshtml.cs` class file. Replace the contents of the file with those shown in Listing 8-17. If you are using Visual Studio Code, just create a `Cart.cshtml` file with the content shown in Listing 8-17.

Listing 8-17. The Contents of the `Cart.cshtml` File in the `SportsStore/Pages` Folder

```
@page
<h4>This is the Cart Page</h4>
```

Restart ASP.NET Core and request `http://localhost:5000/cart` to see the placeholder content from Listing 8-17, which is shown in Figure 8-9. Notice that I have not had to register the page and that the mapping between the `/cart` URL path and the Razor Page has been handled automatically.

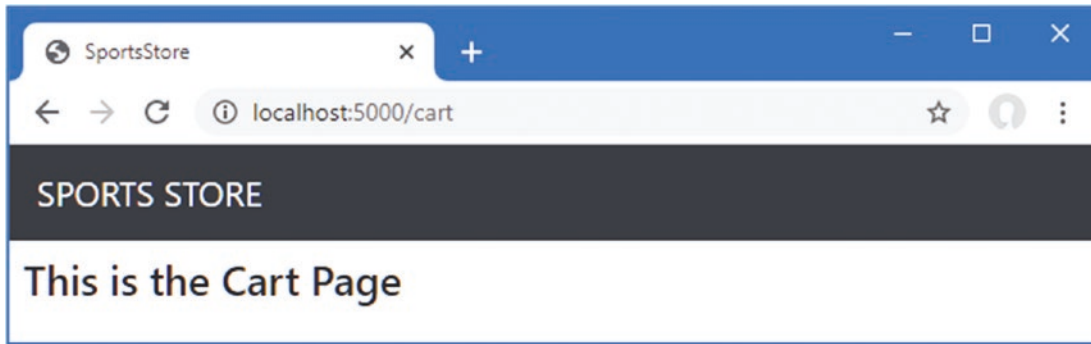


Figure 8-9. Placeholder content from a Razor Page

Creating the Add To Cart Buttons

I have some preparation to do before I can implement the cart feature. First, I need to create the buttons that will add products to the cart. To prepare for this, I added a class file called `UrlExtensions.cs` to the `Infrastructure` folder and defined the extension method shown in Listing 8-18.

Listing 8-18. The Contents of the `UrlExtensions.cs` File in the `SportsStore/Infrastructure` Folder

```
using Microsoft.AspNetCore.Http;

namespace SportsStore.Infrastructure {

    public static class UrlExtensions {

        public static string PathAndQuery(this HttpRequest request) =>
            request.QueryString.HasValue
                ? $"{request.Path}{request.QueryString}"
                : request.Path.ToString();

    }

}
```

The `PathAndQuery` extension method operates on the `HttpRequest` class, which ASP.NET Core uses to describe an HTTP request. The extension method generates a URL that the browser will be returned to after the cart has been updated, taking into account the query string, if there is one. In Listing 8-19, I have added the namespace that contains the extension method to the view imports file so that I can use it in the partial view.

■ **Note** This is the view imports file in the `Views` folder and not the one added to the `Pages` folder.

Listing 8-19. Adding a Namespace in the `_ViewImports.cshtml` File in the `SportsStore/Views` Folder

```
@using SportsStore.Models
@using SportsStore.Models.ViewModels
@using SportsStore.Infrastructure
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper *, SportsStore
```

In Listing 8-20, I have updated the partial view that describes each product so that it contains an Add to Cart button.

Listing 8-20. Adding the Buttons to the ProductSummary.cshtml File View in the SportsStore/Views/Shared Folder

```
@model Product

<div class="card card-outline-primary m-1 p-1">
  <div class="bg-faded p-1">
    <h4>
      @Model.Name
      <span class="badge badge-pill badge-primary" style="float:right">
        <small>@Model.Price.ToString("c")</small>
      </span>
    </h4>
  </div>
  <form id="@Model.ProductID" asp-page="/Cart" method="post">
    <input type="hidden" asp-for="ProductID" />
    <input type="hidden" name="returnUrl"
      value="@ViewContext.HttpContext.Request.PathAndQuery()" />
    <span class="card-text p-1">
      @Model.Description
      <button type="submit"
        class="btn btn-success btn-sm pull-right" style="float:right">
        Add To Cart
      </button>
    </span>
  </form>
</div>
```

I have added a form element that contains hidden input elements specifying the ProductID value from the view model and the URL that the browser should be returned to after the cart has been updated. The form element and one of the input elements are configured using built-in tag helpers, which are a useful way of generating forms that contain model values and that target controllers or Razor Pages, as described in Chapter 27. The other input element uses the extension method I created to set the return URL. I also added a button element that will submit the form to the application.

■ **Note** Notice that I have set the `method` attribute on the form element to `post`, which instructs the browser to submit the form data using an HTTP POST request. You can change this so that forms use the GET method, but you should think carefully about doing so. The HTTP specification requires that GET requests must be *idempotent*, meaning that they must not cause changes, and adding a product to a cart is definitely a change.

Enabling Sessions

I am going to store details of a user's cart using *session state*, which is data associated with a series of requests made by a user. ASP.NET provides a range of different ways to store session state, including storing it in memory, which is the approach that I am going to use. This has the advantage of simplicity, but it means that the session data is lost when the application is stopped or restarted. Enabling sessions requires adding services and middleware in the Startup class, as shown in Listing 8-21.

Listing 8-21. Enabling Sessions in the Startup.cs File in the SportsStore Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
```

```

using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using SportsStore.Models;

namespace SportsStore {
    public class Startup {

        public Startup(IConfiguration config) {
            Configuration = config;
        }

        private IConfiguration Configuration { get; set; }

        public void ConfigureServices(IServiceCollection services) {
            services.AddControllersWithViews();
            services.AddDbContext<StoreDbContext>(opts => {
                opts.UseSqlServer(
                    Configuration["ConnectionStrings:SportsStoreConnection"]);
            });
            services.AddScoped<IStoreRepository, EFStoreRepository>();
            services.AddRazorPages();
            services.AddDistributedMemoryCache();
            services.AddSession();
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseSession();
            app.UseRouting();
            app.UseEndpoints(endpoints => {
                endpoints.MapControllerRoute("catpage",
                    "{category}/Page{productPage:int}",
                    new { Controller = "Home", action = "Index" });

                endpoints.MapControllerRoute("page", "Page{productPage:int}",
                    new { Controller = "Home", action = "Index", productPage = 1 });

                endpoints.MapControllerRoute("category", "{category}",
                    new { Controller = "Home", action = "Index", productPage = 1 });

                endpoints.MapControllerRoute("pagination",
                    "Products/Page{productPage}",
                    new { Controller = "Home", action = "Index", productPage = 1 });
                endpoints.MapDefaultControllerRoute();
                endpoints.MapRazorPages();
            });

            SeedData.EnsurePopulated(app);
        }
    }
}

```

The `AddDistributedMemoryCache` method call sets up the in-memory data store. The `AddSession` method registers the services used to access session data, and the `UseSession` method allows the session system to automatically associate requests with sessions when they arrive from the client.

Implementing the Cart Feature

Now that the preparations are complete, I can implement the cart features. I started by adding a class file called `Cart.cs` to the `Models` folder in the `SportsStore` project and used it to define the classes shown in Listing 8-22.

Listing 8-22. The Contents of the `Cart.cs` File in the `SportsStore/Models` Folder

```
using System.Collections.Generic;
using System.Linq;

namespace SportsStore.Models {

    public class Cart {

        public List<CartLine> Lines { get; set; } = new List<CartLine>();

        public void AddItem(Product product, int quantity) {
            CartLine line = Lines
                .Where(p => p.Product.ProductID == product.ProductID)
                .FirstOrDefault();

            if (line == null) {
                Lines.Add(new CartLine {
                    Product = product,
                    Quantity = quantity
                });
            } else {
                line.Quantity += quantity;
            }
        }

        public void RemoveLine(Product product) =>
            Lines.RemoveAll(l => l.Product.ProductID == product.ProductID);

        public decimal ComputeTotalValue() =>
            Lines.Sum(e => e.Product.Price * e.Quantity);

        public void Clear() => Lines.Clear();
    }

    public class CartLine {
        public int CartLineID { get; set; }
        public Product Product { get; set; }
        public int Quantity { get; set; }
    }
}
```

The `Cart` class uses the `CartLine` class, defined in the same file, to represent a product selected by the customer and the quantity the user wants to buy. I defined methods to add an item to the cart, remove a previously added item from the cart, calculate the total cost of the items in the cart, and reset the cart by removing all the items.

UNIT TEST: TESTING THE CART

The `Cart` class is relatively simple, but it has a range of important behaviors that must work properly. A poorly functioning cart would undermine the entire `SportsStore` application. I have broken down the features and tested them individually. I created a new unit test file called `CartTests.cs` in the `SportsStore.Tests` project to contain these tests.

The first behavior relates to when I add an item to the cart. If this is the first time that a given `Product` has been added to the cart, I want a new `CartLine` to be added. Here is the test, including the unit test class definition:

```
using System.Linq;
using SportsStore.Models;
using Xunit;

namespace SportsStore.Tests {
    public class CartTests {
        [Fact]
        public void Can_Add_New_Lines() {
            // Arrange - create some test products
            Product p1 = new Product { ProductID = 1, Name = "P1" };
            Product p2 = new Product { ProductID = 2, Name = "P2" };

            // Arrange - create a new cart
            Cart target = new Cart();

            // Act
            target.AddItem(p1, 1);
            target.AddItem(p2, 1);
            CartLine[] results = target.Lines.ToArray();

            // Assert
            Assert.Equal(2, results.Length);
            Assert.Equal(p1, results[0].Product);
            Assert.Equal(p2, results[1].Product);
        }
    }
}
```

However, if the customer has already added a `Product` to the cart, I want to increment the quantity of the corresponding `CartLine` and not create a new one. Here is the test:

```
...
[Fact]
public void Can_Add_Quantity_For_Existing_Lines() {
    // Arrange - create some test products
    Product p1 = new Product { ProductID = 1, Name = "P1" };
    Product p2 = new Product { ProductID = 2, Name = "P2" };

    // Arrange - create a new cart
    Cart target = new Cart();

    // Act
    target.AddItem(p1, 1);
    target.AddItem(p2, 1);
    target.AddItem(p1, 10);
    CartLine[] results = target.Lines
        .OrderBy(c => c.Product.ProductID).ToArray();

    // Assert
    Assert.Equal(2, results.Length);
    Assert.Equal(11, results[0].Quantity);
    Assert.Equal(1, results[1].Quantity);
}
...
```

I also need to check that users can change their mind and remove products from the cart. This feature is implemented by the `RemoveLine` method. Here is the test:

```
...
[Fact]
public void Can_Remove_Line() {
    // Arrange - create some test products
    Product p1 = new Product { ProductID = 1, Name = "P1" };
    Product p2 = new Product { ProductID = 2, Name = "P2" };
    Product p3 = new Product { ProductID = 3, Name = "P3" };

    // Arrange - create a new cart
    Cart target = new Cart();
    // Arrange - add some products to the cart
    target.AddItem(p1, 1);
    target.AddItem(p2, 3);
    target.AddItem(p3, 5);
    target.AddItem(p2, 1);

    // Act
    target.RemoveLine(p2);

    // Assert
    Assert.Empty(target.Lines.Where(c => c.Product == p2));
    Assert.Equal(2, target.Lines.Count());
}
...
```

The next behavior I want to test is the ability to calculate the total cost of the items in the cart. Here's the test for this behavior:

```
...
[Fact]
public void Calculate_Cart_Total() {
    // Arrange - create some test products
    Product p1 = new Product { ProductID = 1, Name = "P1", Price = 100M };
    Product p2 = new Product { ProductID = 2, Name = "P2", Price = 50M };

    // Arrange - create a new cart
    Cart target = new Cart();

    // Act
    target.AddItem(p1, 1);
    target.AddItem(p2, 1);
    target.AddItem(p1, 3);
    decimal result = target.ComputeTotalValue();

    // Assert
    Assert.Equal(450M, result);
}
...
```

The final test is simple. I want to ensure that the contents of the cart are properly removed when reset. Here is the test:

```
...
[Fact]
public void Can_Clear_Contents() {
    // Arrange - create some test products
    Product p1 = new Product { ProductID = 1, Name = "P1", Price = 100M };
    Product p2 = new Product { ProductID = 2, Name = "P2", Price = 50M };

    // Arrange - create a new cart
    Cart target = new Cart();
```

```

// Arrange - add some items
target.AddItem(p1, 1);
target.AddItem(p2, 1);

// Act - reset the cart
target.Clear();

// Assert
Assert.Empty(target.Lines);
}
...

```

Sometimes, as in this case, the code required to test the functionality of a class is longer and more complex than the class itself. Do not let that put you off writing the unit tests. Defects in simple classes can have huge impacts, especially ones that play such an important role as `Cart` does in the example application.

Defining Session State Extension Methods

The session state feature in ASP.NET Core stores only `int`, `string`, and `byte[]` values. Since I want to store a `Cart` object, I need to define extension methods to the `ISession` interface, which provides access to the session state data to serialize `Cart` objects into JSON and convert them back. I added a class file called `SessionExtensions.cs` to the `Infrastructure` folder and defined the extension methods shown in Listing 8-23.

Listing 8-23. The Contents of the `SessionExtensions.cs` File in the `SportsStore/Infrastructure` Folder

```

using Microsoft.AspNetCore.Http;
using System.Text.Json;

namespace SportsStore.Infrastructure {

    public static class SessionExtensions {

        public static void SetJson(this ISession session, string key, object value) {
            session.SetString(key, JsonSerializer.Serialize(value));
        }

        public static T GetJson<T>(this ISession session, string key) {
            var sessionData = session.GetString(key);
            return sessionData == null
                ? default(T) : JsonSerializer.Deserialize<T>(sessionData);
        }
    }
}

```

These methods serialize objects into the JavaScript Object Notation format, making it easy to store and retrieve `Cart` objects.

Completing the Razor Page

The `Cart` Razor Page will receive the HTTP POST request that the browser sends when the user clicks an `Add To Cart` button. It will use the request form data to get the `Product` object from the database and use it to update the user's cart, which will be stored as session data for use by future requests. Listing 8-24 implements these features.

Listing 8-24. Handling Requests in the Cart.cshtml File in the SportsStore/Pages Folder

```

@page
@model CartModel

<h2>Your cart</h2>
<table class="table table-bordered table-striped">
  <thead>
    <tr>
      <th>Quantity</th>
      <th>Item</th>
      <th class="text-right">Price</th>
      <th class="text-right">Subtotal</th>
    </tr>
  </thead>
  <tbody>
    @foreach (var line in Model.Cart.Lines) {
      <tr>
        <td class="text-center">@line.Quantity</td>
        <td class="text-left">@line.Product.Name</td>
        <td class="text-right">@line.Product.Price.ToString("c")</td>
        <td class="text-right">
          @((line.Quantity * line.Product.Price).ToString("c"))
        </td>
      </tr>
    }
  </tbody>
  <tfoot>
    <tr>
      <td colspan="3" class="text-right">Total:</td>
      <td class="text-right">
        @Model.Cart.ComputeTotalValue().ToString("c")
      </td>
    </tr>
  </tfoot>
</table>

<div class="text-center">
  <a class="btn btn-primary" href="@Model.ReturnUrl">Continue shopping</a>
</div>

```

Razor Pages allow HTML content, Razor expressions, and code to be combined in a single file, as I explain in Chapter 23, but if you want to unit test a Razor Page, then you need to use a separate class file. If you are using Visual Studio, there will already be a class file named `Cart.cshtml.cs` in the Pages folder, which was created by the Razor Page template item. If you are using Visual Studio Code, you will need to create the class file separately. Use the class file, however it has been created, to define the class shown in Listing 8-25.

Listing 8-25. The Contents of the Cart.cshtml.cs File in the SportsStore/Pages Folder

```

using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using SportsStore.Infrastructure;
using SportsStore.Models;
using System.Linq;

namespace SportsStore.Pages {

```

```

public class CartModel : PageModel {
    private IStoreRepository repository;

    public CartModel(IStoreRepository repo) {
        repository = repo;
    }

    public Cart Cart { get; set; }
    public string returnUrl { get; set; }

    public void OnGet(string returnUrl) {
        returnUrl = returnUrl ?? "/";
        Cart = HttpContext.Session.GetJson<Cart>("cart") ?? new Cart();
    }

    public IActionResult OnPost(long productId, string returnUrl) {
        Product product = repository.Products
            .FirstOrDefault(p => p.ProductID == productId);
        Cart = HttpContext.Session.GetJson<Cart>("cart") ?? new Cart();
        Cart.AddItem(product, 1);
        HttpContext.Session.SetJson("cart", Cart);
        return RedirectToPage(new { returnUrl = returnUrl });
    }
}
}
}

```

The class associated with a Razor Page is known as its page model class, and it defines handler methods that are invoked for different types of HTTP requests, which update state before rendering the view. The page model class in Listing 8-25, which is named `CartModel`, defines an `OnPost` handler method, which is invoked to handle HTTP POST requests. It does this by retrieving a `Product` from the database, retrieving the user's `Cart` from the session data, and updating its content using the `Product`. The modified `Cart` is stored, and the browser is redirected to the same Razor Page, which it will do using a GET request (which prevents reloading the browser from triggering a duplicate POST request).

The GET request is handled by the `OnGet` handler method, which sets the values of the `returnUrl` and `Cart` properties, after which the Razor content section of the page is rendered. The expressions in the HTML content are evaluated using the `CartModel` as the view model object, which means that the values assigned to the `returnUrl` and `Cart` properties can be accessed within the expressions. The content generated by the Razor Page details the products added to the user's cart and provides a button to navigate back to the point where the product was added to the cart.

The handler methods use parameter names that match the input elements in the HTML forms produced by the `ProductSummary.cshtml` view. This allows ASP.NET Core to associate incoming form POST variables with those parameters, meaning I do not need to process the form directly. This is known as *model binding* and is a powerful tool for simplifying development, as I explain in detail in Chapter 28.

UNDERSTANDING RAZOR PAGES

Razor Pages can feel a little odd when you first start using them, especially if you have previous experience with the MVC Framework features provided by ASP.NET Core. But Razor Pages are complementary to the MVC Framework, and I find myself using them alongside controllers and views because they are well-suited to self-contained features that don't require the complexity of the MVC Framework. I describe Razor Pages in Chapter 23 and show their use alongside controllers throughout Part 3 and Part 4 of this book.

The result is that the basic functions of the shopping cart are in place. First, products are listed along with a button to add them to the cart, which you can see by restarting ASP.NET Core and requesting `http://localhost:5000`, as shown in Figure 8-10.

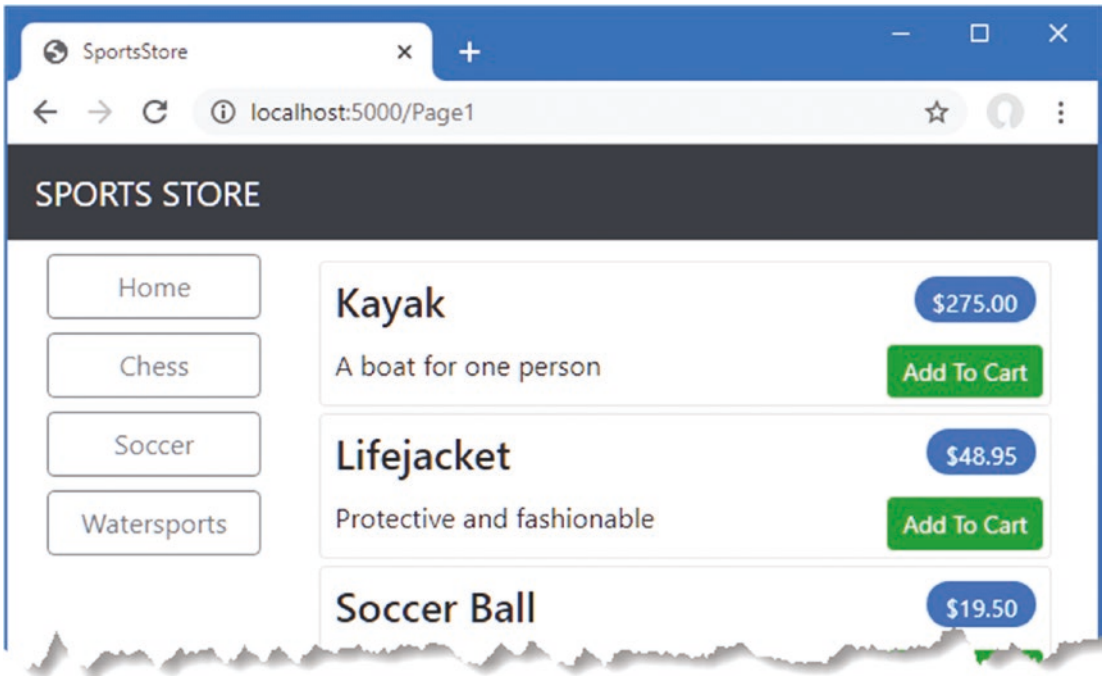


Figure 8-10. The Add To Cart buttons

Second, when the user clicks an Add To Cart button, the appropriate product is added to their cart, and a summary of the cart is displayed, as shown in Figure 8-11. Clicking the Continue Shopping button returns the user to the product page they came from.

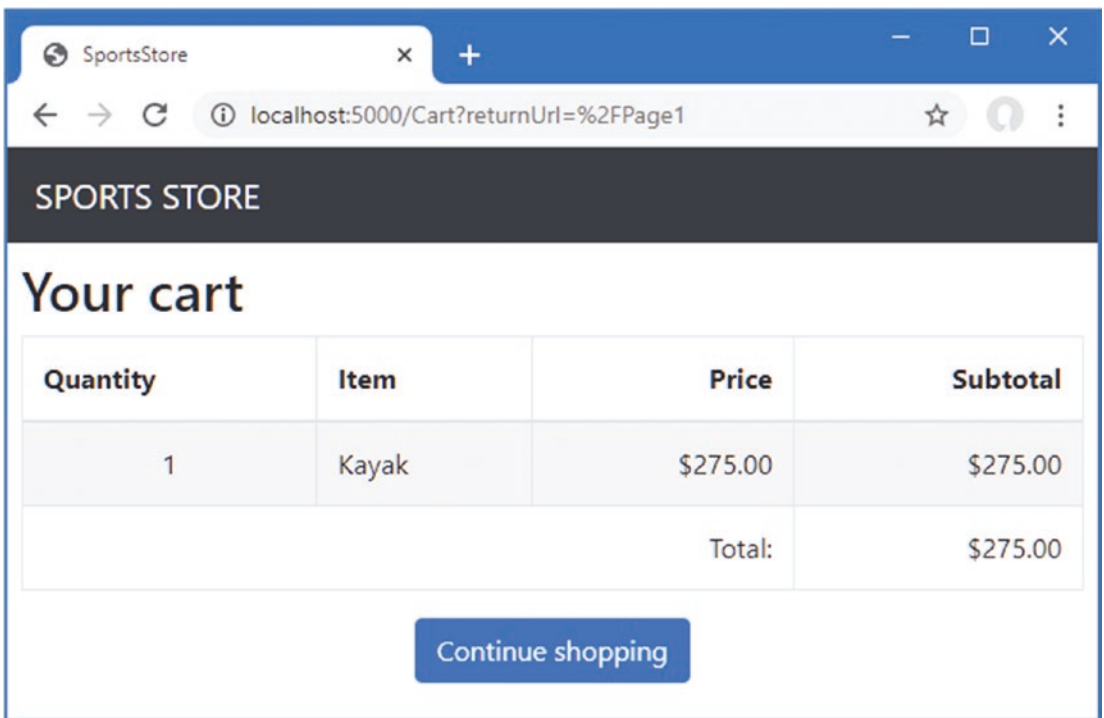


Figure 8-11. Displaying the contents of the shopping cart

UNIT TESTING: RAZOR PAGES

Testing Razor Pages can require a lot of mocking to create the context objects that the page model class requires. To test the behavior of the `OnGet` method defined by the `CartModel` class, I added a class file named `CartPageTests.cs` to the `SportsStore.Tests` project and defined this test:

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using Microsoft.AspNetCore.Routing;
using Moq;
using SportsStore.Models;
using SportsStore.Pages;
using System.Linq;
using System.Text;
using System.Text.Json;
using Xunit;

namespace SportsStore.Tests {
    public class CartPageTests {
        [Fact]
        public void Can_Load_Cart() {
            // Arrange
            // - create a mock repository
            Product p1 = new Product { ProductID = 1, Name = "P1" };
            Product p2 = new Product { ProductID = 2, Name = "P2" };
            Mock<IStoreRepository> mockRepo = new Mock<IStoreRepository>();
            mockRepo.Setup(m => m.Products).Returns((new Product[] {
                p1, p2
            }).AsQueryable<Product>());

            // - create a cart
            Cart testCart = new Cart();
            testCart.AddItem(p1, 2);
            testCart.AddItem(p2, 1);
            // - create a mock page context and session
            Mock<ISession> mockSession = new Mock<ISession>();
            byte[] data =
                Encoding.UTF8.GetBytes(JsonSerializer.Serialize(testCart));
            mockSession.Setup(c => c.TryGetValue(It.IsAny<string>(), out data));
            Mock<HttpContext> mockContext = new Mock<HttpContext>();
            mockContext.SetupGet(c => c.Session).Returns(mockSession.Object);

            // Action
            CartModel cartModel = new CartModel(mockRepo.Object) {
                PageContext = new PageContext(new ActionContext {
                    HttpContext = mockContext.Object,
                    RouteData = new RouteData(),
                    ActionDescriptor = new PageActionDescriptor()
                })
            };
            cartModel.OnGet("myUrl");

            //Assert
            Assert.Equal(2, cartModel.Cart.Lines.Count());
            Assert.Equal("myUrl", cartModel.ReturnUrl);
        }
    }
}
```

I am not going to describe these unit tests in detail because there is a simpler way to perform these tests, which I explain in the next chapter. The complexity in this test is mocking the `ISession` interface so that the page model class can use extension methods to retrieve a JSON representation of a `Cart` object. The `ISession` interface only stored byte arrays, and getting and deserializing a string is performed by extension methods. Once the mock objects are defined, they can be wrapped in context objects and used to configure an instance of the page model class, which can be subjected to tests.

The process of testing the `OnPost` method of the page model class means capturing the byte array that is passed to the `ISession` interface mock and then deserializing it to ensure that it contains the expected content. Here is the unit test I added to the `CartTestsPage` class:

```
...
[Fact]
public void Can_Update_Cart() {
    // Arrange
    // - create a mock repository
    Mock<IStoreRepository> mockRepo = new Mock<IStoreRepository>();
    mockRepo.Setup(m => m.Products).Returns((new Product[] {
        new Product { ProductID = 1, Name = "P1" }
    })).AsQueryable<Product>());

    Cart testCart = new Cart();

    Mock<ISession> mockSession = new Mock<ISession>();
    mockSession.Setup(s => s.Set(It.IsAny<string>(), It.IsAny<byte[]>()))
        .Callback<string, byte[]>((key, val) => {
            testCart =
                JsonSerializer.Deserialize<Cart>(Encoding.UTF8.GetString(val));
        });

    Mock<HttpContext> mockContext = new Mock<HttpContext>();
    mockContext.SetupGet(c => c.Session).Returns(mockSession.Object);

    // Action
    CartModel cartModel = new CartModel(mockRepo.Object) {
        PageContext = new PageContext(new ActionContext {
            HttpContext = mockContext.Object,
            RouteData = new RouteData(),
            ActionDescriptor = new PageActionDescriptor()
        })
    };
    cartModel.OnPost(1, "myUrl");

    //Assert
    Assert.Single(testCart.Lines);
    Assert.Equal("P1", testCart.Lines.First().Product.Name);
    Assert.Equal(1, testCart.Lines.First().Quantity);
}
...
```

Patience and a little experimentation are required to write effective unit tests, especially when the feature you are testing operates on the context objects that ASP.NET Core provides.

Summary

In this chapter, I started to flesh out the customer-facing parts of the `SportsStore` app. I provided the means by which the user can navigate by category and put the basic building blocks in place for adding items to a shopping cart. I have more work to do, and I continue the development of the application in the next chapter.

CHAPTER 9



SportsStore: Completing the Cart

In this chapter, I continue to build the SportsStore example app. In the previous chapter, I added the basic support for a shopping cart, and now I am going to improve on and complete that functionality.

■ **Tip** You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/apress/pro-asp.net-core-3>. See Chapter 1 for how to get help if you have problems running the examples.

Refining the Cart Model with a Service

I defined a Cart model class in the previous chapter and demonstrated how it can be stored using the session feature, allowing the user to build up a set of products for purchase. The responsibility for managing the persistence of the Cart class fell to the Cart Razor Page, which has to deal with getting and storing Cart objects as session data.

The problem with this approach is that I will have to duplicate the code that obtains and stores Cart objects in any other Razor Page or controller that uses them. In this section, I am going to use the services feature that sits at the heart of ASP.NET Core to simplify the way that Cart objects are managed, freeing individual components such as the Cart controller from needing to deal with the details directly.

Services are commonly used to hide details of how interfaces are implemented from the components that depend on them. But services can be used to solve lots of other problems as well and can be used to shape and reshape an application, even when you are working with concrete classes such as Cart.

Creating a Storage-Aware Cart Class

The first step in tidying up the way that the Cart class is used will be to create a subclass that is aware of how to store itself using session state. To prepare, I apply the virtual keyword to the Cart class, as shown in Listing 9-1, so that I can override the members.

Listing 9-1. Applying the Virtual Keyword in the Cart.cs File in the SportsStore/Models Folder

```
using System.Collections.Generic;
using System.Linq;

namespace SportsStore.Models {

    public class Cart {

        public List<CartLine> Lines { get; set; } = new List<CartLine>();

        public virtual void AddItem(Product product, int quantity) {
            CartLine line = Lines
                .Where(p => p.Product.ProductID == product.ProductID)
                .FirstOrDefault();
```

```

        if (line == null) {
            Lines.Add(new CartLine {
                Product = product,
                Quantity = quantity
            });
        } else {
            line.Quantity += quantity;
        }
    }

    public virtual void RemoveLine(Product product) =>
        Lines.RemoveAll(l => l.Product.ProductID == product.ProductID);

    public decimal ComputeTotalValue() =>
        Lines.Sum(e => e.Product.Price * e.Quantity);

    public virtual void Clear() => Lines.Clear();
}

public class CartLine {
    public int CartLineID { get; set; }
    public Product Product { get; set; }
    public int Quantity { get; set; }
}
}

```

Next, I added a class file called `SessionCart.cs` to the `Models` folder and used it to define the class shown in Listing 9-2.

Listing 9-2. The Contents of the `SessionCart.cs` File in the `SportsStore/Models` Folder

```

using System;
using System.Text.Json.Serialization;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using SportsStore.Infrastructure;

namespace SportsStore.Models {

    public class SessionCart : Cart {

        public static Cart GetCart(IServiceProvider services) {
            ISession session = services.GetRequiredService<IHttpContextAccessor>()?
                .HttpContext.Session;
            SessionCart cart = session?.GetJson<SessionCart>("Cart")
                ?? new SessionCart();
            cart.Session = session;
            return cart;
        }

        [JsonIgnore]
        public ISession Session { get; set; }

        public override void AddItem(Product product, int quantity) {
            base.AddItem(product, quantity);
            Session.SetJson("Cart", this);
        }
    }
}

```

```

    public override void RemoveLine(Product product) {
        base.RemoveLine(product);
        Session.SetJson("Cart", this);
    }

    public override void Clear() {
        base.Clear();
        Session.Remove("Cart");
    }
}
}
}

```

The `SessionCart` class subclasses the `Cart` class and overrides the `AddItem`, `RemoveLine`, and `Clear` methods so they call the base implementations and then store the updated state in the session using the extension methods on the `ISession` interface. The static `GetCart` method is a factory for creating `SessionCart` objects and providing them with an `ISession` object so they can store themselves.

Getting hold of the `ISession` object is a little complicated. I obtain an instance of the `IHttpContextAccessor` service, which provides me with access to an `HttpContext` object that, in turn, provides me with the `ISession`. This indirect approach is required because the session isn't provided as a regular service.

Registering the Service

The next step is to create a service for the `Cart` class. My goal is to satisfy requests for `Cart` objects with `SessionCart` objects that will seamlessly store themselves. You can see how I created the service in Listing 9-3.

Listing 9-3. Creating the `Cart` Service in the `Startup.cs` File in the `SportsStore` Folder

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using SportsStore.Models;

namespace SportsStore {
    public class Startup {

        public Startup(IConfiguration config) {
            Configuration = config;
        }

        private IConfiguration Configuration { get; set; }

        public void ConfigureServices(IServiceCollection services) {
            services.AddControllersWithViews();
            services.AddDbContext<StoreDbContext>(opts => {
                opts.UseSqlServer(
                    Configuration["ConnectionStrings:SportsStoreConnection"]);
            });
            services.AddScoped<IStoreRepository, EFStoreRepository>();
            services.AddRazorPages();
        }
    }
}

```

```

        services.AddDistributedMemoryCache();
        services.AddSession();
        services.AddScoped<Cart>(sp => SessionCart.GetCart(sp));
        services.AddSingleton<IHttpContextAccessor, HttpContextAccessor>();
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
        app.UseDeveloperExceptionPage();
        app.UseStatusCodePages();
        app.UseStaticFiles();
        app.UseSession();
        app.UseRouting();
        app.UseEndpoints(endpoints => {
            endpoints.MapControllerRoute("catpage",
                "{category}/Page{productPage:int}",
                new { Controller = "Home", action = "Index" });

            endpoints.MapControllerRoute("page", "Page{productPage:int}",
                new { Controller = "Home", action = "Index", productPage = 1 });

            endpoints.MapControllerRoute("category", "{category}",
                new { Controller = "Home", action = "Index", productPage = 1 });

            endpoints.MapControllerRoute("pagination",
                "Products/Page{productPage}",
                new { Controller = "Home", action = "Index", productPage = 1 });
            endpoints.MapDefaultControllerRoute();
            endpoints.MapRazorPages();
        });

        SeedData.EnsurePopulated(app);
    }
}

```

The `AddScoped` method specifies that the same object should be used to satisfy related requests for `Cart` instances. How requests are related can be configured, but by default, it means that any `Cart` required by components handling the same HTTP request will receive the same object.

Rather than provide the `AddScoped` method with a type mapping, as I did for the repository, I have specified a lambda expression that will be invoked to satisfy `Cart` requests. The expression receives the collection of services that have been registered and passes the collection to the `GetCart` method of the `SessionCart` class. The result is that requests for the `Cart` service will be handled by creating `SessionCart` objects, which will serialize themselves as session data when they are modified.

I also added a service using the `AddSingleton` method, which specifies that the same object should always be used. The service I created tells ASP.NET Core to use the `HttpContextAccessor` class when implementations of the `IHttpContextAccessor` interface are required. This service is required so I can access the current session in the `SessionCart` class.

Simplifying the Cart Razor Page

The benefit of creating this kind of service is that it allows me to simplify the code where `Cart` objects are used. In Listing 9-4, I have reworked the page model class for the `Cart` Razor Page to take advantage of the new service.

Listing 9-4. Using the `Cart` Service in the `Cart.cshtml.cs` File in the `SportsStore/Pages` Folder

```

using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using SportsStore.Infrastructure;
using SportsStore.Models;
using System.Linq;

```

```

namespace SportsStore.Pages {

    public class CartModel : PageModel {
        private IStoreRepository repository;

        public CartModel(IStoreRepository repo, Cart cartService) {
            repository = repo;
            Cart = cartService;
        }

        public Cart Cart { get; set; }
        public string returnUrl { get; set; }

        public void OnGet(string returnUrl) {
            ReturnUrl = returnUrl ?? "/";
        }

        public IActionResult OnPost(long productId, string returnUrl) {
            Product product = repository.Products
                .FirstOrDefault(p => p.ProductID == productId);
            Cart.AddItem(product, 1);
            return RedirectToPage(new { returnUrl = returnUrl });
        }
    }
}

```

The page model class indicates that it needs a `Cart` object by declaring a constructor argument, which has allowed me to remove the statements that load and store sessions from the handler methods. The result is a simpler page model class that focuses on its role in the application without having to worry about how `Cart` objects are created or persisted. And, since services are available throughout the application, any component can get hold of the user's cart using the same technique.

UPDATING THE UNIT TESTS

The simplification of the `CartModel` class in Listing 9-4 requires a corresponding change to the unit tests in the `CartPageTests.cs` file in the unit test project so that the `Cart` is provided as a constructor argument and not accessed through the context objects. Here is the change to the test for reading the cart:

```

...
[Fact]
public void Can_Load_Cart() {
    // Arrange
    // - create a mock repository
    Product p1 = new Product { ProductID = 1, Name = "P1" };
    Product p2 = new Product { ProductID = 2, Name = "P2" };
    Mock<IStoreRepository> mockRepo = new Mock<IStoreRepository>();
    mockRepo.Setup(m => m.Products).Returns((new Product[] {
        p1, p2
    }).AsQueryable<Product>());

    // - create a cart
    Cart testCart = new Cart();
    testCart.AddItem(p1, 2);
    testCart.AddItem(p2, 1);

    // Action
    CartModel cartModel = new CartModel(mockRepo.Object, testCart);
    cartModel.OnGet("myUrl");
}

```



```

    //Assert
    Assert.Equal(2, cartModel.Cart.Lines.Count());
    Assert.Equal("myUrl", cartModel.ReturnUrl);
}
...

```

I applied the same change to the unit test that checks changes to the cart:

```

...
[Fact]
public void Can_Update_Cart() {
    // Arrange
    // - create a mock repository
    Mock<IStoreRepository> mockRepo = new Mock<IStoreRepository>();
    mockRepo.Setup(m => m.Products).Returns((new Product[] {
        new Product { ProductID = 1, Name = "P1" }
    }).AsQueryable<Product>());

    Cart testCart = new Cart();

    // Action
    CartModel cartModel = new CartModel(mockRepo.Object, testCart);
    cartModel.OnPost(1, "myUrl");

    //Assert
    Assert.Single(testCart.Lines);
    Assert.Equal("P1", testCart.Lines.First().Product.Name);
    Assert.Equal(1, testCart.Lines.First().Quantity);
}
...

```

Using services simplifies the testing process and makes it much easier to provide the class being tested with its dependencies.

Completing the Cart Functionality

Now that I have introduced the Cart service, it is time to complete the cart functionality by adding two new features. The first will allow the customer to remove an item from the cart. The second feature will display a summary of the cart at the top of the page.

Removing Items from the Cart

To remove items from the cart, I need to add a Remove button to the content rendered by the Cart Razor Page that will submit an HTTP POST request. The changes are shown in Listing 9-5.

Listing 9-5. Removing Cart Items in the Cart.cshtml File in the SportsStore/Pages Folder

```

@page
@model CartModel

<h2>Your cart</h2>
<table class="table table-bordered table-striped">
  <thead>
    <tr>
      <th>Quantity</th>
      <th>Item</th>
      <th class="text-right">Price</th>
      <th class="text-right">Subtotal</th>
      <th></th>
    </tr>
  </thead>

```

```

<tbody>
  @foreach (var line in Model.Cart.Lines) {
    <tr>
      <td class="text-center">@line.Quantity</td>
      <td class="text-left">@line.Product.Name</td>
      <td class="text-right">@line.Product.Price.ToString("c")</td>
      <td class="text-right">
        @((line.Quantity * line.Product.Price).ToString("c"))
      </td>
      <td class="text-center">
        <form asp-page-handler="Remove" method="post">
          <input type="hidden" name="ProductID"
            value="@line.Product.ProductID" />
          <input type="hidden" name="returnUrl"
            value="@Model.ReturnUrl" />
          <button type="submit" class="btn btn-sm btn-danger">
            Remove
          </button>
        </form>
      </td>
    </tr>
  }
</tbody>
<tfoot>
  <tr>
    <td colspan="3" class="text-right">Total:</td>
    <td class="text-right">
      @Model.Cart.ComputeTotalValue().ToString("c")
    </td>
  </tr>
</tfoot>
</table>

<div class="text-center">
  <a class="btn btn-primary" href="@Model.ReturnUrl">Continue shopping</a>
</div>

```

The button requires a new handler method in the page model class that will receive the request and modify the cart, as shown in Listing 9-6.

Listing 9-6. Removing an Item in the Cart.cshtml.cs File in the SportsStore/Pages Folder

```

using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.RazorPages;
using SportsStore.Infrastructure;
using SportsStore.Models;
using System.Linq;

namespace SportsStore.Pages {

  public class CartModel : PageModel {
    private IStoreRepository repository;

    public CartModel(IStoreRepository repo, Cart cartService) {
      repository = repo;
      Cart = cartService;
    }
  }
}

```

```

public Cart Cart { get; set; }
public string returnUrl { get; set; }

public void OnGet(string returnUrl) {
    returnUrl = returnUrl ?? "/";
}

public IActionResult OnPost(long productId, string returnUrl) {
    Product product = repository.Products
        .FirstOrDefault(p => p.ProductID == productId);
    Cart.AddItem(product, 1);
    return RedirectToPage(new { returnUrl = returnUrl });
}

public IActionResult OnPostRemove(long productId, string returnUrl) {
    Cart.RemoveLine(Cart.Lines.First(cl =>
        cl.Product.ProductID == productId).Product);
    return RedirectToPage(new { returnUrl = returnUrl });
}
}
}

```

The new HTML content defines an HTML form. The handler method that will receive the request is specified with the `asp-page-handler` tag helper attribute, like this:

```

...
<form asp-page-handler="Remove" method="post">
...

```

The specified name is prefixed with `On` and given a suffix that matches the request type so that a value of `Remove` selects the `OnRemovePost` handler method. The handler method uses the value it receives to locate the item in the cart and remove it.

Restart ASP.NET Core and request `http://localhost:5000`. Click the Add To Cart buttons to add items to the cart and then click a Remove button. The cart will be updated to remove the item you specified, as shown in Figure 9-1.

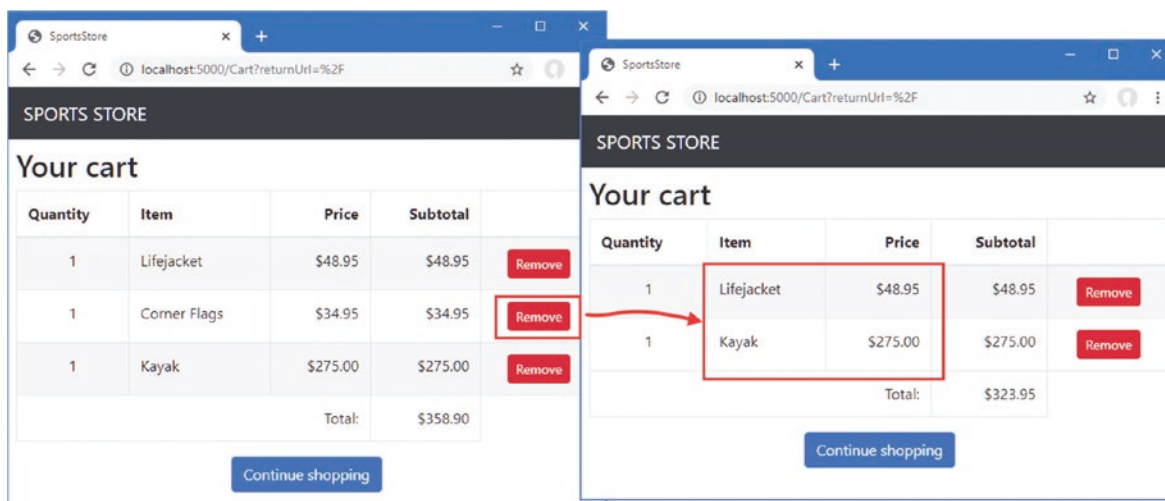


Figure 9-1. Removing items from the shopping cart

Adding the Cart Summary Widget

I may have a functioning cart, but there is an issue with the way it is integrated into the interface. Customers can tell what is in their cart only by viewing the cart summary screen. And they can view the cart summary screen only by adding a new item to the cart.

To solve this problem, I am going to add a widget that summarizes the contents of the cart and that can be clicked to display the cart contents throughout the application. I will do this in much the same way that I added the navigation widget—as a view component whose output I can include in a Razor layout.

Adding the Font Awesome Package

As part of the cart summary, I am going to display a button that allows the user to check out. Rather than display the word *checkout* in the button, I want to use a cart symbol. Since I have no artistic skills, I am going to use the Font Awesome package, which is an excellent set of open source icons that are integrated into applications as fonts, where each character in the font is a different image. You can learn more about Font Awesome, including inspecting the icons it contains, at <http://fontawesome.github.io/Font-Awesome>.

To install the client-side package, use a PowerShell command prompt to run the command shown in Listing 9-7 in the SportsStore project.

Listing 9-7. Installing the Icon Package

```
libman install font-awesome@5.12.0 -d wwwroot/lib/font-awesome
```

Creating the View Component Class and View

I added a class file called `CartSummaryViewComponent.cs` in the Components folder and used it to define the view component shown in Listing 9-8.

Listing 9-8. The Contents of the `CartSummaryViewComponent.cs` File in the `SportsStore/Components` Folder

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;

namespace SportsStore.Components {

    public class CartSummaryViewComponent : ViewComponent {
        private Cart cart;

        public CartSummaryViewComponent(Cart cartService) {
            cart = cartService;
        }

        public IViewComponentResult Invoke() {
            return View(cart);
        }
    }
}
```

This view component is able to take advantage of the service that I created earlier in the chapter to receive a `Cart` object as a constructor argument. The result is a simple view component class that passes on the `Cart` to the `View` method to generate the fragment of HTML that will be included in the layout. To create the view for the component, I created the `Views/Shared/Components/CartSummary` folder and added to it a Razor View named `Default.cshtml` with the content shown in Listing 9-9.

Listing 9-9. The `Default.cshtml` File in the `Views/Shared/Components/CartSummary` Folder

```
@model Cart

<div class="">
    @if (Model.Lines.Count() > 0) {
        <small class="navbar-text">
            <b>Your cart:</b>
        </small>
    }
}
```

```

        @Model.Lines.Sum(x => x.Quantity) item(s)
        @Model.ComputeTotalValue().ToString("c")
    </small>
}
<a class="btn btn-sm btn-secondary navbar-btn" asp-page="/Cart"
    asp-route-returnurl="@ViewContext.HttpContext.Request.PathAndQuery()">
    <i class="fa fa-shopping-cart"></i>
</a>
</div>

```

The view displays a button with the Font Awesome cart icon and, if there are items in the cart, provides a snapshot that details the number of items and their total value. Now that I have a view component and a view, I can modify the layout so that the cart summary is included in the responses generated by the Home controller, as shown in Listing 9-10.

Listing 9-10. Adding the Cart Summary in the `_Layout.cshtml` File in the Views/Shared Folder

```

<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>SportsStore</title>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
    <link href="/lib/font-awesome/css/all.min.css" rel="stylesheet" />
</head>
<body>
    <div class="bg-dark text-white p-2">
        <div class="container-fluid">
            <div class="row">
                <div class="col navbar-brand">SPORTS STORE</div>
                <div class="col-6 text-right">
                    <vc:cart-summary />
                </div>
            </div>
        </div>
    </div>
    <div class="row m-1 p-1">
        <div id="categories" class="col-3">
            <vc:navigation-menu />
        </div>
        <div class="col-9">
            @RenderBody()
        </div>
    </div>
</body>
</html>

```

You can see the cart summary by starting the application. When the cart is empty, only the checkout button is shown. If you add items to the cart, then the number of items and their combined cost are shown, as illustrated in Figure 9-2. With this addition, customers know what is in their cart and have an obvious way to check out from the store.

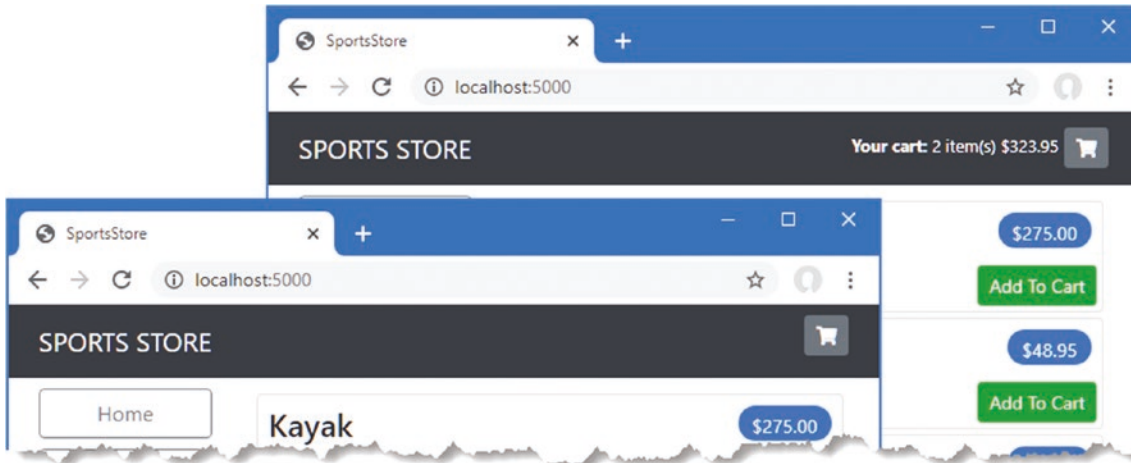


Figure 9-2. Displaying a summary of the cart

Submitting Orders

I have now reached the final customer feature in SportsStore: the ability to check out and complete an order. In the following sections, I will extend the data model to provide support for capturing the shipping details from a user and add the application support to process those details.

Creating the Model Class

I added a class file called `Order.cs` to the `Models` folder and used it to define the class shown in Listing 9-11. This is the class I will use to represent the shipping details for a customer.

Listing 9-11. The Contents of the `Order.cs` File in the `SportsStore/Models` Folder

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using Microsoft.AspNetCore.Mvc.ModelBinding;

namespace SportsStore.Models {

    public class Order {

        [BindNever]
        public int OrderID { get; set; }
        [BindNever]
        public ICollection<CartLine> Lines { get; set; }

        [Required(ErrorMessage = "Please enter a name")]
        public string Name { get; set; }

        [Required(ErrorMessage = "Please enter the first address line")]
        public string Line1 { get; set; }
        public string Line2 { get; set; }
        public string Line3 { get; set; }

        [Required(ErrorMessage = "Please enter a city name")]
        public string City { get; set; }
    }
}
```

```

    [Required(ErrorMessage = "Please enter a state name")]
    public string State { get; set; }

    public string Zip { get; set; }

    [Required(ErrorMessage = "Please enter a country name")]
    public string Country { get; set; }

    public bool GiftWrap { get; set; }
}
}

```

I am using the validation attributes from the `System.ComponentModel.DataAnnotations` namespace, just as I did in Chapter 3. I describe validation further in Chapter 29.

I also use the `BindNever` attribute, which prevents the user from supplying values for these properties in an HTTP request. This is a feature of the model binding system, which I describe in Chapter 28, and it stops ASP.NET Core using values from the HTTP request to populate sensitive or important model properties.

Adding the Checkout Process

The goal is to reach the point where users are able to enter their shipping details and submit an order. To start, I need to add a Checkout button to the cart view, as shown in Listing 9-12.

Listing 9-12. Adding a Button in the `Cart.cshtml` File in the `SportsStore/Pages` Folder

```

...
<div class="text-center">
  <a class="btn btn-primary" href="@Model.ReturnUrl">Continue shopping</a>
  <a class="btn btn-primary" asp-action="Checkout" asp-controller="Order">
    Checkout
  </a>
</div>
...

```

This change generates a link that I have styled as a button and that, when clicked, calls the `Checkout` action method of the `Order` controller, which I create in the following section. To show how Razor Pages and controllers can work together, I am going to handle the order processing in a controller and then return to a Razor Page at the end of the process. To see the Checkout button, restart ASP.NET Core, request `http://localhost:5000`, and click one of the Add To Cart buttons. The new button is shown as part of the cart summary, as shown in Figure 9-3.

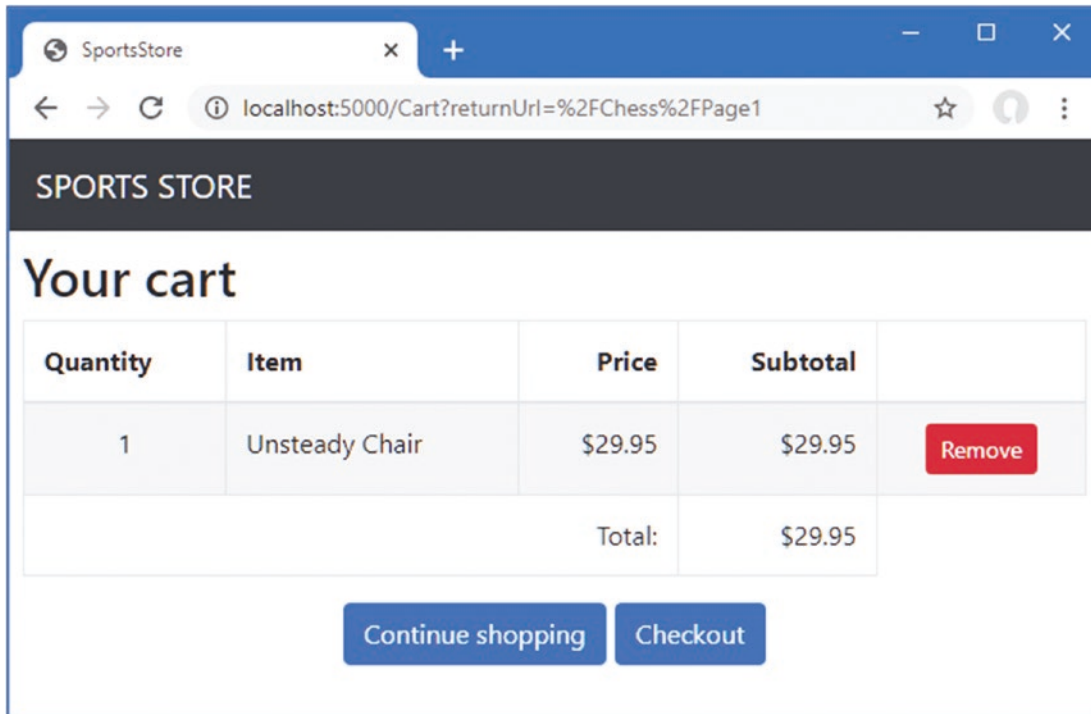


Figure 9-3. The Checkout button

Creating the Controller and View

I now need to define the controller that will deal with the order. I added a class file called `OrderController.cs` to the `Controllers` folder and used it to define the class shown in Listing 9-13.

Listing 9-13. The Contents of the `OrderController.cs` File in the `SportsStore/Controllers` Folder

```
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;

namespace SportsStore.Controllers {
    public class OrderController : Controller {
        public IActionResult Checkout() => View(new Order());
    }
}
```

The `Checkout` method returns the default view and passes a new `Order` object as the view model. To create the view, I created the `Views/Order` folder and added to it a Razor View called `Checkout.cshtml` with the markup shown in Listing 9-14.

Listing 9-14. The Contents of the `Checkout.cshtml` File in the `SportsStore/Views/Order` Folder

```
@model Order

<h2>Check out now</h2>
<p>Please enter your details, and we'll ship your goods right away!</p>

<form asp-action="Checkout" method="post">
    <h3>Ship to</h3>
```



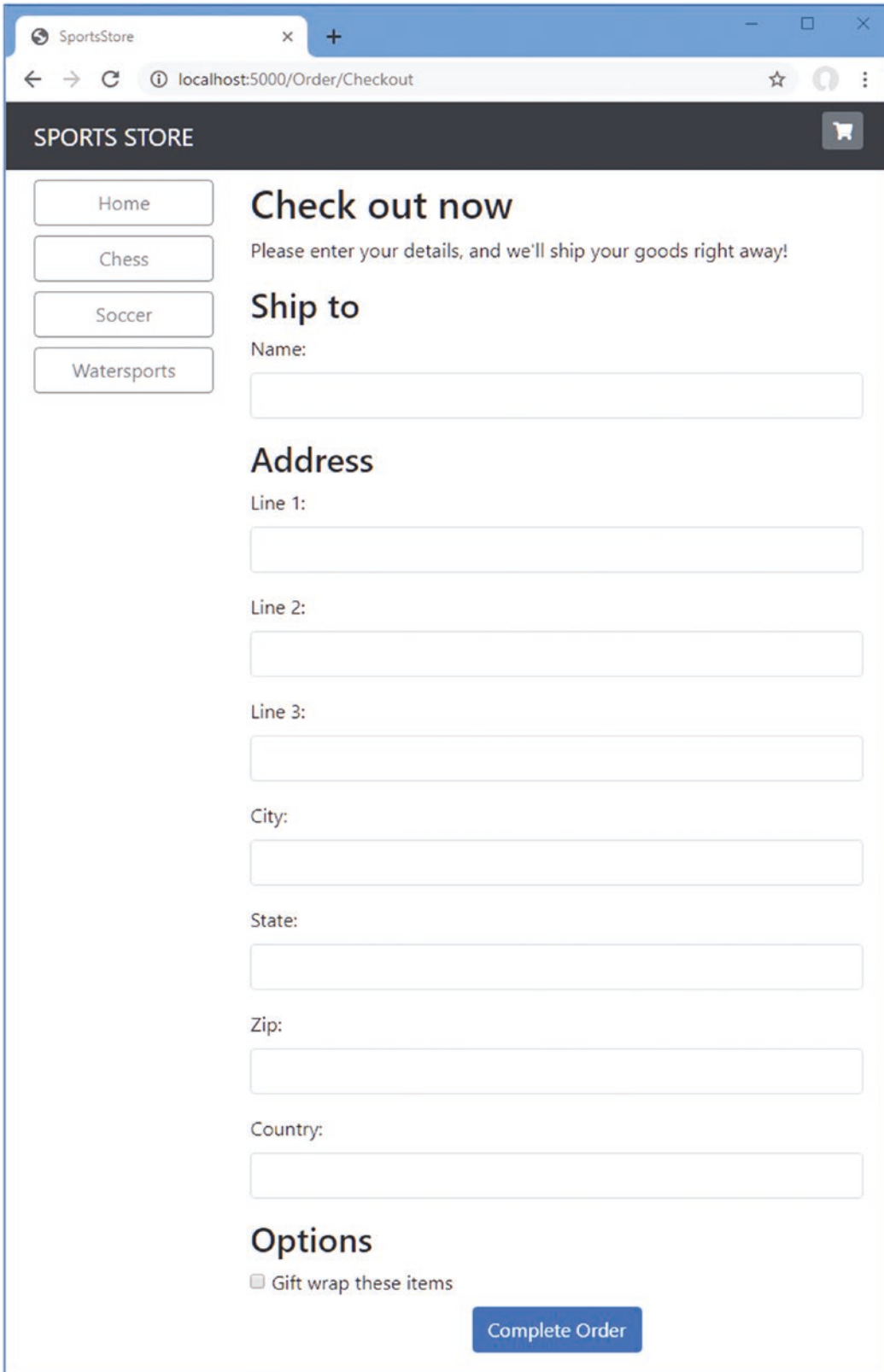
```

<div class="form-group">
  <label>Name:</label><input asp-for="Name" class="form-control" />
</div>
<h3>Address</h3>
<div class="form-group">
  <label>Line 1:</label><input asp-for="Line1" class="form-control" />
</div>
<div class="form-group">
  <label>Line 2:</label><input asp-for="Line2" class="form-control" />
</div>
<div class="form-group">
  <label>Line 3:</label><input asp-for="Line3" class="form-control" />
</div>
<div class="form-group">
  <label>City:</label><input asp-for="City" class="form-control" />
</div>
<div class="form-group">
  <label>State:</label><input asp-for="State" class="form-control" />
</div>
<div class="form-group">
  <label>Zip:</label><input asp-for="Zip" class="form-control" />
</div>
<div class="form-group">
  <label>Country:</label><input asp-for="Country" class="form-control" />
</div>
<h3>Options</h3>
<div class="checkbox">
  <label>
    <input asp-for="GiftWrap" /> Gift wrap these items
  </label>
</div>
<div class="text-center">
  <input class="btn btn-primary" type="submit" value="Complete Order" />
</div>
</form>

```

For each of the properties in the model, I have created a label and input elements to capture the user input, styled with Bootstrap and configured using a tag helper. The `asp-for` attribute on the input elements is handled by a built-in tag helper that generates the `type`, `id`, `name`, and `value` attributes based on the specified model property, as described in Chapter 27.

You can see the form, shown in Figure 9-4, by restarting ASP.NET Core, requesting `http://localhost:5000`, adding an item to the basket, and clicking the Checkout button. Or, more directly, you can request `http://localhost:5000/order/checkout`.



The screenshot shows a web browser window with the URL `localhost:5000/Order/Checkout`. The page header includes the text "SPORTS STORE" and a shopping cart icon. On the left side, there is a vertical navigation menu with buttons for "Home", "Chess", "Soccer", and "Watersports". The main content area is titled "Check out now" and contains the instruction "Please enter your details, and we'll ship your goods right away!". Below this is a "Ship to" section with a "Name:" label and an input field. The "Address" section follows, with labels for "Line 1:", "Line 2:", "Line 3:", "City:", "State:", "Zip:", and "Country:", each accompanied by an input field. At the bottom of the form is an "Options" section with a checkbox labeled "Gift wrap these items". A blue "Complete Order" button is positioned at the bottom right of the form area.

SportsStore

localhost:5000/Order/Checkout

SPORTS STORE

Home

Chess

Soccer

Watersports

Check out now

Please enter your details, and we'll ship your goods right away!

Ship to

Name:

Address

Line 1:

Line 2:

Line 3:

City:

State:

Zip:

Country:

Options

Gift wrap these items

Complete Order

Figure 9-4. The shipping details form

Implementing Order Processing

I will process orders by writing them to the database. Most e-commerce sites would not simply stop there, of course, and I have not provided support for processing credit cards or other forms of payment. But I want to keep things focused on ASP.NET Core, so a simple database entry will do.

Extending the Database

Adding a new kind of model to the database is simple because of the initial setup I went through in Chapter 7. First, I added a new property to the database context class, as shown in Listing 9-15.

Listing 9-15. Adding a Property in the StoreDbContext.cs File in the SportsStore/Models Folder

```
using Microsoft.EntityFrameworkCore;

namespace SportsStore.Models {
    public class StoreDbContext: DbContext {

        public StoreDbContext(DbContextOptions<StoreDbContext> options)
            : base(options) { }

        public DbSet<Product> Products { get; set; }
        public DbSet<Order> Orders { get; set; }
    }
}
```

This change is enough for Entity Framework Core to create a database migration that will allow Order objects to be stored in the database. To create the migration, use a PowerShell command prompt to run the command shown in Listing 9-16 in the SportsStore folder.

Listing 9-16. Creating a Migration

```
dotnet ef migrations add Orders
```

This command tells Entity Framework Core to take a new snapshot of the application data model, work out how it differs from the previous database version, and generate a new migration called Orders. The new migration will be applied automatically when the application starts because the SeedData calls the Migrate method provided by Entity Framework Core.

RESETTING THE DATABASE

When you are making frequent changes to the model, there will come a point when your migrations and your database schema get out of sync. The easiest thing to do is delete the database and start over. However, this applies only during development, of course, because you will lose any data you have stored. Run this command to delete the database:

```
dotnet ef database drop --force --context StoreDbContext
```

Once the database has been removed, run the following command from the SportsStore folder to re-create the database and apply the migrations you have created by running the following command:

```
dotnet ef database update --context StoreDbContext
```

The migrations will also be applied by the SeedData class if you just start the application. Either way, the database will be reset so that it accurately reflects your data model and allows you to return to developing your application.

Creating the Order Repository

I am going to follow the same pattern I used for the product repository to provide access to the `Order` objects. I added a class file called `IOrderRepository.cs` to the `Models` folder and used it to define the interface shown in Listing 9-17.

Listing 9-17. The Contents of the `IOrderRepository.cs` File in the `SportsStore/Models` Folder

```
using System.Linq;

namespace SportsStore.Models {

    public interface IOrderRepository {

        IQueryable<Order> Orders { get; }
        void SaveOrder(Order order);
    }
}
```

To implement the order repository interface, I added a class file called `EFOOrderRepository.cs` to the `Models` folder and defined the class shown in Listing 9-18.

Listing 9-18. The Contents of the `EFOOrderRepository.cs` File in the `SportsStore/Models` Folder

```
using Microsoft.EntityFrameworkCore;
using System.Linq;

namespace SportsStore.Models {

    public class EFOOrderRepository : IOrderRepository {
        private StoreDbContext context;

        public EFOOrderRepository(StoreDbContext ctx) {
            context = ctx;
        }

        public IQueryable<Order> Orders => context.Orders
            .Include(o => o.Lines)
            .ThenInclude(l => l.Product);

        public void SaveOrder(Order order) {
            context.AttachRange(order.Lines.Select(l => l.Product));
            if (order.OrderID == 0) {
                context.Orders.Add(order);
            }
            context.SaveChanges();
        }
    }
}
```

This class implements the `IOrderRepository` interface using Entity Framework Core, allowing the set of `Order` objects that have been stored to be retrieved and allowing for orders to be created or changed.

UNDERSTANDING THE ORDER REPOSITORY

Entity Framework Core requires instruction to load related data if it spans multiple tables. In Listing 9-18, I used the `Include` and `ThenInclude` methods to specify that when an `Order` object is read from the database, the collection associated with the `Lines` property should also be loaded along with each `Product` object associated with each collection object.

```

...
public IQueryable<Order> Orders => context.Orders
    .Include(o => o.Lines)
    .ThenInclude(l => l.Product);
...

```

This ensures that I receive all the data objects that I need without having to perform separate queries and then assemble the data myself.

An additional step is also required when I store an `Order` object in the database. When the user's cart data is de-serialized from the session store, new objects are created that are not known to Entity Framework Core, which then tries to write all the objects into the database. For the `Product` objects associated with an `Order`, this means that Entity Framework Core tries to write objects that have already been stored, which causes an error. To avoid this problem, I notify Entity Framework Core that the objects exist and shouldn't be stored in the database unless they are modified, as follows:

```

...
context.AttachRange(order.Lines.Select(l => l.Product));
...

```

This ensures that Entity Framework Core won't try to write the de-serialized `Product` objects that are associated with the `Order` object.

In Listing 9-19, I have registered the order repository as a service in the `ConfigureServices` method of the `Startup` class.

Listing 9-19. Registering the Order Repository Service in the `Startup.cs` File in the `SportsStore` Folder

```

...
public void ConfigureServices(IServiceCollection services) {
    services.AddControllersWithViews();
    services.AddDbContext<StoreDbContext>(opts => {
        opts.UseSqlServer(
            Configuration["ConnectionStrings:SportsStoreConnection"]);
    });
    services.AddScoped<IStoreRepository, EFStoreRepository>();
    services.AddScoped<IOrderRepository, EFOrderRepository>();
    services.AddRazorPages();
    services.AddDistributedMemoryCache();
    services.AddSession();
    services.AddScoped<Cart>(sp => SessionCart.GetCart(sp));
    services.AddSingleton<IHttpContextAccessor, HttpContextAccessor>();
}
...

```

Completing the Order Controller

To complete the `OrderController` class, I need to modify the constructor so that it receives the services it requires to process an order and add an action method that will handle the HTTP form POST request when the user clicks the Complete Order button. Listing 9-20 shows both changes.

Listing 9-20. Completing the Controller in the `OrderController.cs` File in the `SportsStore/Controllers` Folder

```

using Microsoft.AspNetCore.Mvc;
using SportsStore.Models;
using System.Linq;

namespace SportsStore.Controllers {

```

```

public class OrderController : Controller {
    private IOrderRepository repository;
    private Cart cart;

    public OrderController(IOrderRepository repoService, Cart cartService) {
        repository = repoService;
        cart = cartService;
    }

    public IActionResult Checkout() => View(new Order());

    [HttpPost]
    public IActionResult Checkout(Order order) {
        if (cart.Lines.Count() == 0) {
            ModelState.AddModelError("", "Sorry, your cart is empty!");
        }
        if (ModelState.IsValid) {
            order.Lines = cart.Lines.ToArray();
            repository.SaveOrder(order);
            cart.Clear();
            return RedirectToPage("/Completed", new { orderId = order.OrderID });
        } else {
            return View();
        }
    }
}
}

```

The Checkout action method is decorated with the `HttpPost` attribute, which means that it will be used to handle POST requests—in this case, when the user submits the form.

In Chapter 8, I use the ASP.NET Core model binding feature to receive simple data values from the request. This same feature is used in the new action method to receive a completed `Order` object. When a request is processed, the model binding system tries to find values for the properties defined by the `Order` class. This works on a best-effort basis, which means I may receive an `Order` object lacking property values if there is no corresponding data item in the request.

To ensure I have the data I require, I applied validation attributes to the `Order` class. ASP.NET Core checks the validation constraints that I applied to the `Order` class and provides details of the result through the `ModelState` property. I can see whether there are any problems by checking the `ModelState.IsValid` property. I call the `ModelState.AddModelError` method to register an error message if there are no items in the cart. I will explain how to display such errors shortly, and I have much more to say about model binding and validation in Chapters 28 and 29.

UNIT TEST: ORDER PROCESSING

To perform unit testing for the `OrderController` class, I need to test the behavior of the POST version of the Checkout method. Although the method looks short and simple, the use of model binding means that there is a lot going on behind the scenes that needs to be tested.

I want to process an order only if there are items in the cart *and* the customer has provided valid shipping details. Under all other circumstances, the customer should be shown an error. Here is the first test method, which I defined in a class file called `OrderControllerTests.cs` in the `SportsStore.Tests` project:

```

using Microsoft.AspNetCore.Mvc;
using Moq;
using SportsStore.Controllers;
using SportsStore.Models;
using Xunit;

namespace SportsStore.Tests {

```

```

public class OrderControllerTests {
    [Fact]
    public void Cannot_Checkout_Empty_Cart() {
        // Arrange - create a mock repository
        Mock<IOrderRepository> mock = new Mock<IOrderRepository>();
        // Arrange - create an empty cart
        Cart cart = new Cart();
        // Arrange - create the order
        Order order = new Order();
        // Arrange - create an instance of the controller
        OrderController target = new OrderController(mock.Object, cart);

        // Act
        ViewResult result = target.Checkout(order) as ViewResult;

        // Assert - check that the order hasn't been stored
        mock.Verify(m => m.SaveOrder(It.IsAny<Order>()), Times.Never);
        // Assert - check that the method is returning the default view
        Assert.True(string.IsNullOrEmpty(result.ViewName));
        // Assert - check that I am passing an invalid model to the view
        Assert.False(result.ViewData.ModelState.IsValid);
    }
}

```

This test ensures that I cannot check out with an empty cart. I check this by ensuring that the `SaveOrder` of the mock `IOrderRepository` implementation is never called, that the view the method returns is the default view (which will redisplay the data entered by customers and give them a chance to correct it), and that the model state being passed to the view has been marked as invalid. This may seem like a belt-and-braces set of assertions, but I need all three to be sure that I have the right behavior. The next test method works in much the same way but injects an error into the view model to simulate a problem reported by the model binder (which would happen in production when the customer enters invalid shipping data):

```

...
[Fact]
public void Cannot_Checkout_Invalid_ShippingDetails() {
    // Arrange - create a mock order repository
    Mock<IOrderRepository> mock = new Mock<IOrderRepository>();
    // Arrange - create a cart with one item
    Cart cart = new Cart();
    cart.AddItem(new Product(), 1);
    // Arrange - create an instance of the controller
    OrderController target = new OrderController(mock.Object, cart);
    // Arrange - add an error to the model
    target.ModelState.AddModelError("error", "error");

    // Act - try to checkout
    ViewResult result = target.Checkout(new Order()) as ViewResult;

    // Assert - check that the order hasn't been passed stored
    mock.Verify(m => m.SaveOrder(It.IsAny<Order>()), Times.Never);
    // Assert - check that the method is returning the default view
    Assert.True(string.IsNullOrEmpty(result.ViewName));
    // Assert - check that I am passing an invalid model to the view
    Assert.False(result.ViewData.ModelState.IsValid);
}
...

```

Having established that an empty cart or invalid details will prevent an order from being processed, I need to ensure that I process orders when appropriate. Here is the test:

```
...
[Fact]
public void Can_Checkout_And_Submit_Order() {
    // Arrange - create a mock order repository
    Mock<IOrderRepository> mock = new Mock<IOrderRepository>();
    // Arrange - create a cart with one item
    Cart cart = new Cart();
    cart.AddItem(new Product(), 1);
    // Arrange - create an instance of the controller
    OrderController target = new OrderController(mock.Object, cart);

    // Act - try to checkout
    RedirectToPageResult result =
        target.Checkout(new Order()) as RedirectToPageResult;

    // Assert - check that the order has been stored
    mock.Verify(m => m.SaveOrder(It.IsAny<Order>()), Times.Once);
    // Assert - check that the method is redirecting to the Completed action
    Assert.Equal("/Completed", result.PageName);
}
...
```

I did not need to test that I can identify valid shipping details. This is handled for me automatically by the model binder using the attributes applied to the properties of the `Order` class.

Displaying Validation Errors

ASP.NET Core uses the validation attributes applied to the `Order` class to validate user data, but I need to make a simple change to display any problems. This relies on another built-in tag helper that inspects the validation state of the data provided by the user and adds warning messages for each problem that has been discovered. Listing 9-21 shows the addition of an HTML element that will be processed by the tag helper to the `Checkout.cshtml` file.

Listing 9-21. Adding a Validation Summary to the `Checkout.cshtml` File in the `SportsStore/Views/Order` Folder

```
@model Order
```

```
<h2>Check out now</h2>
```

```
<p>Please enter your details, and we'll ship your goods right away!</p>
```

```
<div asp-validation-summary="All" class="text-danger"></div>
```

```
<form asp-action="Checkout" method="post">
```

```
  <h3>Ship to</h3>
```

```
  <div class="form-group">
```

```
    <label>Name:</label><input asp-for="Name" class="form-control" />
```

```
  </div>
```

```
  <h3>Address</h3>
```

```
  <div class="form-group">
```

```
    <label>Line 1:</label><input asp-for="Line1" class="form-control" />
```

```
  </div>
```

```
  <div class="form-group">
```

```
    <label>Line 2:</label><input asp-for="Line2" class="form-control" />
```

```
  </div>
```

```
  <div class="form-group">
```

```
    <label>Line 3:</label><input asp-for="Line3" class="form-control" />
```

```
  </div>
```



```

<div class="form-group">
  <label>City:</label><input asp-for="City" class="form-control" />
</div>
<div class="form-group">
  <label>State:</label><input asp-for="State" class="form-control" />
</div>
<div class="form-group">
  <label>Zip:</label><input asp-for="Zip" class="form-control" />
</div>
<div class="form-group">
  <label>Country:</label><input asp-for="Country" class="form-control" />
</div>
<h3>Options</h3>
<div class="checkbox">
  <label>
    <input asp-for="GiftWrap" /> Gift wrap these items
  </label>
</div>
<div class="text-center">
  <input class="btn btn-primary" type="submit" value="Complete Order" />
</div>
</form>

```

With this simple change, validation errors are reported to the user. To see the effect, restart ASP.NET Core, request `http://localhost:5000/Order/Checkout`, and click the Complete Order button without filling out the form. ASP.NET Core will process the form data, detect that the required values were not found, and generate the validation errors shown in Figure 9-5.

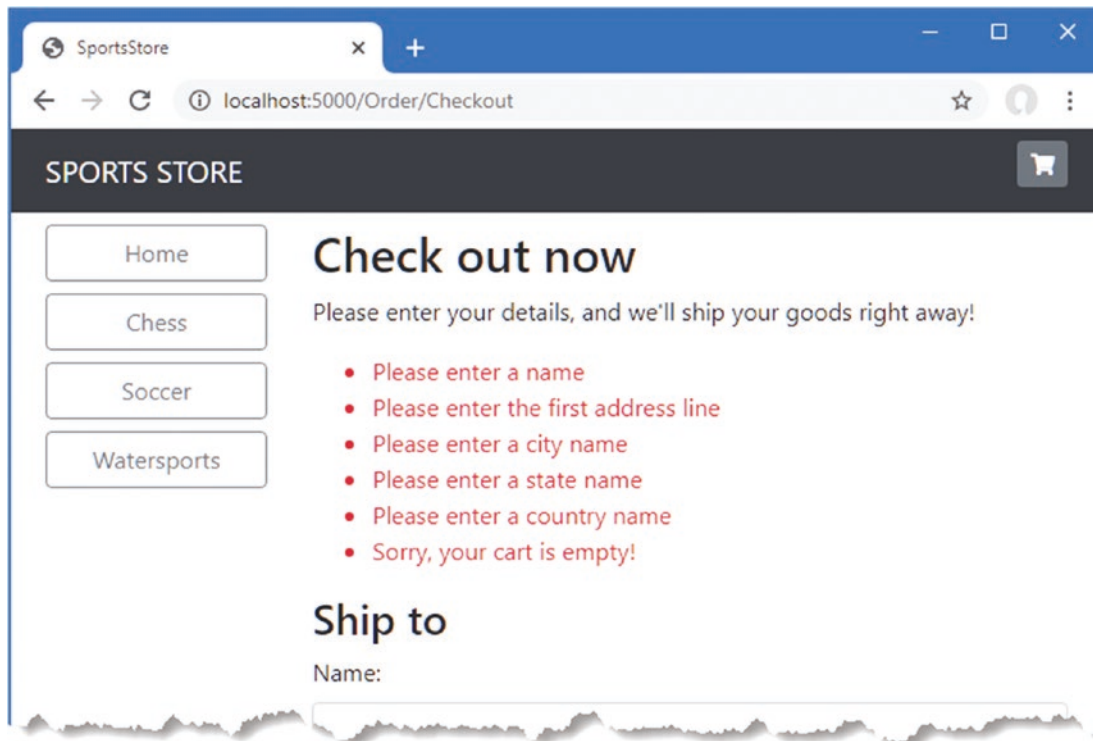


Figure 9-5. Displaying validation messages

■ **Tip** The data submitted by the user is sent to the server before it is validated, which is known as *server-side validation* and for which ASP.NET Core has excellent support. The problem with server-side validation is that the user isn't told about errors until after the data has been sent to the server and processed and the result page has been generated—something that can take a few seconds on a busy server. For this reason, server-side validation is usually complemented by *client-side validation*, where JavaScript is used to check the values that the user has entered before the form data is sent to the server. I describe client-side validation in Chapter 29.

Displaying a Summary Page

To complete the checkout process, I am going to create a Razor Page that displays a thank-you message with a summary of the order. Add a Razor Page named `Completed.cshtml` to the Pages folder with the contents shown in Listing 9-22.

Listing 9-22. The Contents of the `Completed.cshtml` File in the `SportsStore/Pages` Folder

```
@page
<div class="text-center">
  <h2>Thanks!</h2>
  <p>Thanks for placing order #@OrderId</p>
  <p>We'll ship your goods as soon as possible.</p>
  <a class="btn btn-primary" asp-controller="Home">Return to Store</a>
</div>

@functions {
    [BindProperty(SupportsGet = true)]
    public string OrderId { get; set; }
}
```

Although Razor Pages usually have page model classes, they are not a requirement, and simple features can be developed without them. In this example, I have defined a property named `OrderId` and decorated it with the `BindProperty` attribute, which specifies that a value for this property should be obtained from the request by the model binding system.

Now customers can go through the entire process, from selecting products to checking out. If they provide valid shipping details (and have items in their cart), they will see the summary page when they click the Complete Order button, as shown in Figure 9-6.

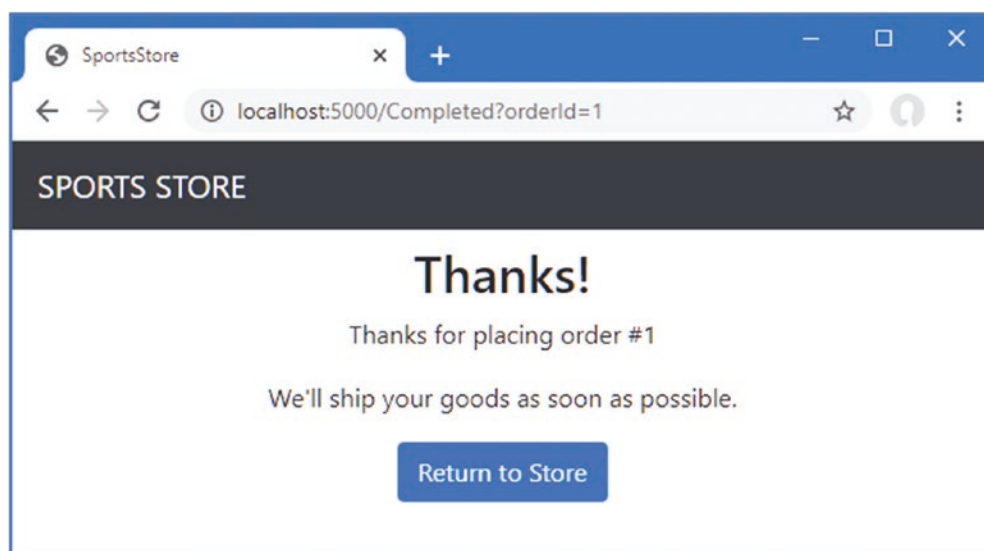


Figure 9-6. The completed order summary view

Notice the way the application moves between controllers and Razor Pages. The application features that ASP.NET Core provides are complementary and can be mixed freely in projects.

Summary

I have completed all the major parts of the customer-facing portion of SportsStore. It might not be enough to worry Amazon, but I have a product catalog that can be browsed by category and page, a neat shopping cart, and a simple checkout process.

The approach I have taken means I can easily change the behavior of any piece of the application without causing problems or inconsistencies elsewhere. For example, I could change the way that orders are stored, and it would not have any impact on the shopping cart, the product catalog, or any other area of the application. In the next chapter, I add the features required to administer the SportsStore application.

CHAPTER 10



SportsStore: Administration

In this chapter, I continue to build the SportsStore application in order to give the site administrator a way to manage orders and products. In this chapter, I use Blazor to create administration features. Blazor is a new addition to ASP.NET Core, and it combines client-side JavaScript code with server-side code executed by ASP.NET Core, connected by a persistent HTTP connection. I describe Blazor in detail in Chapters 32–35, but it is important to understand that the Blazor model is not suited to all projects. (I use Blazor Server in this chapter, which is a supported part of the ASP.NET Core platform. There is also Blazor WebAssembly, which is, at the time of writing, experimental and runs entirely in the browser. I describe Blazor WebAssembly in Chapter 36.)

■ **Tip** You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/apress/pro-asp.net-core-3>. See Chapter 1 for how to get help if you have problems running the examples.

Preparing Blazor Server

The first step is to enable the services and middleware for Blazor, as shown in Listing 10-1.

Listing 10-1. Enabling Blazor in the Startup.cs File in the SportsStore Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using SportsStore.Models;

namespace SportsStore {
    public class Startup {

        public Startup(IConfiguration config) {
            Configuration = config;
        }

        private IConfiguration Configuration { get; set; }

        public void ConfigureServices(IServiceCollection services) {
            services.AddControllersWithViews();
            services.AddDbContext<StoreDbContext>(opts => {
```

```

        opts.UseSqlServer(
            Configuration["ConnectionStrings:SportsStoreConnection"]);
    });
    services.AddScoped<IStoreRepository, EFStoreRepository>();
    services.AddScoped<IOrderRepository, EFOrderRepository>();
    services.AddRazorPages();
    services.AddDistributedMemoryCache();
    services.AddSession();
    services.AddScoped<Cart>(sp => SessionCart.GetCart(sp));
    services.AddSingleton<IHttpContextAccessor, HttpContextAccessor>();
    services.AddServerSideBlazor();
}

public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
    app.UseDeveloperExceptionPage();
    app.UseStatusCodePages();
    app.UseStaticFiles();
    app.UseSession();
    app.UseRouting();

    app.UseEndpoints(endpoints => {
        endpoints.MapControllerRoute("catpage",
            "{category}/Page{productPage:int}",
            new { Controller = "Home", action = "Index" });

        endpoints.MapControllerRoute("page", "Page{productPage:int}",
            new { Controller = "Home", action = "Index", productPage = 1 });

        endpoints.MapControllerRoute("category", "{category}",
            new { Controller = "Home", action = "Index", productPage = 1 });

        endpoints.MapControllerRoute("pagination",
            "Products/Page{productPage}",
            new { Controller = "Home", action = "Index", productPage = 1 });
        endpoints.MapDefaultControllerRoute();
        endpoints.MapRazorPages();
        endpoints.MapBlazorHub();
        endpoints.MapFallbackToPage("/admin/{*catchall}", "/Admin/Index");
    });

    SeedData.EnsurePopulated(app);
}
}
}

```

The `AddServerSideBlazor` method creates the services that Blazor uses, and the `MapBlazorHub` method registers the Blazor middleware components. The final addition is to finesse the routing system to ensure that Blazor works seamlessly with the rest of the application.

Creating the Imports File

Blazor requires its own imports file to specify the namespaces that it uses. Create the `Pages/Admin` folder and add to it a file named `_Imports.razor` with the content shown in Listing 10-2. (If you are using Visual Studio, you can use the Razor Components template to create this file.)

■ **Note** The conventional location for Blazor files is within the `Pages` folder, but Blazor files can be defined anywhere in the project. In Part 4, for example, I used a folder named `Blazor` to help emphasize which features were provided by Blazor and which by Razor Pages.

Listing 10-2. The Contents of the `_Imports.razor` File in the `SportsStore/Pages/Admin` Folder

```
@using Microsoft.AspNetCore.Components
@using Microsoft.AspNetCore.Components.Forms
@using Microsoft.AspNetCore.Components.Routing
@using Microsoft.AspNetCore.Components.Web
@using Microsoft.EntityFrameworkCore
@using SportsStore.Models
```

The first four `@using` expressions are for the namespaces required for Blazor. The last two expressions are for convenience in the examples that follow because they will allow me to use Entity Framework Core and the classes in the `Models` namespace.

Creating the Startup Razor Page

Blazor relies on a Razor Page to provide the initial content to the browser, which includes the JavaScript code that connects to the server and renders the Blazor HTML content. Add a Razor Page named `Index.cshtml` to the `Pages/Admin` folder with the contents shown in Listing 10-3.

Listing 10-3. The Contents of the `Index.cshtml` File in the `SportsStore/Pages/Admin` Folder

```
@page "/admin"
@{ Layout = null; }

<!DOCTYPE html>
<html>
<head>
  <title>SportsStore Admin</title>
  <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
  <base href="/" />
</head>
<body>
  <component type="typeof(Routed)" render-mode="Server" />
  <script src="/_framework/blazor.server.js"></script>
</body>
</html>
```

The component element is used to insert a Razor Component in the output from the Razor Page. Razor Components are the confusingly named Blazor building blocks, and the component element applied in Listing 10-3 is named `Routed` and will be created shortly. The Razor Page also contains a `script` element that tells the browser to load the JavaScript file that Blazor Server uses. Requests for this file are intercepted by the Blazor Server middleware, and you don't need to explicitly add the JavaScript file to the project.

Creating the Routing and Layout Components

Add a Razor Component named `Routed.razor` to the `Pages/Admin` folder and add the content shown in Listing 10-4.

Listing 10-4. The Contents of the `Routed.razor` File in the `SportsStore/Pages/Admin` Folder

```
<Router AppAssembly="typeof(Startup).Assembly">
  <Found>
    <RouteView RouteData="@context" DefaultLayout="typeof(AdminLayout)" />
  </Found>
  <NotFound>
    <h4 class="bg-danger text-white text-center p-2">
      No Matching Route Found
    </h4>
  </NotFound>
</Router>
```

The content of this component is described in detail in Part 4 of this book, but, for this chapter, it is enough to know that the component will use the browser's current URL to locate a Razor Component that can be displayed to the user. If no matching component can be found, then an error message is displayed.

Blazor has its own system of layouts. To create the layout for the administration tools, add a Razor Component named `AdminLayout.razor` to the `Pages/Admin` folder with the content shown in Listing 10-5.

Listing 10-5. The Contents of the `AdminLayout.razor` File in the `SportsStore/Pages/Admin` Folder

```
@inherits LayoutComponentBase

<div class="bg-info text-white p-2">
  <span class="navbar-brand ml-2">SPORTS STORE Administration</span>
</div>
<div class="container-fluid">
  <div class="row p-2">
    <div class="col-3">
      <NavLink class="btn btn-outline-primary btn-block"
        href="/admin/products"
        ActiveClass="btn-primary text-white"
        Match="NavLinkMatch.Prefix">
        Products
      </NavLink>
      <NavLink class="btn btn-outline-primary btn-block"
        href="/admin/orders"
        ActiveClass="btn-primary text-white"
        Match="NavLinkMatch.Prefix">
        Orders
      </NavLink>
    </div>
    <div class="col">
      @Body
    </div>
  </div>
</div>
```

Blazor uses Razor syntax to generate HTML but introduces its own directives and features. This layout renders a two-column display with `Product` and `Order` navigation buttons, which are created using `NavLink` elements. These elements apply a built-in Razor Component that changes the URL without triggering a new HTTP request, which allows Blazor to respond to user interaction without losing the application state.

Creating the Razor Components

To complete the initial setup, I need to add the components that will provide the administration tools, although they will contain placeholder messages at first. Add a Razor Component named `Products.razor` to the `Pages/Admin` folder with the content shown in Listing 10-6.

Listing 10-6. The Contents of the `Products.razor` File in the `SportsStore/Pages/Admin` Folder

```
@page "/admin/products"
@page "/admin"

<h4>This is the products component</h4>
```

The `@page` directives specify the URLs for which this component will be displayed, which is `/admin/products` and `/admin`. Next, add a Razor Component named `Orders.razor` to the `Pages/Admin` folder with the content shown in Listing 10-7.

Listing 10-7. The Contents of the Orders.razor File in the SportsStore/Pages/Admin Folder

```
@page "/admin/orders"

<h4>This is the orders component</h4>
```

Checking the Blazor Setup

To make sure that Blazor is working correctly, start ASP.NET Core and request `http://localhost:5000/admin`. This request will be handled by the Index Razor Page in the Pages/Admin folder, which will include the Blazor JavaScript file in the content it sends to the browser. The JavaScript code will open a persistent HTTP connection to the ASP.NET Core server, and the initial Blazor content will be rendered, as shown in Figure 10-1.

■ **Note** Microsoft has not yet released the tools required to test Razor Components, which is why there are no unit testing examples in this chapter.

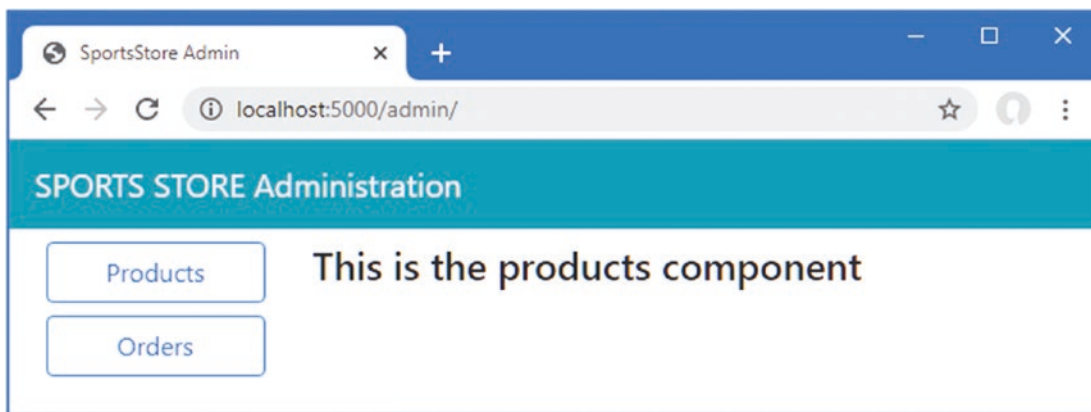


Figure 10-1. The Blazor application

Click the Orders button, and content generated by the Orders Razor Component will be displayed, as shown in Figure 10-2. Unlike the other ASP.NET Core application frameworks I used in earlier chapters, the new content is displayed without a new HTTP request being sent, even though the URL displayed by the browser changes.

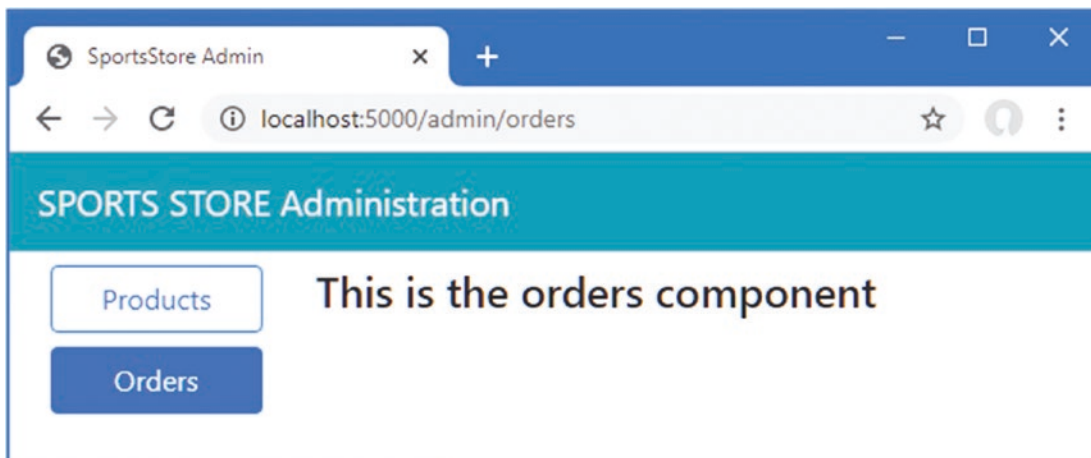


Figure 10-2. Navigating in the Blazor application

Managing Orders

Now that Blazor has been set up and tested, I am going to start implementing administration features. In the previous chapter, I added support for receiving orders from customers and storing them in a database. In this section, I am going to create a simple administration tool that will let me view the orders that have been received and mark them as shipped.

Enhancing the Model

The first change I need to make is to enhance the data model so that I can record which orders have been shipped. Listing 10-8 shows the addition of a new property to the `Order` class, which is defined in the `Order.cs` file in the `Models` folder.

Listing 10-8. Adding a Property in the `Order.cs` File in the `SportsStore/Models` Folder

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using Microsoft.AspNetCore.Mvc.ModelBinding;

namespace SportsStore.Models {
    public class Order {
        [BindNever]
        public int OrderID { get; set; }
        [BindNever]
        public ICollection<CartLine> Lines { get; set; }

        [Required(ErrorMessage = "Please enter a name")]
        public string Name { get; set; }

        [Required(ErrorMessage = "Please enter the first address line")]

        public string Line1 { get; set; }
        public string Line2 { get; set; }
        public string Line3 { get; set; }

        [Required(ErrorMessage = "Please enter a city name")]
        public string City { get; set; }

        [Required(ErrorMessage = "Please enter a state name")]
        public string State { get; set; }

        public string Zip { get; set; }

        [Required(ErrorMessage = "Please enter a country name")]
        public string Country { get; set; }

        public bool GiftWrap { get; set; }

        [BindNever]
        public bool Shipped { get; set; }
    }
}
```

This iterative approach of extending and adapting the data model to support different features is typical of ASP.NET Core development. In an ideal world, you would be able to completely define the data model at the start of the project and just build the application around it, but that happens only for the simplest of projects, and, in practice, iterative development is to be expected as the understanding of what is required develops and evolves.

Entity Framework Core migrations make this process easier because you don't have to manually keep the database schema synchronized to the model class by writing your own SQL commands. To update the database to reflect the addition of the `Shipped` property to the `Order` class, open a new PowerShell window and run the command shown in Listing 10-9 in the `SportsStore` project.

Listing 10-9. Creating a New Migration

```
dotnet ef migrations add ShippedOrders
```

The migration will be applied automatically when the application is started and the `SeedData` class calls the `Migrate` method provided by Entity Framework Core.

Displaying Orders to the Administrator

I am going to display two tables, one of which shows the orders waiting to be shipped and the other the shipped orders. Each order will be presented with a button that changes the shipping state. This is not entirely realistic because orders processing is typically more complex than simply updating a field in the database, but integration with warehouse and fulfillment systems is well beyond the scope of this book.

To avoid duplicating code and content, I am going to create a Razor Component that displays a table without knowing which category of order it is dealing with. Add a Razor Component named `OrderTable.razor` to the `Pages/Admin` folder with the content shown in Listing 10-10.

Listing 10-10. The Contents of the `OrderTable.razor` File in the `SportsStore/Pages/Admin` Folder

```
<table class="table table-sm table-striped table-bordered">
  <thead>
    <tr><th colspan="5" class="text-center">@TableTitle</th></tr>
  </thead>
  <tbody>
    @if (Orders?.Count() > 0) {
      @foreach (Order o in Orders) {
        <tr>
          <td>@o.Name</td><td>@o.Zip</td><th>Product</th><th>Quantity</th>
          <td>
            <button class="btn btn-sm btn-danger"
              @onclick="@{e => OrderSelected.InvokeAsync(o.OrderID)}">
              @ButtonLabel
            </button>
          </td>
        </tr>
        @foreach (CartLine line in o.Lines) {
          <tr>
            <td colspan="2"></td>
            <td>@line.Product.Name</td><td>@line.Quantity</td>
            <td></td>
          </tr>
        }
      }
    } else {
      <tr><td colspan="5" class="text-center">No Orders</td></tr>
    }
  </tbody>
</table>

@code {
  [Parameter]
  public string TableTitle { get; set; } = "Orders";

  [Parameter]
  public IEnumerable<Order> Orders { get; set; }
```

```

[Parameter]
public string ButtonLabel { get; set; } = "Ship";

[Parameter]
public EventCallback<int> OrderSelected { get; set; }
}

```

Razor Components, as the name suggests, rely on the Razor approach to annotated HTML elements. The view part of the component is supported by the statements in the `@code` section. The `@code` section in this component defines four properties that are decorated with the `Parameter` attribute, which means the values will be provided at runtime by the parent component, which I will create shortly. The values provided for the parameters are used in the view section of the component to display details of a sequence of `Order` objects.

Blazor adds expressions to the Razor syntax. The view section of this component includes this button element, which has an `onclick` attribute.

```

...
<button class="btn btn-sm btn-danger"
        @onclick="@{e => OrderSelected.InvokeAsync(o.OrderID)}">
    @ButtonLabel
</button>
...

```

This tells Blazor how to react when the user clicks the button. In this case, the expression tells Razor to call the `InvokeAsync` method of the `OrderSelected` property. This is how the table will communicate with the rest of the Blazor application and will become clearer as I build out additional features.

■ **Tip** I describe Blazor in-depth in Part 4 of this book, so don't worry if the Razor Components in this chapter do not make immediate sense. The purpose of the `SportsStore` example is to show the overall development process, even if individual features are not understood.

The next step is to create a component that will get the `Order` data from the database and use the `OrderTable` component to display it to the user. Remove the placeholder content in the `Orders` component and replace it with the code and content shown in Listing 10-11.

Listing 10-11. The Revised Contents of the `Orders.razor` File in the `SportsStore/Pages/Admin` Folder

```

@page "/admin/orders"
@inherits OwningComponentBase<IOrderRepository>

<OrderTable TableTitle="Unshipped Orders"
    Orders="UnshippedOrders" ButtonLabel="Ship" OrderSelected="ShipOrder" />
<OrderTable TableTitle="Shipped Orders"
    Orders="ShippedOrders" ButtonLabel="Reset" OrderSelected="ResetOrder" />
<button class="btn btn-info" @onclick="@{e => UpdateData()}">Refresh Data</button>

@code {

    public IOrderRepository Repository => Service;

    public IEnumerable<Order> AllOrders { get; set; }
    public IEnumerable<Order> UnshippedOrders { get; set; }
    public IEnumerable<Order> ShippedOrders { get; set; }

    protected async override Task OnInitializedAsync() {
        await UpdateData();
    }
}

```

```

public async Task UpdateData() {
    AllOrders = await Repository.Orders.ToListAsync();
    UnshippedOrders = AllOrders.Where(o => !o.Shipped);
    ShippedOrders = AllOrders.Where(o => o.Shipped);
}

public void ShipOrder(int id) => UpdateOrder(id, true);
public void ResetOrder(int id) => UpdateOrder(id, false);

private void UpdateOrder(int id, bool shipValue) {
    Order o = Repository.Orders.FirstOrDefault(o => o.OrderID == id);
    o.Shipped = shipValue;
    Repository.SaveOrder(o);
}
}
}

```

Blazor Components are not like the other application framework building blocks used for the user-facing sections of the SportsStore application. Instead of dealing with individual requests, components can be long-lived and deal with multiple user interactions over a longer period. This requires a different style of development, especially when it comes to dealing with data using Entity Framework Core. The `@inherits` expression ensures that this component gets its own repository object, which ensures its operations are separate from those performed by other components displayed to the same user. And to avoid repeatedly querying the database—which can be a serious problem in Blazor, as I explain in Part 4—the repository is used only when the component is initialized, when Blazor invokes the `OnInitializedAsync` method, or when the user clicks a Refresh Data button.

To display its data to the user, the `OrderTable` component is used, which is applied as an HTML element, like this:

```

...
<OrderTable TableTitle="Unshipped Orders"
    Orders="UnshippedOrders" ButtonLabel="Ship" OrderSelected="ShipOrder" />
...

```

The values assigned to the `OrderTable` element's attributes are used to set the properties decorated with the `Parameter` attribute in Listing 10-10. In this way, a single component can be configured to present two different sets of data without the need to duplicate code and content.

The `ShipOrder` and `ResetOrder` methods are used as the values for the `OrderSelected` attributes, which means they are invoked when the user clicks one of the buttons presented by the `OrderTable` component, updating the data in the database through the repository.

To see the new features, restart ASP.NET Core, request `http://localhost:5000`, and create an order. Once you have at least one order in the database, request `http://localhost:5000/admin/orders`, and you will see a summary of the order you created displayed in the Unshipped Orders table. Click the Ship button, and the order will be updated and moved to the Shipped Orders table, as shown in Figure 10-3.

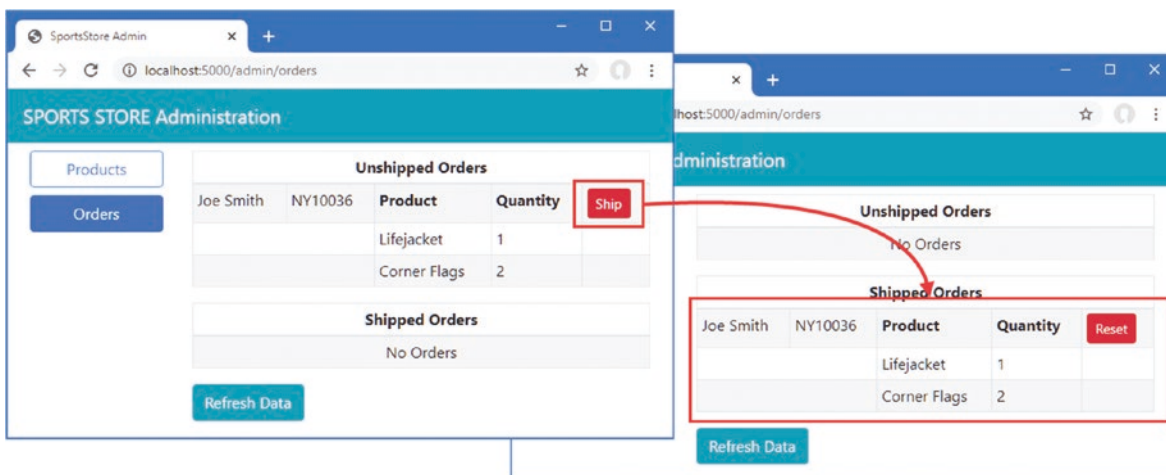


Figure 10-3. Administering orders

Adding Catalog Management

The convention for managing more complex collections of items is to present the user with two interfaces: a *list* interface and an *edit* interface, as shown in Figure 10-4.

Figure 10-4. Sketch of a CRUD UI for the product catalog

Together, these interfaces allow a user to create, read, update, and delete items in the collection. Collectively, these actions are known as *CRUD*. In this section, I will implement these interfaces using Blazor.

■ **Tip** Developers need to implement CRUD so often that Visual Studio scaffolding includes scenarios for creating CRUD controllers or Razor Pages. But, like all Visual Studio scaffolding, I think it is better to learn how to create these features directly, which is why I demonstrate CRUD operations for all the ASP.NET Core application frameworks in later chapters.

Expanding the Repository

The first step is to add features to the repository that will allow *Product* objects to be created, modified, and deleted. Listing 10-12 adds new methods to the *IStoreRepository* interface.

Listing 10-12. Adding Methods in the *IStoreRepository.cs* File in the *SportsStore/Models* Folder

```
using System.Linq;

namespace SportsStore.Models {
    public interface IStoreRepository {

        IQueryable<Product> Products { get; }

        void SaveProduct(Product p);
        void CreateProduct(Product p);
        void DeleteProduct(Product p);
    }
}
```

Listing 10-13 adds implementations of these methods to the Entity Framework Core repository class.

Listing 10-13. Implementing Methods in the EFStoreRepository.cs File in the SportsStore/Models Folder

```
using System.Linq;

namespace SportsStore.Models {
    public class EFStoreRepository : IStoreRepository {
        private StoreDbContext context;

        public EFStoreRepository(StoreDbContext ctx) {
            context = ctx;
        }

        public IQueryable<Product> Products => context.Products;

        public void CreateProduct(Product p) {
            context.Add(p);
            context.SaveChanges();
        }

        public void DeleteProduct(Product p) {
            context.Remove(p);
            context.SaveChanges();
        }

        public void SaveProduct(Product p) {
            context.SaveChanges();
        }
    }
}
```

Applying Validation Attributes to the Data Model

I want to validate the values the user provides when editing or creating Product objects, just as I did for the customer checkout process. In Listing 10-14, I have added validation attributes to the Product data model class.

Listing 10-14. Adding Validation Attributes in the Product.cs File in the SportsStore/Models Folder

```
using System.ComponentModel.DataAnnotations.Schema;
using System.ComponentModel.DataAnnotations;

namespace SportsStore.Models {

    public class Product {
        public long ProductID { get; set; }

        [Required(ErrorMessage = "Please enter a product name")]
        public string Name { get; set; }

        [Required(ErrorMessage = "Please enter a description")]
        public string Description { get; set; }

        [Required]
        [Range(0.01, double.MaxValue,
            ErrorMessage = "Please enter a positive price")]
        [Column(TypeName = "decimal(8, 2)")]
        public decimal Price { get; set; }
    }
}
```

```

    [Required(ErrorMessage = "Please specify a category")]
    public string Category { get; set; }
}
}

```

Blazor uses the same approach to validation as the rest of ASP.NET Core but, as you will see, applies it a different way to deal with the more interactive nature of Razor Components.

Creating the List Component

I am going to start by creating the table that will present the user with a table of products and the links that will allow them to be inspected and edited. Replace the contents of the `Products.razor` file with those shown in Listing 10-15.

Listing 10-15. The Revised Contents of the `Products.razor` File in the `SportsStore/Pages/Admin` Folder

```

@page "/admin/products"
@page "/admin"
@inherits OwningComponentBase<IStoreRepository>

<table class="table table-sm table-striped table-bordered">
  <thead>
    <tr>
      <th>ID</th><th>Name</th>
      <th>Category</th><th>Price</th><td/>
    </tr>
  </thead>
  <tbody>
    @if (ProductData?.Count() > 0) {
      @foreach (Product p in ProductData) {
        <tr>
          <td>@p.ProductID</td>
          <td>@p.Name</td>
          <td>@p.Category</td>
          <td>@p.Price.ToString("c")</td>
          <td>
            <NavLink class="btn btn-info btn-sm"
              href="@GetDetailsUrl(p.ProductID)">
              Details
            </NavLink>
            <NavLink class="btn btn-warning btn-sm"
              href="@GetEditUrl(p.ProductID)">
              Edit
            </NavLink>
          </td>
        </tr>
      }
    } else {
      <tr>
        <td colspan="5" class="text-center">No Products</td>
      </tr>
    }
  </tbody>
</table>

<NavLink class="btn btn-primary" href="/admin/products/create">Create</NavLink>

```

```
@code {
    public IStoreRepository Repository => Service;

    public IEnumerable<Product> ProductData { get; set; }

    protected async override Task OnInitializedAsync() {
        await UpdateData();
    }

    public async Task UpdateData() {
        ProductData = await Repository.Products.ToListAsync();
    }

    public string GetDetailsUrl(long id) => $"/admin/products/details/{id}";
    public string GetEditUrl(long id) => $"/admin/products/edit/{id}";
}
```

The component presents each Product object in the repository in a table row with NavLink components that will navigate to the components that will provide a detailed view and an editor. There is also a button that navigates to the component that will allow new Product objects to be created and stored in the database. Restart ASP.NET Core and request `http://localhost:5000/admin/products`, and you will see the content shown in Figure 10-5, although none of the buttons presented by the Products component work currently because I have yet to create the components they target.

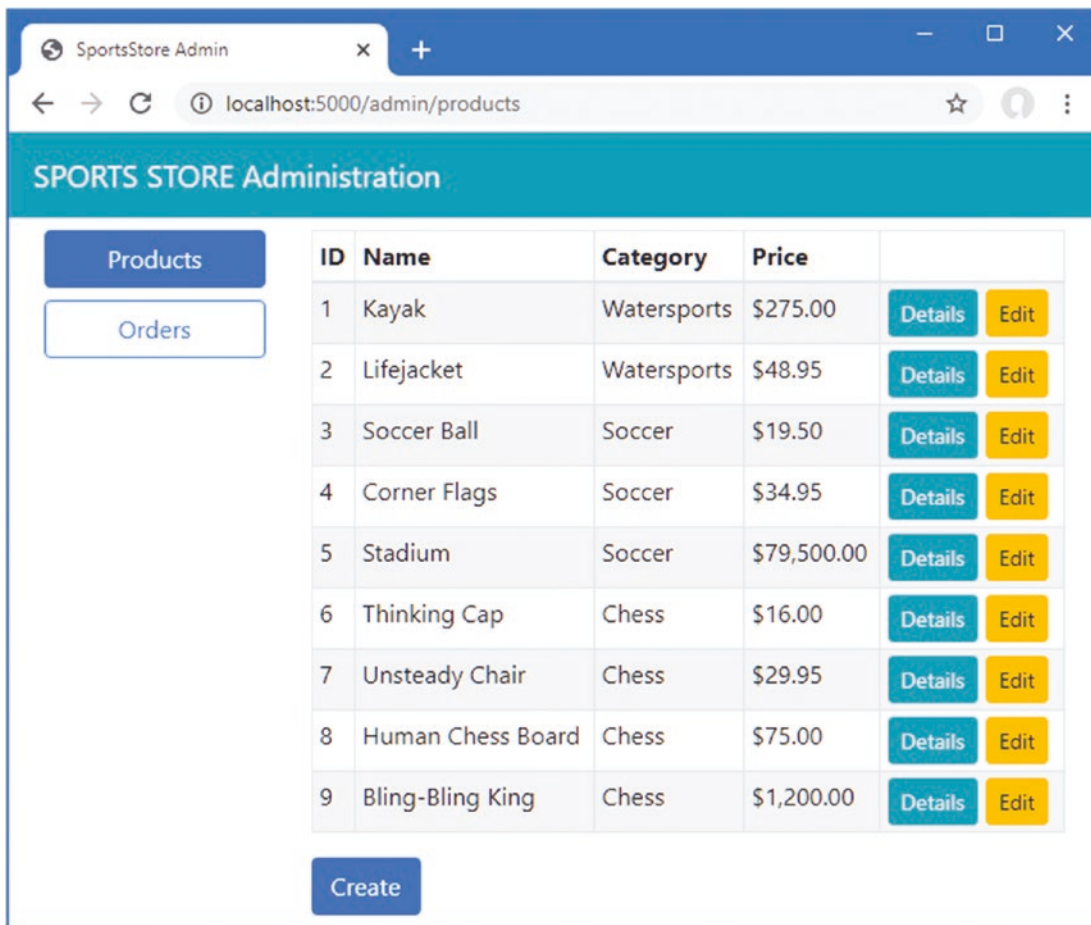


Figure 10-5. Presenting a list of products

Creating the Detail Component

The job of the detail component is to display all the fields for a single `Product` object. Add a Razor Component named `Details.razor` to the `Pages/Admin` folder with the content shown in Listing 10-16.

Listing 10-16. The Contents of the `Details.razor` File in the `SportsStore/Pages/Admin` Folder

```
@page "/admin/products/details/{id:long}"

<h3 class="bg-info text-white text-center p-1">Details</h3>

<table class="table table-sm table-bordered table-striped">
  <tbody>
    <tr><th>ID</th><td>@Product.ProductID</td></tr>
    <tr><th>Name</th><td>@Product.Name</td></tr>
    <tr><th>Description</th><td>@Product.Description</td></tr>
    <tr><th>Category</th><td>@Product.Category</td></tr>
    <tr><th>Price</th><td>@Product.Price.ToString("C")</td></tr>
  </tbody>
</table>

<NavLink class="btn btn-warning" href="@EditUrl">Edit</NavLink>
<NavLink class="btn btn-secondary" href="/admin/products">Back</NavLink>

@code {

  [Inject]
  public IStoreRepository Repository { get; set; }

  [Parameter]
  public long Id { get; set; }

  public Product Product { get; set; }

  protected override void OnParametersSet() {
    Product = Repository.Products.FirstOrDefault(p => p.ProductID == Id);
  }

  public string EditUrl => $"/admin/products/edit/{Product.ProductID}";
}
```

The component uses the `Inject` attribute to declare that it requires an implementation of the `IStoreRepository` interface, which is one of the ways that Blazor provides access to the application's services. The value of the `Id` property will be populated from the URL that has been used to navigate to the component, which is used to retrieve the `Product` object from the database. To see the detail view, restart ASP.NET Core, request `http://localhost:5000/admin/products`, and click one of the `Details` buttons, as shown in Figure 10-6.

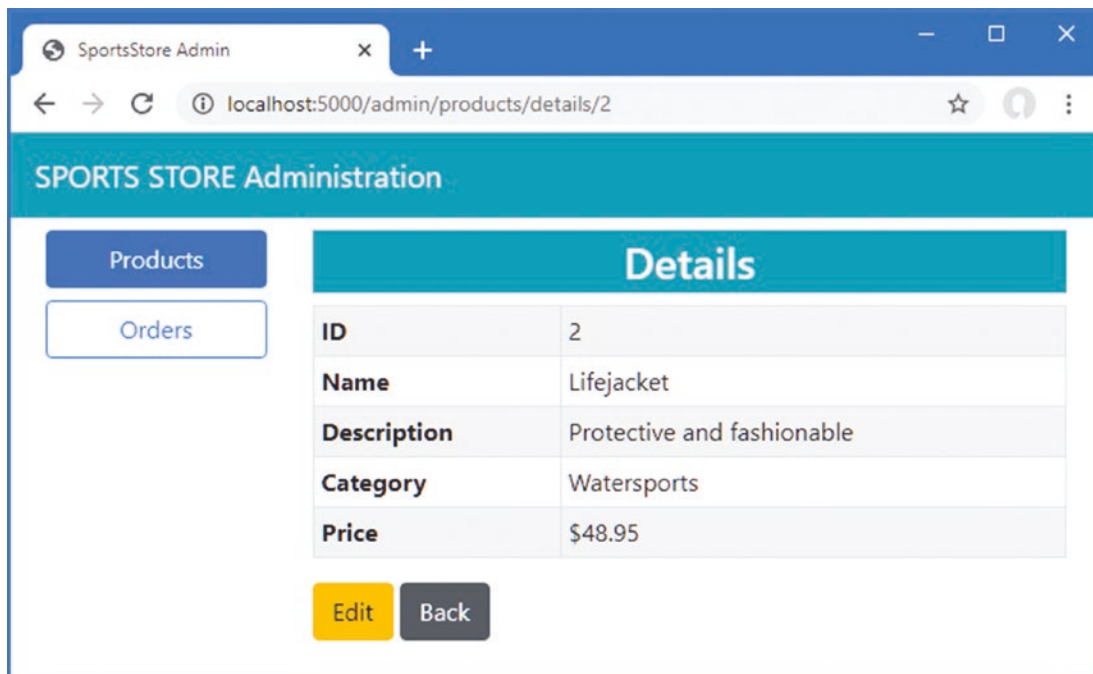


Figure 10-6. Displaying details of a product

Creating the Editor Component

The operations to create and edit data will be handled by the same component. Add a Razor Component named `Editor.razor` to the `Pages/Admin` folder with the content shown in Listing 10-17.

Listing 10-17. The Contents of the `Editor.razor` File in the `SportsStore/Pages/Admin` Folder

```
@page "/admin/products/edit/{id:long}"
@page "/admin/products/create"
@inherits OwinComponentBase<IStoreRepository>

<style>
  div.validation-message { color: rgb(220, 53, 69); font-weight: 500 }
</style>

<h3 class="bg-@ThemeColor text-white text-center p-1">@TitleText a Product</h3>
<EditForm Model="Product" OnValidSubmit="SaveProduct">
  <DataAnnotationsValidator />
  @if(Product.ProductID != 0) {
    <div class="form-group">
      <label>ID</label>
      <input class="form-control" disabled value="@Product.ProductID" />
    </div>
  }
  <div class="form-group">
    <label>Name</label>
    <ValidationMessage For="@(() => Product.Name)" />
    <InputText class="form-control" @bind-Value="Product.Name" />
  </div>
</EditForm>
```

```

<div class="form-group">
  <label>Description</label>
  <ValidationMessage For="@(() => Product.Description)" />
  <InputText class="form-control" @bind-Value="Product.Description" />
</div>
<div class="form-group">
  <label>Category</label>
  <ValidationMessage For="@(() => Product.Category)" />
  <InputText class="form-control" @bind-Value="Product.Category" />
</div>
<div class="form-group">
  <label>Price</label>
  <ValidationMessage For="@(() => Product.Price)" />
  <InputNumber class="form-control" @bind-Value="Product.Price" />
</div>
<button type="submit" class="btn btn-@ThemeColor">Save</button>
<NavLink class="btn btn-secondary" href="/admin/products">Cancel</NavLink>
</EditForm>

```

```

@code {

  public IStoreRepository Repository => Service;

  [Inject]
  public NavigationManager NavManager { get; set; }

  [Parameter]
  public long Id { get; set; } = 0;

  public Product Product { get; set; } = new Product();

  protected override void OnParametersSet() {
    if (Id != 0) {
      Product = Repository.Products.FirstOrDefault(p => p.ProductID == Id);
    }
  }

  public void SaveProduct() {
    if (Id == 0) {
      Repository.CreateProduct(Product);
    } else {
      Repository.SaveProduct(Product);
    }
    NavManager.NavigateTo("/admin/products");
  }

  public string ThemeColor => Id == 0 ? "primary" : "warning";
  public string TitleText => Id == 0 ? "Create" : "Edit";
}

```

Blazor provides a set of built-in Razor Components that are used to display and validate forms, which is important because the browser can't submit data using a POST request in a Blazor Component. The `EditForm` component is used to render a Blazor-friendly form, and the `InputText` and `InputNumber` components render input elements that accept string and number values and that automatically update a model property when the user makes a change.

Data validation is integrated into these built-in components, and the `OnValidSubmit` attribute on the `EditForm` component is used to specify a method that is invoked only if the data entered into the form conforms to the rules defined by the validation attributes.

Blazor also provides the `NavigationManager` class, which is used to programmatically navigate between components without triggering a new HTTP request. The `Editor` component uses `NavigationManager`, which is obtained as a service, to return to the `Products` component after the database has been updated.

To see the editor, restart ASP.NET Core, request `http://localhost:5000/admin`, and click the `Create` button. Click the `Save` button without filling out the form fields, and you will see the validation errors that Blazor produces automatically, as shown in Figure 10-7. Fill out the form and click `Save` again, and you will see the product you created displayed in the table, also as shown in Figure 10-7.

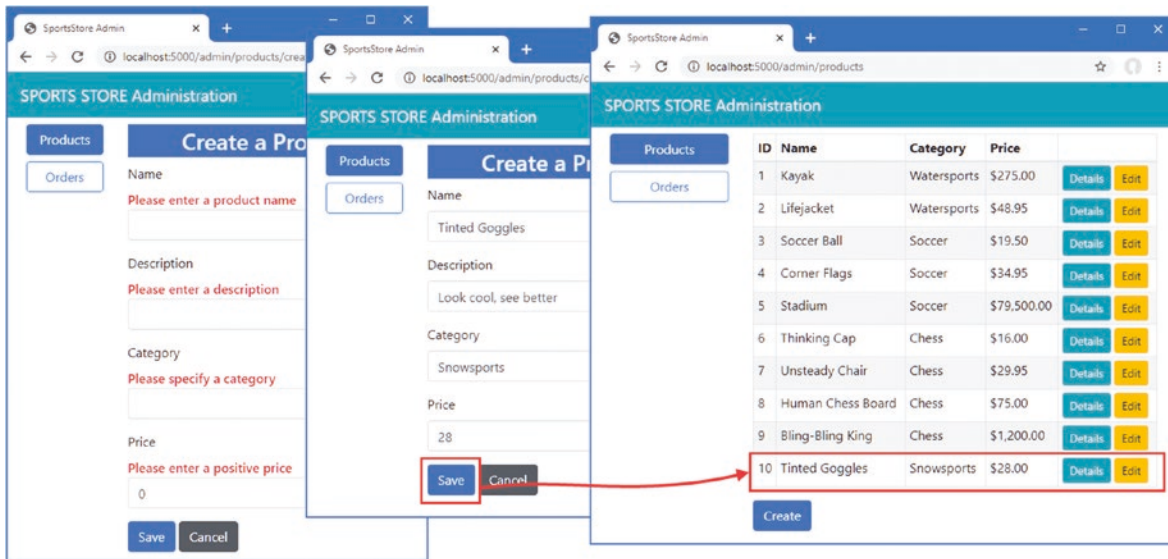


Figure 10-7. Using the Editor component

Click the `Edit` button for one of the products, and the same component will be used to edit the selected `Product` object's properties. Click the `Save` button, and any changes you made—if they pass validation—will be stored in the database, as shown in Figure 10-8.

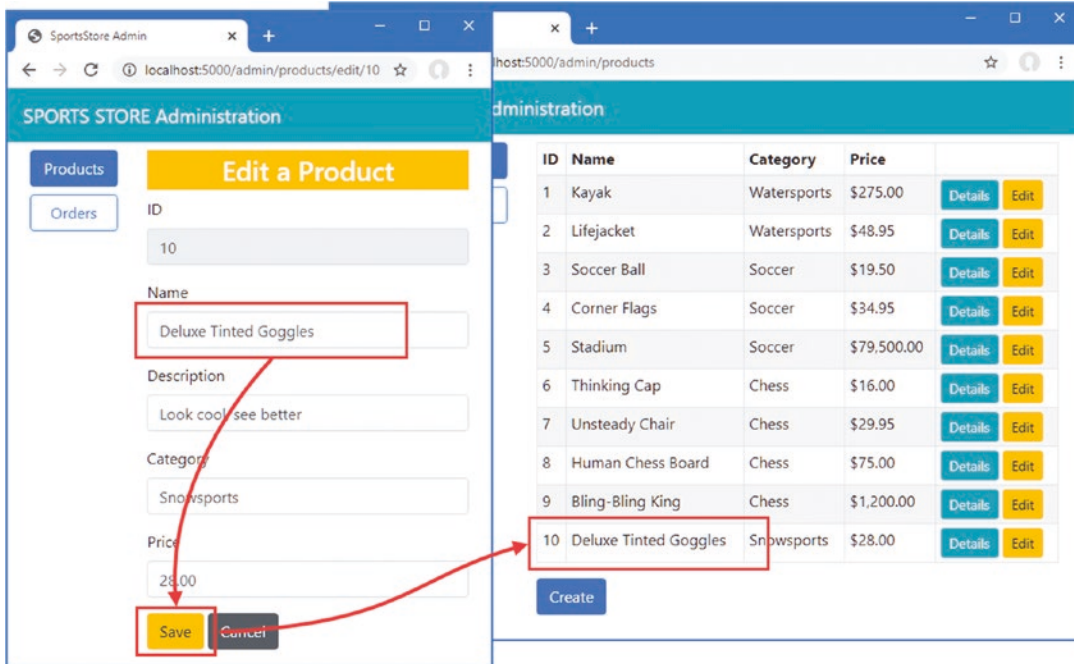


Figure 10-8. Editing products

Deleting Products

The final CRUD feature is deleting products, which is easily implemented in the Products component, as shown in Listing 10-18.

Listing 10-18. Adding Delete Support in the Products.razor File in the SportsStore/Pages/Admin Folder

```
@page "/admin/products"
@page "/admin"
@inherits OwningComponentBase<IStoreRepository>

<table class="table table-sm table-striped table-bordered">
  <thead>
    <tr>
      <th>ID</th><th>Name</th>
      <th>Category</th><th>Price</th><td/>
    </tr>
  </thead>
  <tbody>
    @if (ProductData?.Count() > 0) {
      @foreach (Product p in ProductData) {
        <tr>
          <td>@p.ProductID</td>
          <td>@p.Name</td>
          <td>@p.Category</td>
          <td>@p.Price.ToString("c")</td>
          <td>
            <NavLink class="btn btn-info btn-sm"
              href="@GetDetailsUrl(p.ProductID)">
              Details
            </NavLink>
            <NavLink class="btn btn-warning btn-sm"
              href="@GetEditUrl(p.ProductID)">
              Edit
            </NavLink>
            <button class="btn btn-danger btn-sm"
              @onclick="@{e => DeleteProduct(p)}">
              Delete
            </button>
          </td>
        </tr>
      }
    } else {
      <tr>
        <td colspan="5" class="text-center">No Products</td>
      </tr>
    }
  </tbody>
</table>

<NavLink class="btn btn-primary" href="/admin/products/create">Create</NavLink>

@code {
    public IStoreRepository Repository => Service;
    public IEnumerable<Product> ProductData { get; set; }
```

```

protected async override Task OnInitializedAsync() {
    await UpdateData();
}

public async Task UpdateData() {
    ProductData = await Repository.Products.ToListAsync();
}

public async Task DeleteProduct(Product p) {
    Repository.DeleteProduct(p);
    await UpdateData();
}

public string GetDetailsUrl(long id) => $"/admin/products/details/{id}";
public string GetEditUrl(long id) => $"/admin/products/edit/{id}";
}

```

The new button element is configured with the `@onclick` attribute, which invokes the `DeleteProduct` method. The selected `Product` object is removed from the database, and the data displayed by the component is updated. Restart ASP.NET Core, request `http://localhost:5000/admin/products`, and click a Delete button to remove an object from the database, as shown in Figure 10-9.

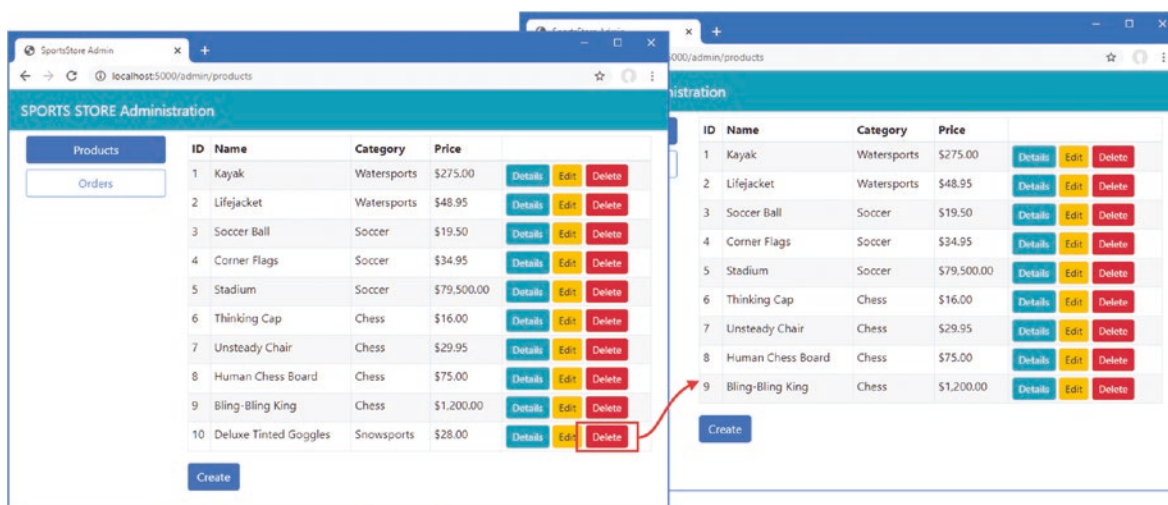


Figure 10-9. Deleting objects from the database

Summary

In this chapter, I introduced the administration capability and showed you how to use Blazor Server to implement CRUD operations that allow the administrator to create, read, update, and delete products from the repository and mark orders as shipped. In the next chapter, I show you how to secure the administration functions so that they are not available to all users, and I prepare the `SportsStore` application for deployment into production.



SportsStore: Security and Deployment

Securing the Administration Features

Authentication and authorization are provided by the ASP.NET Core Identity system, which integrates neatly into the ASP.NET Core platform and the individual application frameworks. In the sections that follow, I will create a basic security setup that allows one user, called `Admin`, to authenticate and access the administration features in the application. ASP.NET Core Identity provides many more features for authenticating users and authorizing access to application features and data, and you can find more information in Chapters 37 and 38, where I show you how to create and manage user accounts and how to perform authorization using roles. But, as I noted previously, ASP.NET Core Identity is a large framework in its own right, and I cover only the basic features in this book.

My goal in this chapter is just to get enough functionality in place to prevent customers from being able to access the sensitive parts of the SportsStore application and, in doing so, give you a flavor of how authentication and authorization fit into an ASP.NET Core application.

■ **Tip** You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/apress/pro-asp.net-core-3>. See Chapter 1 for how to get help if you have problems running the examples.

Creating the Identity Database

The ASP.NET Identity system is endlessly configurable and extensible and supports lots of options for how its user data is stored. I am going to use the most common, which is to store the data using Microsoft SQL Server accessed using Entity Framework Core.

Installing the Identity Package for Entity Framework Core

To add the package that contains the ASP.NET Core Identity support for Entity Framework Core, use a PowerShell command prompt to run the command shown in Listing 11-1 in the SportsStore folder.

Listing 11-1. Installing the Entity Framework Core Package

```
dotnet add package Microsoft.AspNetCore.Identity.EntityFrameworkCore --version 3.1.0
```

Creating the Context Class

I need to create a database context file that will act as the bridge between the database and the Identity model objects it provides access to. I added a class file called `AppIdentityDbContext.cs` to the `Models` folder and used it to define the class shown in Listing 11-2.

Listing 11-2. The Contents of the `AppIdentityDbContext.cs` File in the `SportsStore/Models` Folder

```
using Microsoft.AspNetCore.Identity;  
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;  
using Microsoft.EntityFrameworkCore;
```

```
namespace SportsStore.Models {

    public class AppIdentityDbContext : IdentityDbContext<IdentityUser> {

        public AppIdentityDbContext(DbContextOptions<AppIdentityDbContext> options)
            : base(options) { }

    }

}
```

The `AppIdentityDbContext` class is derived from `IdentityDbContext`, which provides Identity-specific features for Entity Framework Core. For the type parameter, I used the `IdentityUser` class, which is the built-in class used to represent users.

Defining the Connection String

The next step is to define the connection string that will be for the database. Listing 11-3 shows the connection string to the `appsettings.json` file of the `SportsStore` project, which follows the same format as the connection string that I defined for the product database.

Listing 11-3. Defining a Connection String in the `appsettings.json` File in the `SportsStore` Folder

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "SportsStoreConnection": "Server=(localdb)\\MSSQLLocalDB;Database=SportsStore;MultipleActiveResultSets=true",
    "IdentityConnection": "Server=(localdb)\\MSSQLLocalDB;Database=Identity;MultipleActiveResultSets=true"
  }
}
```

Remember that the connection string has to be defined in a single unbroken line in the `appsettings.json` file and is shown across multiple lines in the listing only because of the fixed width of a book page. The addition in the listing defines a connection string called `IdentityConnection` that specifies a LocalDB database called `Identity`.

Configuring the Application

Like other ASP.NET Core features, Identity is configured in the `Startup` class. Listing 11-4 shows the additions I made to set up Identity in the `SportsStore` project, using the context class and connection string defined previously.

Listing 11-4. Configuring Identity in the `Startup.cs` File in the `SportsStore` Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Configuration;
```



```

using Microsoft.EntityFrameworkCore;
using SportsStore.Models;
using Microsoft.AspNetCore.Identity;

namespace SportsStore {
    public class Startup {

        public Startup(IConfiguration config) {
            Configuration = config;
        }

        private IConfiguration Configuration { get; set; }

        public void ConfigureServices(IServiceCollection services) {
            services.AddControllersWithViews();
            services.AddDbContext<StoreDbContext>(opts => {
                opts.UseSqlServer(
                    Configuration["ConnectionStrings:SportsStoreConnection"]);
            });
            services.AddScoped<IStoreRepository, EFStoreRepository>();
            services.AddScoped<IOrderRepository, EFOrderRepository>();
            services.AddRazorPages();
            services.AddDistributedMemoryCache();
            services.AddSession();
            services.AddScoped<Cart>(sp => SessionCart.GetCart(sp));
            services.AddSingleton<IHttpContextAccessor, HttpContextAccessor>();
            services.AddServerSideBlazor();

            services.AddDbContext<AppIdentityDbContext>(options =>
                options.UseSqlServer(
                    Configuration["ConnectionStrings:IdentityConnection"]));

            services.AddIdentity<IdentityUser, IdentityRole>()
                .AddEntityFrameworkStores<AppIdentityDbContext>();
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseStatusCodePages();
            app.UseStaticFiles();
            app.UseSession();
            app.UseRouting();

            app.UseAuthentication();
            app.UseAuthorization();

            app.UseEndpoints(endpoints => {
                endpoints.MapControllerRoute("catpage",
                    "{category}/Page{productPage:int}",
                    new { Controller = "Home", action = "Index" });

                endpoints.MapControllerRoute("page", "Page{productPage:int}",
                    new { Controller = "Home", action = "Index", productPage = 1 });

                endpoints.MapControllerRoute("category", "{category}",
                    new { Controller = "Home", action = "Index", productPage = 1 });
            });
        }
    }
}

```

```
        endpoints.MapControllerRoute("pagination",
            "Products/Page{productPage}",
            new { Controller = "Home", action = "Index", productPage = 1 });
        endpoints.MapDefaultControllerRoute();
        endpoints.MapRazorPages();
        endpoints.MapBlazorHub();
        endpoints.MapFallbackToPage("/admin/{*catchall}", "/Admin/Index");
    });
}
SeedData.EnsurePopulated(app);
}
}
```

In the `ConfigureServices` method, I extended the Entity Framework Core configuration to register the context class and used the `AddIdentity` method to set up the Identity services using the built-in classes to represent users and roles.

In the `Configure` method, I called the `UseAuthentication` and `UseAuthorization` methods to set up the middleware components that implement the security policy. These methods must appear between the `UseRouting` and `UseEndpoints` methods.

Creating and Applying the Database Migration

The basic configuration is in place, and it is time to use the Entity Framework Core migrations feature to define the schema and apply it to the database. Open a new command prompt or PowerShell window and run the command shown in Listing 11-5 in the `SportsStore` folder to create a new migration for the Identity database.

Listing 11-5. Creating the Identity Migration

```
dotnet ef migrations add Initial --context AppIdentityDbContext
```

The important difference from previous database commands is that I have used the `-context` argument to specify the name of the context class associated with the database that I want to work with, which is `AppIdentityDbContext`. When you have multiple databases in the application, it is important to ensure that you are working with the right context class.

Once Entity Framework Core has generated the initial migration, run the command shown in Listing 11-6 in the `SportsStore` folder to create the database and apply the migration.

Listing 11-6. Applying the Identity Migration

```
dotnet ef database update --context AppIdentityDbContext
```

The result is a new LocalDB database called Identity that you can inspect using the Visual Studio SQL Server Object Explorer.

Defining the Seed Data

I am going to explicitly create the Admin user by seeding the database when the application starts. I added a class file called `IdentitySeedData.cs` to the `Models` folder and defined the static class shown in Listing 11-7.

Listing 11-7. The Contents of the IdentitySeedData.cs File in the SportsStore/Models Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Identity;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.EntityFrameworkCore;
using System.Linq;
```

```

namespace SportsStore.Models {

    public static class IdentitySeedData {
        private const string adminUser = "Admin";
        private const string adminPassword = "Secret123$";

        public static async void EnsurePopulated(IApplicationBuilder app) {

            AppIdentityDbContext context = app.ApplicationServices
                .CreateScope().ServiceProvider
                .GetRequiredService<AppIdentityDbContext>();
            if (context.Database.GetPendingMigrations().Any()) {
                context.Database.Migrate();
            }

            UserManager<IdentityUser> userManager = app.ApplicationServices
                .CreateScope().ServiceProvider
                .GetRequiredService<UserManager<IdentityUser>>();

            IdentityUser user = await userManager.FindByIdAsync(adminUser);
            if (user == null) {
                user = new IdentityUser("Admin");
                user.Email = "admin@example.com";
                user.PhoneNumber = "555-1234";
                await userManager.CreateAsync(user, adminPassword);
            }
        }
    }
}

```

This code ensures the database is created and up-to-date and uses the `UserManager<T>` class, which is provided as a service by ASP.NET Core Identity for managing users, as described in Chapter 38. The database is searched for the `Admin` user account, which is created—with a password of `Secret123$`—if it is not present. Do not change the hard-coded password in this example because Identity has a validation policy that requires passwords to contain a number and range of characters. See Chapter 38 for details of how to change the validation settings.

■ **Caution** Hard-coding the details of an administrator account is often required so that you can log into an application once it has been deployed and start administering it. When you do this, you must remember to change the password for the account you have created. See Chapter 38 for details of how to change passwords using Identity. See Chapter 15 for how to keep sensitive data, such as default passwords, out of source code control.

To ensure that the Identity database is seeded when the application starts, I added the statement shown in Listing 11-8 to the `Configure` method of the `Startup` class.

Listing 11-8. Seeding the Identity Database in the `Startup.cs` File in the `SportsStore` Folder

```

...
public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
    app.UseDeveloperExceptionPage();
    app.UseStatusCodePages();
    app.UseStaticFiles();
    app.UseSession();
    app.UseAuthentication();
    app.UseRouting();
}

```

```

app.UseEndpoints(endpoints => {
    endpoints.MapControllerRoute("catpage",
        "{category}/Page{productPage:int}",
        new { Controller = "Home", action = "Index" });

    endpoints.MapControllerRoute("page", "Page{productPage:int}",
        new { Controller = "Home", action = "Index", productPage = 1 });

    endpoints.MapControllerRoute("category", "{category}",
        new { Controller = "Home", action = "Index", productPage = 1 });

    endpoints.MapControllerRoute("pagination",
        "Products/Page{productPage}",
        new { Controller = "Home", action = "Index", productPage = 1 });
    endpoints.MapDefaultControllerRoute();
    endpoints.MapRazorPages();
    endpoints.MapBlazorHub();
    endpoints.MapFallbackToPage("/admin/{*catchall}", "/Admin/Index");
});

SeedData.EnsurePopulated(app);
IdentitySeedData.EnsurePopulated(app);
}
...

```

DELETING AND RE-CREATING THE ASP.NET CORE IDENTITY DATABASE

If you need to reset the Identity database, then run the following command:

```
dotnet ef database drop --force --context AppIdentityDbContext
```

Restart the application, and the database will be re-created and populated with seed data.

Adding a Conventional Administration Feature

In Listing 11-9, I used Blazor to create the administration features so that I could demonstrate a wide range of ASP.NET Core features in the SportsStore project. Although Blazor is useful, it is not suitable for all projects—as I explain in Part 4—and most projects are likely to use controllers or Razor Pages for their administration features. I describe the way that ASP.NET Core Identity works with all the application frameworks in Chapter 38, but just to provide a balance to the all-Blazor tools created in Chapter 10, I am going to create a Razor Page that will display the list of users in the ASP.NET Core Identity database. I describe how to manage the Identity database in more detail in Chapter 38, and this Razor Page is just to add a sensitive feature to the SportsStore application that isn't created with Blazor. Add a Razor Page named `IdentityUsers.cshtml` to the `SportsStore/Pages/Admin` folder with the contents shown in Listing 11-9.

Listing 11-9. The Contents of the `IdentityUsers.cshtml` File in the `SportsStore/Pages/Admin` Folder

```

@page
@model IdentityUsersModel
@using Microsoft.AspNetCore.Identity

<h3 class="bg-primary text-white text-center p-2">Admin User</h3>

<table class="table table-sm table-striped table-bordered">
    <tbody>
        <tr><th>User</th><td>@Model.AdminUser.UserName</td></tr>
        <tr><th>Email</th><td>@Model.AdminUser.Email</td></tr>
    </tbody>
</table>

```

```

        <tr><th>Phone</th><td>@Model.AdminUser.PhoneNumber</td></tr>
    </tbody>
</table>

@functions{

    public class IdentityUsersModel: PageModel {
        private UserManager<IdentityUser> userManager;

        public IdentityUsersModel(UserManager<IdentityUser> mgr) {
            userManager = mgr;
        }

        public IdentityUser AdminUser{ get; set; }

        public async Task OnGetAsync() {
            AdminUser = await userManager.FindByNameAsync("Admin");
        }
    }
}

```

Restart ASP.NET Core and request <http://localhost:5000/admin/identityusers> to see the content generated by the Razor Page, which is shown in Figure 11-1.

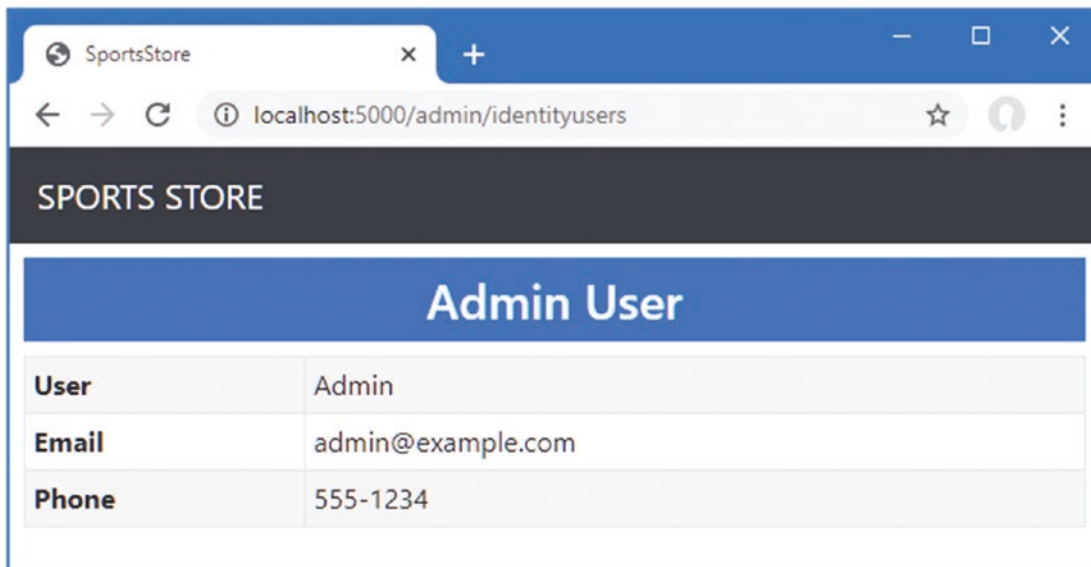


Figure 11-1. A Razor Page administration feature

Applying a Basic Authorization Policy

Now that I have configured ASP.NET Core Identity, I can apply an authorization policy to the parts of the application that I want to protect. I am going to use the most basic authorization policy possible, which is to allow access to any authenticated user. Although this can be a useful policy in real applications as well, there are also options for creating finer-grained authorization controls, as described in Chapters 37 and 38, but since the SportsStore application has only one user, distinguishing between anonymous and authenticated requests is sufficient.

For controllers and Razor pages, the `Authorize` attribute is used to restrict access, as shown in Listing 11-10.

Listing 11-10. Restricting Access in the IdentityUsers.cshtml File in the SportsStore/Pages/Admin Folder

```

@page
@model IdentityUsersModel
@using Microsoft.AspNetCore.Identity
@using Microsoft.AspNetCore.Authorization

<h3 class="bg-primary text-white text-center p-2">Admin User</h3>

<table class="table table-sm table-striped table-bordered">
  <tbody>
    <tr><th>User</th><td>@Model.AdminUser.UserName</td></tr>
    <tr><th>Email</th><td>@Model.AdminUser.Email</td></tr>
    <tr><th>Phone</th><td>@Model.AdminUser.PhoneNumber</td></tr>
  </tbody>
</table>

@functions{
    [Authorize]
    public class IdentityUsersModel: PageModel {
        private UserManager<IdentityUser> userManager;

        public IdentityUsersModel(UserManager<IdentityUser> mgr) {
            userManager = mgr;
        }

        public IdentityUser AdminUser{ get; set; }

        public async Task OnGetAsync() {
            AdminUser = await userManager.FindByNameAsync("Admin");
        }
    }
}

```

When there are only authorized and unauthorized users, the Authorize attribute can be applied to the Razor Page that acts as the entry point for the Blazor part of the application, as shown in Listing 11-11.

Listing 11-11. Applying Authorization in the Index.cshtml File in the SportsStore/Pages/Admin Folder

```

@page "/admin"
@{ Layout = null; }
@using Microsoft.AspNetCore.Authorization
@attribute [Authorize]

<!DOCTYPE html>
<html>
<head>
  <title>SportsStore Admin</title>
  <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
  <base href="/" />
</head>
<body>
  <component type="typeof(Routed)" render-mode="Server" />
  <script src="/_framework/blazor.server.js"></script>
</body>
</html>

```

Since this Razor Page has been configured with a page model class, I can apply the attribute with an @attribute expression.

Creating the Account Controller and Views

When an unauthenticated user sends a request that requires authorization, the user is redirected to the `/Account/Login` URL, which the application can use to prompt the user for their credentials. In Listing 11-12, I show you how to handle authentication using Razor Pages, so, for variety, I am going to use controllers and views for SportsStore. In preparation, I added a view model to represent the user's credentials by adding a class file called `LoginModel.cs` to the `Models/ViewModels` folder and using it to define the class shown in Listing 11-12.

Listing 11-12. The Contents of the `LoginModel.cs` File in the `SportsStore/Models/ViewModels` Folder

```
using System.ComponentModel.DataAnnotations;

namespace SportsStore.Models.ViewModels {

    public class LoginModel {

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

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

        public string returnUrl { get; set; } = "/";
    }
}
```

The `Name` and `Password` properties have been decorated with the `Required` attribute, which uses model validation to ensure that values have been provided. Next, I added a class file called `AccountController.cs` to the `Controllers` folder and used it to define the controller shown in Listing 11-13. This is the controller that will respond to requests to the `/Account/Login` URL.

Listing 11-13. The Contents of the `AccountController.cs` File in the `SportsStore/Controllers` Folder

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Mvc;
using SportsStore.Models.ViewModels;

namespace SportsStore.Controllers {

    public class AccountController : Controller {
        private UserManager<IdentityUser> userManager;
        private SignInManager<IdentityUser> signInManager;

        public AccountController(UserManager<IdentityUser> userMgr,
            SignInManager<IdentityUser> signInMgr) {
            userManager = userMgr;
            signInManager = signInMgr;
        }

        public IActionResult Login(string returnUrl) {
            return View(new LoginModel {
                returnUrl = returnUrl
            });
        }

        [HttpPost]
        [ValidateAntiForgeryToken]
    }
}
```

```

public async Task<IActionResult> Login(LoginModel loginModel) {
    if (ModelState.IsValid) {
        IdentityUser user =
            await userManager.FindByNameAsync(loginModel.Name);
        if (user != null) {
            await signInManager.SignOutAsync();
            if ((await signInManager.PasswordSignInAsync(user,
                loginModel.Password, false, false)).Succeeded) {
                return Redirect(loginModel?.ReturnUrl ?? "/Admin");
            }
        }
        ModelState.AddModelError("", "Invalid name or password");
        return View(loginModel);
    }

    [Authorize]
    public async Task<RedirectResult> Logout(string returnUrl = "/") {
        await signInManager.SignOutAsync();
        return Redirect(returnUrl);
    }
}
}

```

When the user is redirected to the /Account/Login URL, the GET version of the Login action method renders the default view for the page, providing a view model object that includes the URL that the browser should be redirected to if the authentication request is successful.

Authentication credentials are submitted to the POST version of the Login method, which uses the `UserManager<IdentityUser>` and `SignInManager<IdentityUser>` services that have been received through the controller's constructor to authenticate the user and log them into the system. I explain how these classes work in Chapters 37 and 38, but for now, it is enough to know that if there is an authentication failure, then I create a model validation error and render the default view; however, if authentication is successful, then I redirect the user to the URL that they want to access before they are prompted for their credentials.

■ **Caution** In general, using client-side data validation is a good idea. It offloads some of the work from your server and gives users immediate feedback about the data they are providing. However, you should not be tempted to perform authentication at the client, as this would typically involve sending valid credentials to the client so they can be used to check the username and password that the user has entered, or at least trusting the client's report of whether they have successfully authenticated. Authentication should always be done at the server.

To provide the Login method with a view to render, I created the Views/Account folder and added a Razor View file called `Login.cshtml` with the contents shown in Listing 11-14.

Listing 11-14. The Contents of the Login.cshtml File in the SportsStore/Views/Account Folder

```

@model LoginModel
@{ Layout = null; }
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>SportsStore</title>
    <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
</head>

```



```

<body>
  <div class="bg-dark text-white p-2">
    <span class="navbar-brand ml-2">SPORTS STORE</span>
  </div>
  <div class="m-1 p-1">
    <div class="text-danger asp-validation-summary="All"></div>

    <form asp-action="Login" asp-controller="Account" method="post">
      <input type="hidden" asp-for="ReturnUrl" />
      <div class="form-group">
        <label asp-for="Name"></label>
        <div asp-validation-for="Name" class="text-danger"></div>
        <input asp-for="Name" class="form-control" />
      </div>
      <div class="form-group">
        <label asp-for="Password"></label>
        <div asp-validation-for="Password" class="text-danger"></div>
        <input asp-for="Password" type="password" class="form-control" />
      </div>
      <button class="btn btn-primary" type="submit">Log In</button>
    </form>
  </div>
</body>
</html>

```

The final step is a change to the shared administration layout to add a button that will log the current user out by sending a request to the Logout action, as shown in Listing 11-15. This is a useful feature that makes it easier to test the application, without which you would need to clear the browser's cookies to return to the unauthenticated state.

Listing 11-15. Adding a Logout Button in the AdminLayout.razor File in the SportsStore/Pages/Admin Folder

```

@inherits LayoutComponentBase

<div class="bg-info text-white p-2">
  <div class="container-fluid">
    <div class="row">
      <div class="col">
        <span class="navbar-brand ml-2">SPORTS STORE Administration</span>
      </div>
      <div class="col-2 text-right">
        <a class="btn btn-sm btn-primary" href="/account/logout">Log Out</a>
      </div>
    </div>
  </div>
</div>
<div class="container-fluid">
  <div class="row p-2">
    <div class="col-3">
      <NavLink class="btn btn-outline-primary btn-block"
        href="/admin/products"
        ActiveClass="btn-primary text-white"
        Match="NavLinkMatch.Prefix">
        Products
      </NavLink>
      <NavLink class="btn btn-outline-primary btn-block"
        href="/admin/orders"
        ActiveClass="btn-primary text-white"
        Match="NavLinkMatch.Prefix">

```

```

        Orders
    </NavLink>
</div>
<div class="col">
    @Body
</div>
</div>
</div>

```

Testing the Security Policy

Everything is in place, and you can test the security policy by restarting ASP.NET Core and requesting `http://localhost:5000/admin` or `http://localhost:5000/admin/identityusers`.

Since you are presently unauthenticated and you are trying to target an action that requires authorization, your browser will be redirected to the `/Account/Login` URL. Enter **Admin** and **Secret123\$** as the name and password and submit the form. The Account controller will check the credentials you provided with the seed data added to the Identity database and—assuming you entered the right details—authenticate you and redirect you to the URL you requested, to which you now have access. Figure 11-2 illustrates the process.

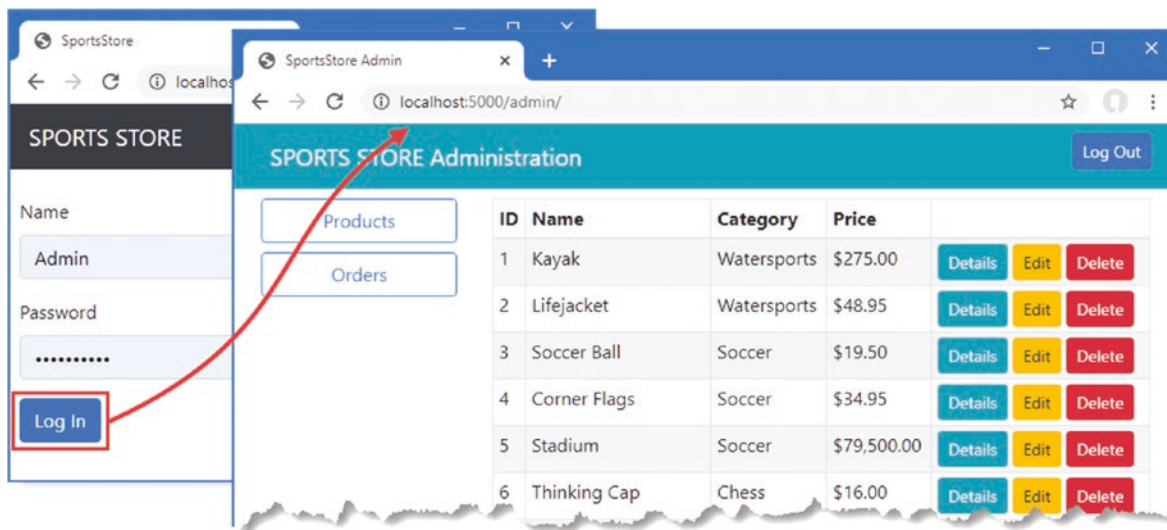


Figure 11-2. The administration authentication/authorization process

Preparing ASP.NET Core for Deployment

In this section, I will prepare SportsStore and create a container that can be deployed into production. There is a wide range of deployment models available for ASP.NET Core applications, but I have picked Docker containers because they can be run on most hosting platforms or be deployed into a private data center. This is not a complete guide to deployment, but it will give you a sense of the process to prepare an application.

Configuring Error Handling

At the moment, the application is configured to use the developer-friendly error pages, which provide helpful information when a problem occurs. This is not information that end users should see, so I added a Razor Page named `Error.cshtml` to the Pages folder with the content shown in Listing 11-16.

Listing 11-16. The Contents of the `Error.cshtml` File in the Pages Folder

```

@page "/error"
@{ Layout = null; }
<!DOCTYPE html>

```

```

<html>
<head>
  <meta name="viewport" content="width=device-width" />
  <link href="/lib/twitter-bootstrap/css/bootstrap.min.css" rel="stylesheet" />
  <title>Error</title>
</head>
<body class="text-center">
  <h2 class="text-danger">Error.</h2>
  <h3 class="text-danger">An error occurred while processing your request</h3>
</body>
</html>

```

This kind of error page is the last resort, and it is best to keep it as simple as possible and not to rely on shared views, view components, or other rich features. In this case, I have disabled shared layouts and defined a simple HTML document that explains that there has been an error, without providing any information about what has happened.

In Listing 11-17, I have reconfigured the application so that the Error page is used for unhandled exceptions when the application is in the production environment.

Listing 11-17. Configuring Error Handling in the Startup.cs File in the SportsStore Folder

```

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

    if (env.IsProduction()) {
        app.UseExceptionHandler("/error");
     } else {
        app.UseDeveloperExceptionPage();
        app.UseStatusCodePages();
     }

    app.UseStaticFiles();
    app.UseSession();

    app.UseRouting();

    app.UseAuthentication();
    app.UseAuthorization();

    app.UseEndpoints(endpoints => {
        endpoints.MapControllerRoute("catpage",
            "{category}/Page{productPage:int}",
            new { Controller = "Home", action = "Index" });

        endpoints.MapControllerRoute("page", "Page{productPage:int}",
            new { Controller = "Home", action = "Index", productPage = 1 });

        endpoints.MapControllerRoute("category", "{category}",
            new { Controller = "Home", action = "Index", productPage = 1 });

        endpoints.MapControllerRoute("pagination",
            "Products/Page{productPage}",
            new { Controller = "Home", action = "Index", productPage = 1 });
        endpoints.MapDefaultControllerRoute();
        endpoints.MapRazorPages();
        endpoints.MapBlazorHub();
        endpoints.MapFallbackToPage("/admin/{*catchall}", "/Admin/Index");
    });
}

```

```

    SeedData.EnsurePopulated(app);
    IdentitySeedData.EnsurePopulated(app);
}
...

```

As I explain in Chapter 12, the `IWebHostEnvironment` parameter defined by the `Configure` method describes the environment in which the application is running. The changes mean that the `UseExceptionHandler` method is called when the application is in production, but the developer-friendly error pages are used otherwise.

Creating the Production Configuration Settings

The JSON configuration files that are used to define settings such as connection strings can be created so they apply only when the application is in a specific environment, such as development, staging, or production. The template I used to create the `SportsStore` project in Chapter 7 created the `appsettings.json` and `appsettings.Development.json` files, which are intended to be the default settings that are overridden with those that are specific for development. I am going to take the reverse approach for this chapter and define a file that contains just those settings that are specific to production. Add a JSON file named `appsettings.Production.json` to the `SportsStore` folder with the content shown in Listing 11-18.

■ **Caution** Do not use these connection strings in real projects. You must correctly describe the connection to your production database, which is unlikely to be the same as the ones in the listing.

Listing 11-18. The Contents of the `appsettings.Production.json` File in the `SportsStore` Folder

```

{
  "ConnectionStrings": {
    "SportsStoreConnection": "Server=sqlserver;Database=SportsStore;MultipleActiveResultSets=true;User=sa;Password=MyDatabaseSecret123",
    "IdentityConnection": "Server=sqlserver;Database=Identity;MultipleActiveResultSets=true;User=sa;Password=MyDatabaseSecret123"
  }
}

```

These connection strings, each of which is defined on a single line, describe connections to SQL Server running on `sqlserver`, which is another Docker container running SQL Server.

Creating the Docker Image

In the sections that follow, I configure and create the Docker image for the application that can be deployed into a container environment such as Microsoft Azure or Amazon Web Services. Bear in mind that containers are only one style of deployment and there are many others available if this approach does not suit you.

■ **Note** Bear in mind that I am going to connect to a database running on the development machine, which is not how most real applications are configured. Be sure to configure the database connection strings and the container networking settings to match your production environment.

Installing Docker Desktop

Go to [Docker.com](https://www.docker.com) and download and install the Docker Desktop package. Follow the installation process, reboot your Windows machine, and run the command shown in Listing 11-19 to check that Docker has been installed and is in your path. (The Docker installation process seems to change often, which is why I have not been more specific about the process.)

■ **Note** You will have to create an account on Docker.com to download the installer.

Listing 11-19. Checking the Docker Desktop Installation

```
docker --version
```

Creating the Docker Configuration Files

Docker is configured using a file named `Dockerfile`. There is no Visual Studio item template for this file, so use the Text File template to add a file named `Dockerfile.text` to the project and then rename the file to `Dockerfile`. If you are using Visual Studio Code, you can just create a file named `Dockerfile` without the extension. Use the configuration settings shown in Listing 11-20 as the contents for the new file.

Listing 11-20. The Contents of the Dockerfile File in the SportsStore Folder

```
FROM mcr.microsoft.com/dotnet/core/aspnet:3.1
FROM mcr.microsoft.com/dotnet/core/sdk:3.1

COPY /bin/Release/netcoreapp3.1/publish/ SportsStore/

ENV ASPNETCORE_ENVIRONMENT Production

EXPOSE 5000
WORKDIR /SportsStore
ENTRYPOINT ["dotnet", "SportsStore.dll", "--urls=http://0.0.0.0:5000"]
```

These instructions copy the SportsStore application into a Docker image and configure its execution. Next, create a file called `docker-compose.yml`, with the content shown in Listing 11-21. Visual Studio doesn't have a template for this type of file, but if you select the Text File template and enter the complete filename, it will create the file. Visual Studio Code users can simply create a file named `docker-compose.yml`.

Listing 11-21. The Contents of the `docker-compose.yml` File in the SportsStore Folder

```
version: "3"
services:
  sportsstore:
    build: .
    ports:
      - "5000:5000"
    environment:
      - ASPNETCORE_ENVIRONMENT=Production
    depends_on:
      - sqlserver
  sqlserver:
    image: "mcr.microsoft.com/mssql/server"
    environment:
      SA_PASSWORD: "MyDatabaseSecret123"
      ACCEPT_EULA: "Y"
```

The YML files are especially sensitive to formatting and indentation, and it is important to create this file exactly as shown. If you have problems, then use the `docker-compose.yml` file from the GitHub repository for this book, <https://github.com/apress/pro-asp.net-core-3>.

Publishing and Imaging the Application

Prepare the SportsStore application by using a PowerShell prompt to run the command shown Listing 11-22 in the SportsStore folder.

Listing 11-22. Preparing the Application

```
dotnet publish -c Release
```

Next, run the command shown in Listing 11-23 to create the Docker image for the SportsStore application. This command will take some time to complete the first time it is run because it will download the Docker images for ASP.NET Core.

Listing 11-23. Performing the Docker Build

```
docker-compose build
```

The first time you run this command, you may be prompted to allow Docker to use the network, as shown in Figure 11-3.

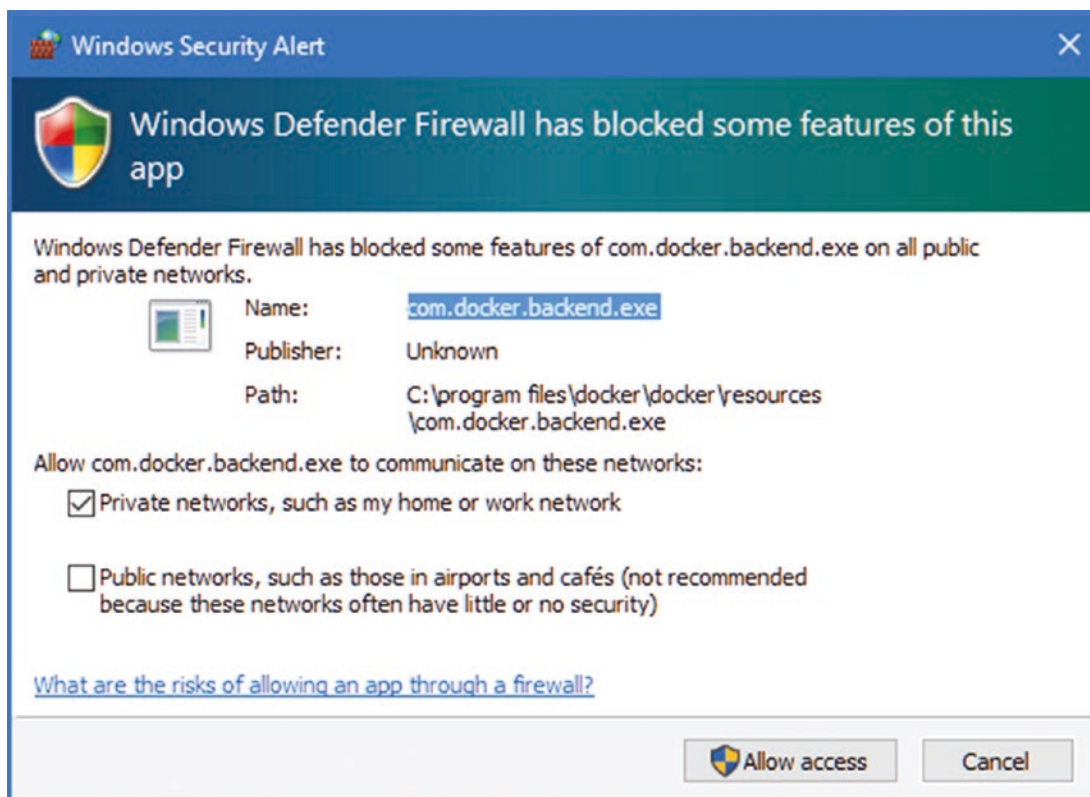


Figure 11-3. Granting network access

Click the Allow button, return to the PowerShell prompt, use Control+C to terminate the Docker containers, and run the command in Listing 11-23 again.

Running the Containerized Application

Run the command shown in Listing 11-24 in the SportsStore folder to start the Docker containers for the SportsStore application and SQL Server. This command will take some time to complete the first time it is run because it will download the Docker images for SQL Server.

Listing 11-24. Starting the Containers

```
docker-compose up
```

It can take some time for both containers to start. There is a lot of output, mostly from SQL Server, but the application will be ready when you see output like this:

```
...
sportsstore_1 | info: Microsoft.Hosting.Lifetime[0]
sportsstore_1 | Now listening on: http://0.0.0.0:5000
sportsstore_1 | info: Microsoft.Hosting.Lifetime[0]
sportsstore_1 | Application started. Press Ctrl+C to shut down.
sportsstore_1 | info: Microsoft.Hosting.Lifetime[0]
sportsstore_1 | Hosting environment: Production
sportsstore_1 | info: Microsoft.Hosting.Lifetime[0]
sportsstore_1 | Content root path: /SportsStore
...
```

Open a new browser window and request `http://localhost:5000`, and you will receive a response from the containerized version of SportsStore, as shown in Figure 11-4, which is now ready for deployment. Use Control+C at the PowerShell command prompt to terminate the Docker containers.

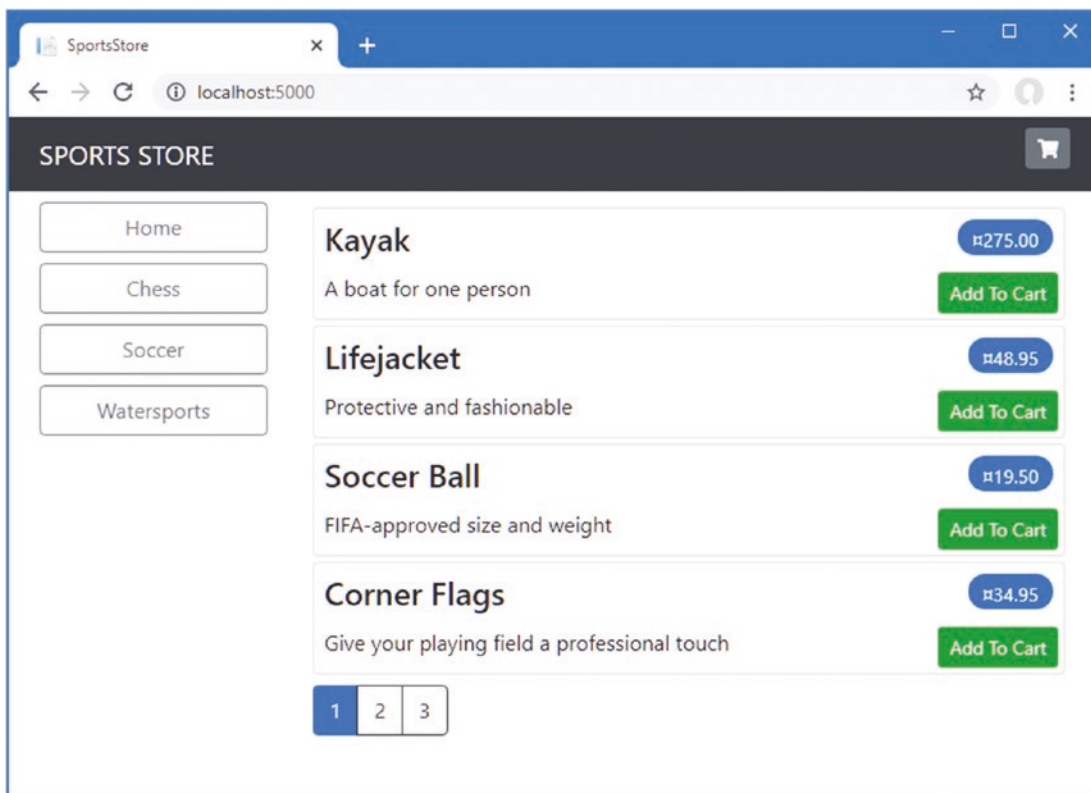


Figure 11-4. Running the SportsStore application in a container

Summary

In this and previous chapters, I demonstrated how the ASP.NET Core can be used to create a realistic e-commerce application. This extended example introduced many key features: controllers, action methods, views, Razor Pages, Blazor, routing, validation, authentication, and more. You also saw how some of the key technologies related to how ASP.NET Core can be used. These included the Entity Framework Core, ASP.NET Core Identity, and unit testing. And that's the end of the SportsStore application. In the next part of the book, I start to dig into the details of ASP.NET Core.

PART II



The ASP.NET Core Platform



Understanding the ASP.NET Core Platform

The ASP.NET Core platform is the foundation for creating web applications and provides the features that allow frameworks like MVC and Blazor to be used. In this chapter, I explain how the basic ASP.NET Core features work, describe the purpose of the files in an ASP.NET Core project, and explain how the ASP.NET Core request pipeline is used to process HTTP requests and demonstrate the different ways that it can be customized.

Don't worry if not everything in this chapter makes immediate sense or appears to apply to the applications you intend to create. The features I describe in this chapter are the underpinnings for everything that ASP.NET Core does, and understanding how they work helps provide a context for understanding the features that you will use on a daily basis, as well as giving you the knowledge you need to diagnose problems when you don't get the behavior you expect. Table 12-1 puts the ASP.NET Core platform in context.

Table 12-1. *Putting the ASP.NET Core Platform in Context*

Question	Answer
What is it?	The ASP.NET Core platform is the foundation on which web applications are built and provides features for processing HTTP requests.
Why is it useful?	The ASP.NET Core platform takes care of the low-level details of web applications so that developers can focus on features for the end user.
How is it used?	The key building blocks are services and middleware components, both of which can be created in the <code>Startup</code> class.
Are there any pitfalls or limitations?	The use of the <code>Startup</code> class can be confusing, and close attention must be paid to the order of the statements it contains.
Are there any alternatives?	The ASP.NET Core platform is required for ASP.NET Core applications, but you can choose not to work with the platform directly and rely on just the higher-level ASP.NET Core features, which are described in later chapters.

Table 12-2 summarizes the chapter.

Table 12-2. *Chapter Summary*

Problem	Solution	Listing
Creating a middleware component	Call the <code>Use</code> or <code>UseMiddleware</code> method to add a function or class to the request pipeline	4-6
Modifying a response	Write a middleware component that uses the return pipeline path	7
Preventing other components from processing a request	Short-circuit the request pipeline or create terminal middleware	8, 11, 12
Using different sets of middleware	Create a pipeline branch	9
Configuring middleware components	Use the options pattern	13-16

Preparing for This Chapter

To prepare for this chapter, I am going to create a new project named Platform, using the template that provides the minimal ASP.NET Core setup. Open a new PowerShell command prompt from the Windows Start menu and run the commands shown in Listing 12-1.

■ **Tip** You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/apress/pro-asp.net-core-3>. See Chapter 1 for how to get help if you have problems running the examples.

Listing 12-1. Creating the Project

```
dotnet new globaljson --sdk-version 3.1.101 --output Platform
dotnet new web --no-https --output Platform --framework netcoreapp3.1
dotnet new sln -o Platform

dotnet sln Platform add Platform
```

If you are using Visual Studio, open the Platform.sln file in the Platform folder. Select Project ► Platform Properties, navigate to the Debug page, and change the App URL field to **http://localhost:5000**, as shown in Figure 12-1. This changes the port that will be used to receive HTTP requests. Select File ► Save All to save the configuration changes.

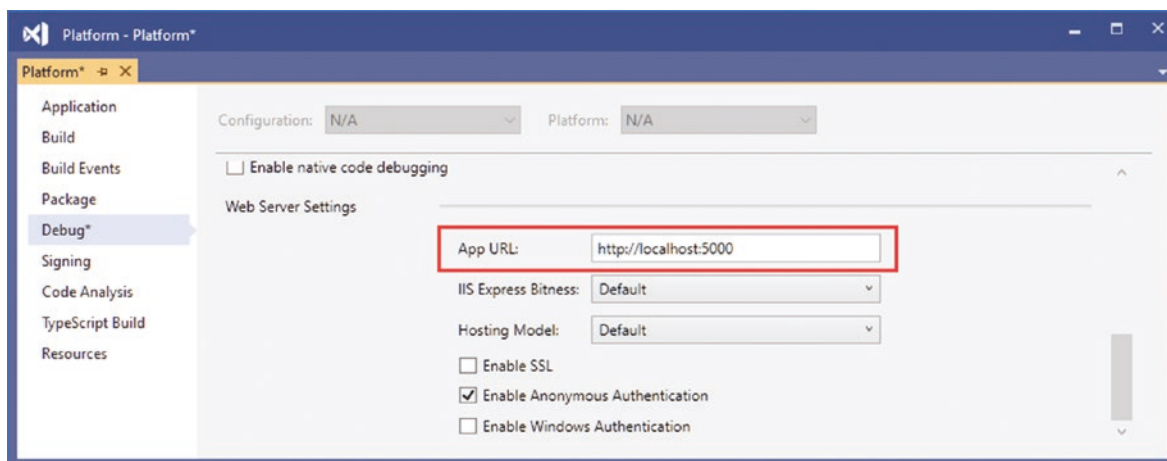


Figure 12-1. Changing the HTTP port

If you are using Visual Studio Code, open the Platform folder. Click the Yes button when prompted to add the assets required for building and debugging the project, as shown in Figure 12-2.

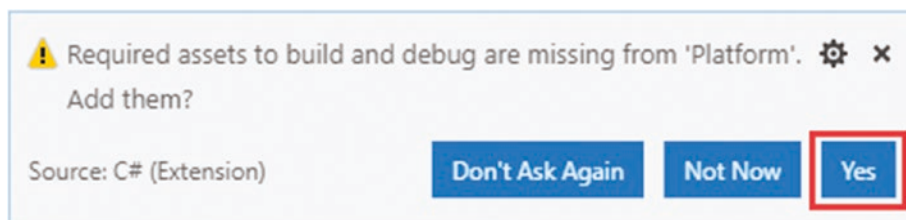


Figure 12-2. Adding project assets

Running the Example Application

Start the example application by selecting Start Without Debugging or Run Without Debugging from the Debug menu. If you are using Visual Studio Code, select .NET Core when prompted to select an environment. This is a selection that is made only when the project is first started. A new browser window will be opened, and you will see the output shown in Figure 12-3.

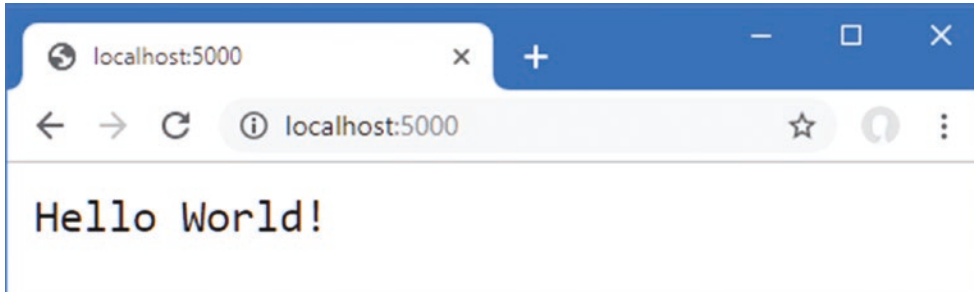


Figure 12-3. Running the example application

You can also start the application from the command line by running the command shown in Listing 12-2 in the Platform folder.

Listing 12-2. Starting the Example Application

```
dotnet run
```

This command doesn't open a new browser window, and you will have to manually navigate to `http://localhost:5000` to see the output shown in Figure 12-3.

Understanding the ASP.NET Core Platform

To understand ASP.NET Core, it is helpful to focus on just the key features: the request pipeline, middleware, and services. Understanding how these features fit together—even without going into detail—provides useful context for understanding the contents of the ASP.NET Core project and the shape of the ASP.NET Core platform.

Understanding Middleware and the Request Pipeline

The purpose of the ASP.NET Core platform is to receive HTTP requests and send responses to them, which ASP.NET Core delegates to *middleware components*. Middleware components are arranged in a chain, known as the *request pipeline*.

When a new HTTP request arrives, the ASP.NET Core platform creates an object that describes it and a corresponding object that describes the response that will be sent in return. These objects are passed to the first middleware component in the chain, which inspects the request and modifies the response. The request is then passed to the next middleware component in the chain, with each component inspecting the request and adding to the response. Once the request has made its way through the pipeline, the ASP.NET Core platform sends the response, as illustrated in Figure 12-4.

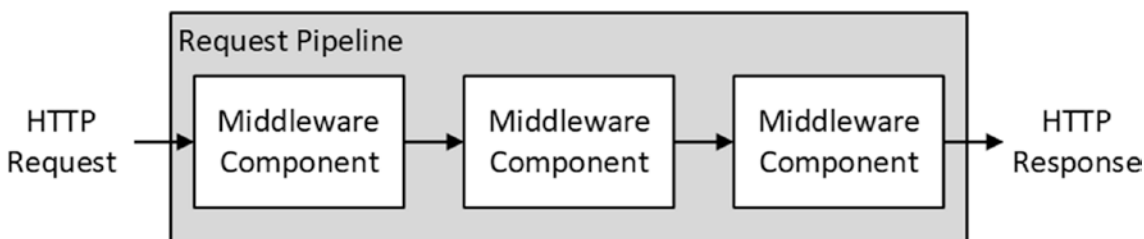


Figure 12-4. The ASP.NET Core request pipeline

Some components focus on generating responses for requests, but others are there to provide supporting features, such as formatting specific data types or reading and writing cookies. ASP.NET Core includes middleware components that solve common problems, as described in Chapters 15 and 16, and I show how to create custom middleware components later in this chapter. If no response is generated by the middleware components, then ASP.NET Core will return a response with the HTTP 404 Not Found status code.

Understanding Services

Services are objects that provide features in a web application. Any class can be used as a service, and there are no restrictions on the features that services provide. What makes services special is that they are managed by ASP.NET Core, and a feature called *dependency injection* makes it possible to easily access services anywhere in the application, including middleware components.

Dependency injection can be a difficult topic to understand, and I describe it in detail in Chapter 14. For now, it is enough to know that there are objects that are managed by the ASP.NET Core platform that can be shared by middleware components, either to coordinate between components or to avoid duplicating common features, such as logging or loading configuration data, as shown in Figure 12-5.

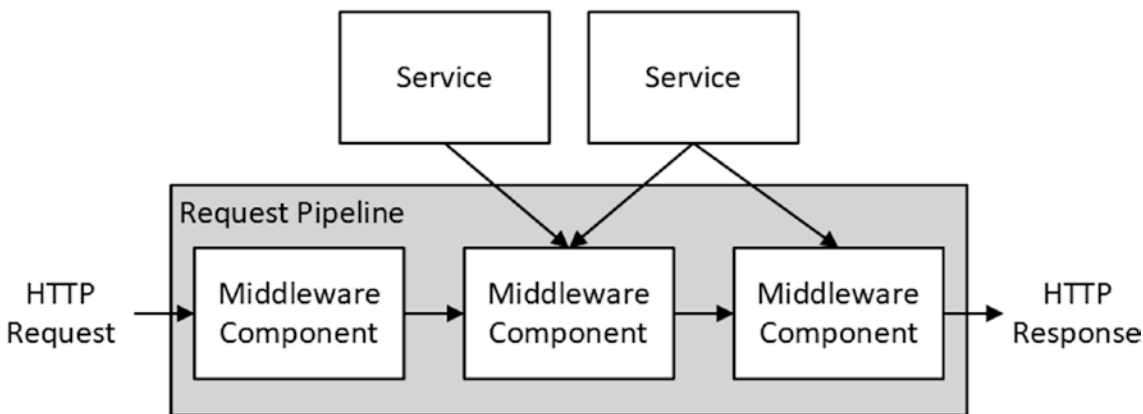


Figure 12-5. Services in the ASP.NET Core platform

As the figure shows, middleware components use only the services they require to do their work. As you will learn in later chapters, ASP.NET Core provides some basic services that can be supplemented by additional services that are specific to an application.

Understanding the ASP.NET Core Project

The Empty template produces a project with just enough code and configuration to start the ASP.NET Core runtime with some basic services and middleware components. Figure 12-6 shows the files added to the project by the template.

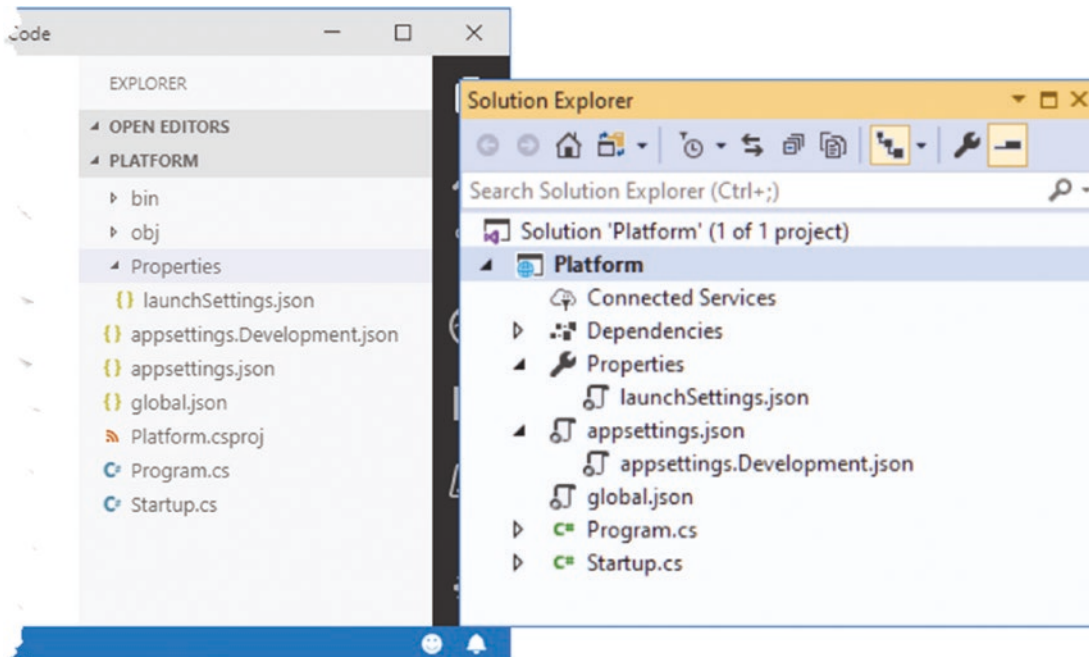


Figure 12-6. The files in the example project

Visual Studio and Visual Studio Code take different approaches to displaying files and folders. Visual Studio hides items that are not commonly used by the development and nests related items together, while Visual Studio Code shows everything.

This is why the two project views shown in the figure are different: Visual Studio has hidden the `bin` and `obj` folders and nested the `appsettings.Development.json` file within the `appsettings.json` file. The buttons at the top of the Solution Explorer window can be used to prevent nesting and to show all of the files in the project.

Although there are few files in the project, they underpin ASP.NET Core development and are described in Table 12-3.

Table 12-3. The Files and Folders in the Example Project

Name	Description
<code>appsettings.json</code>	This file is used to configure the application, as described in Chapter 15.
<code>appsettings.Development.json</code>	This file is used to define configuration settings that are specific to development, as explained in Chapter 15.
<code>bin</code>	This folder contains the compiled application files. Visual Studio hides this folder.
<code>global.json</code>	This file is used to select a specific version of the .NET Core SDK.
<code>Properties/launchSettings.json</code>	This file is used to configure the application when it starts. Visual Studio hides this folder and file.
<code>obj</code>	This folder contains the intermediate output from the compiler. Visual Studio hides this folder.
<code>Platform.csproj</code>	This file describes the project to the .NET Core tools, including the package dependencies and build instructions, as described in the “Understanding the Project File” section. Visual Studio hides this file, but it can be edited by right-clicking the project item in the Solution Explorer and selecting Edit Project File from the pop-up menu.
<code>Program.cs</code>	This file is the entry point for the ASP.NET Core platform.
<code>Startup.cs</code>	This file is used to configure the ASP.NET Core runtime and its associated frameworks, as described in the “Understanding the Startup Class” section.

Understanding the Entry Point

.NET Core applications define a main method, which is invoked when the application is executed and which is known as the application's *entry point*. For ASP.NET Core, the main method is defined by the `Program` class in the `Program.cs` file. Here is the content of the `Program.cs` in the example project:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;

namespace Platform {
    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>();
                });
    }
}
```

The .NET Core runtime invokes the `Main` method, which calls the `CreateHostBuilder` method. The first step in the setup process is the call to the `Host.CreateDefaultBuilder` method.

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

This method is responsible for setting up the basic features of the ASP.NET Core platform, including creating services responsible for configuration data and logging, both of which are described in Chapter 15. This method also sets up the HTTP server, named Kestrel, that is used to receive HTTP requests and adds support for working with Internet Information Services (IIS).

The result from the `CreateDefaultBuilder` method is passed to the `ConfigureWebHostDefaults` method, which selects the `Startup` class as the next step in the startup process.

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

The statements added to the `Program` class by the `Empty` template are suitable for most ASP.NET Core projects, but you can change them if an application has specific requirements.

Understanding the Startup Class

The Startup class is where most of the setup required by an application is performed, using the two methods that are added to the class when the project is created.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

namespace Platform {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {

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

            app.UseRouting();

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

The `ConfigureServices` method is used to define the services that the application requires. By default, only the services created by the Program class are available, but you will see examples of different types of service used in later chapters, and [Chapter 14](#) describes in detail the mechanism for using services.

The `Configure` method is used to register the middleware components for the request pipeline. Three middleware components are added to the pipeline by default when a project is created using the Empty template, each of which is described in [Table 12-4](#).

Table 12-4. The Middleware Added to the Pipeline by the Empty Template

Middleware Method	Description
<code>UseDeveloperExceptionPage</code>	This method adds a middleware component that includes details of unhandled exceptions. The <code>IsDevelopment</code> method is used to ensure this information isn't presented to users, as described in Chapter 16 .
<code>UseRouting</code>	This method adds the endpoint routing middleware component to the pipeline, which is responsible for determining how some requests are handled and is used with other parts of ASP.NET Core such as the MVC Framework. Routing is described in detail in Chapter 13 .
<code>UseEndpoints</code>	This method provides the configuration for the endpoint routing middleware added by the <code>UseRouting</code> method. The configuration is described in Chapter 13 .

Microsoft provides other middleware as part of ASP.NET Core that deals with the most common features required by web applications, which I describe in Chapters 15 and 16. You can also create your own middleware, as described in the “Creating Custom Middleware” section, when the built-in features don’t suit your requirements.

Understanding the Project File

The Platform.csproj file, known as the *project file*, contains the information that .NET Core uses to build the project and keep track of dependencies. Here is the content that was added to the file by the Empty template when the project was created:

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

The csproj file is hidden when using Visual Studio; you can edit it by right-clicking the Platform project item in the Solution Explorer and selecting Edit Project File from the popup menu.

The project file contains XML elements that describe the project to MSBuild, the Microsoft build engine. MSBuild can be used to create complex build processes and is described in detail at <https://docs.microsoft.com/en-gb/visualstudio/msbuild/msbuild>.

There is no need to edit the project file directly in most projects. The most common change to the file is to add dependencies on other .NET packages, but these are typically added using the command-line tools or through the interface provided by Visual Studio.

To add a package to the project using the command line, open a new PowerShell command prompt, navigate to the Platform project folder (the one that contains the csproj file), and run the command shown in Listing 12-3.

Listing 12-3. Adding a Package to the Project

```
dotnet add package Swashbuckle.AspNetCore --version 5.0.0-rc2
```

This command adds the Swashbuckle.AspNetCore package to the project. You will see this package used in Chapter 20, but for now, it is the effect of the dotnet add package command that is important.

If you are using Visual Studio, you can add the package by right-clicking the Platform item in the Solution Explorer and selecting Manage NuGet packages from the popup menu. Click Browse and enter **Swashbuckle.AspNetCore** into the search text box. Select the Swashbuckle.AspNetCore package from the list, choose the 5.0.0-rc2 version from the Version drop-down list, and click the Install button, as shown in Figure 12-7. You will be prompted to accept the license of the package and its dependencies.

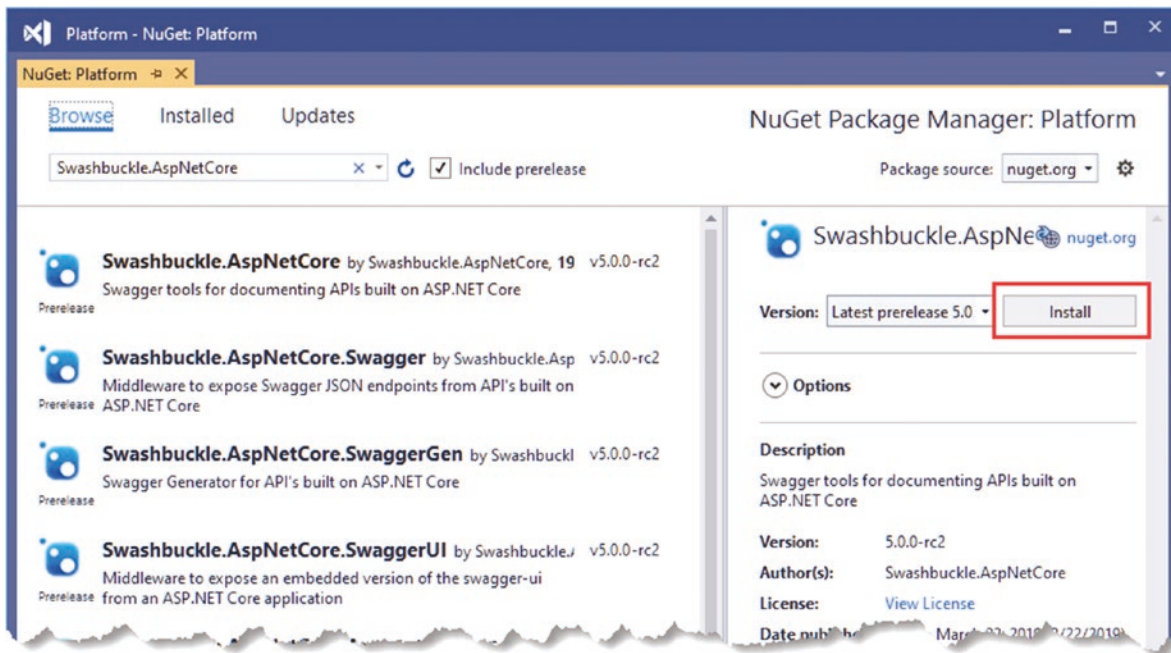


Figure 12-7. Installing a package in Visual Studio

The new dependency will be shown in the Platform.csproj file, regardless of which approach you use to install the package.

```
<Project Sdk="Microsoft.NET.Sdk.Web">
  <PropertyGroup>
    <TargetFramework>netcoreapp3.1</TargetFramework>
  </PropertyGroup>
  <ItemGroup>
    <PackageReference Include="Swashbuckle.AspNetCore" Version="5.0.0-rc2" />
  </ItemGroup>
</Project>
```

Creating Custom Middleware

As mentioned, Microsoft provides various middleware components for ASP.NET Core that handle the features most commonly required by web applications. You can also create your own middleware, which is a useful way to understand how ASP.NET Core works, even if you use only the standard components in your projects. The key method for creating middleware is `Use`, as shown in Listing 12-4.

Listing 12-4. Creating Custom Middleware in the Startup.cs File in the Platform Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
```

```

namespace Platform {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {

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

            app.Use(async (context, next) => {
                if (context.Request.Method == HttpMethod.Get
                    && context.Request.Query["custom"] == "true") {
                    await context.Response.WriteAsync("Custom Middleware \n");
                }
                await next();
            });

            app.UseRouting();

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

```

The `Use` method registers a middleware component that is typically expressed as a lambda function that receives each request as it passes through the pipeline (there is another method used for classes, as described in the next section).

The arguments to the lambda function are an `HttpContext` object and a function that is invoked to tell ASP.NET Core to pass the request to the next middleware component in the pipeline.

The `HttpContext` object describes the HTTP request and the HTTP response and provides additional context, including details of the user associated with the request. Table 12-5 describes the most useful members provided by the `HttpContext` class, which is defined in the `Microsoft.AspNetCore.Http` namespace.

Table 12-5. Useful `HttpContext` Members

Name	Description
Connection	This property returns a <code>ConnectionInfo</code> object that provides information about the network connection underlying the HTTP request, including details of local and remote IP addresses and ports.
Request	This property returns an <code>HttpRequest</code> object that describes the HTTP request being processed.
RequestServices	This property provides access to the services available for the request, as described in Chapter 14.
Response	This property returns an <code>HttpResponse</code> object that is used to create a response to the HTTP request.
Session	This property returns the session data associated with the request. The session data feature is described in Chapter 16.
User	This property returns details of the user associated with the request, as described in Chapters 37 and 38.
Features	This property provides access to request features, which allow access to the low-level aspects of request handling. See Chapter 16 for an example of using a request feature.

The ASP.NET Core platform is responsible for processing the HTTP request to create the `HttpRequest` object, which means that middleware and endpoints don't have to worry about the raw request data. Table 12-6 describes the most useful members of the `HttpRequest` class.

Table 12-6. *Useful HttpRequest Members*

Name	Description
Body	This property returns a stream that can be used to read the request body.
ContentLength	This property returns the value of the Content-Length header.
ContentType	This property returns the value of the Content-Type header.
Cookies	This property returns the request cookies.
Form	This property returns a representation of the request body as a form.
Headers	This property returns the request headers.
IsHttps	This property returns true if the request was made using HTTPS.
Method	This property returns the HTTP verb used for the request.
Path	This property returns the path section of the request URL.
Query	This property returns the query string section of the request URL as key/value pairs.

The `HttpResponse` object describes the HTTP response that will be sent back to the client when the request has made its way through the pipeline. Table 12-7 describes the most useful members of the `HttpResponse` class. The ASP.NET Core platform makes dealing with responses as easy as possible, sets headers automatically, and makes it easy to send content to the client.

Table 12-7. *Useful HttpResponse Members*

Name	Description
ContentLength	This property sets the value of the Content-Length header.
ContentType	This property sets the value of the Content-Type header.
Cookies	This property allows cookies to be associated with the request.
HasStarted	This property returns true if ASP.NET Core has started to send the response headers to the client, after which it is not possible to make changes.
Headers	This property allows the response headers to be set.
StatusCode	This property sets the status code for the response.
WriteAsync(data)	This asynchronous method writes a data string to the response body.
Redirect(url)	This method sends a redirection response.

When creating custom middleware, the `HttpContext`, `HttpRequest`, and `HttpResponse` objects are used directly, but, as you will learn in later chapters, this isn't usually required when using the higher-level ASP.NET Core features such as the MVC Framework and Razor Pages.

The middleware function I defined in Listing 12-4 uses the `HttpRequest` object to check the HTTP method and query string to identify GET requests that have a `custom` parameter in the query string whose value is `true`, like this:

```
...
if (context.Request.Method == HttpMethod.Get
    && context.Request.Query["custom"] == "true") {
...

```

The `HttpMethods` class defines static strings for each HTTP method. For GET requests with the expected query string, the middleware function uses the `WriteAsync` method to add a string to the body of the response.

```
...
await context.Response.WriteAsync("Custom Middleware \n");
...
```

■ **Note** In this part of the book, all the examples send simple string results to the browser. In Part 3, I show you how to create web services that return JSON data and introduce the different ways that ASP.NET Core can produce HTML results.

The second argument to the middleware is the function conventionally named `next` that tells ASP.NET Core to pass the request to the next component in the request pipeline.

```
...
if (context.Request.Method == HttpMethods.Get
    && context.Request.Query["custom"] == "true") {
    await context.Response.WriteAsync("Custom Middleware \n");
}
await next();
...
```

No arguments are required when invoking the next middleware component because ASP.NET Core takes care of providing the component with the `HttpContext` object and its own `next` function so that it can process the request. The `next` function is asynchronous, which is why the `await` keyword is used and why the lambda function is defined with the `async` keyword.

■ **Tip** You may encounter middleware that calls `next.Invoke()` instead of `next()`. These are equivalent, and `next()` is provided as a convenience by the compiler to produce concise code.

Restart ASP.NET Core and navigate to `http://localhost:5000/?custom=true`. You will see that the new middleware appends its message to the response body before passing on the request to the next middleware component, as shown in Figure 12-8. Remove the query string, or change `true` to `false`, and the middleware component will pass on the request without adding to the response.

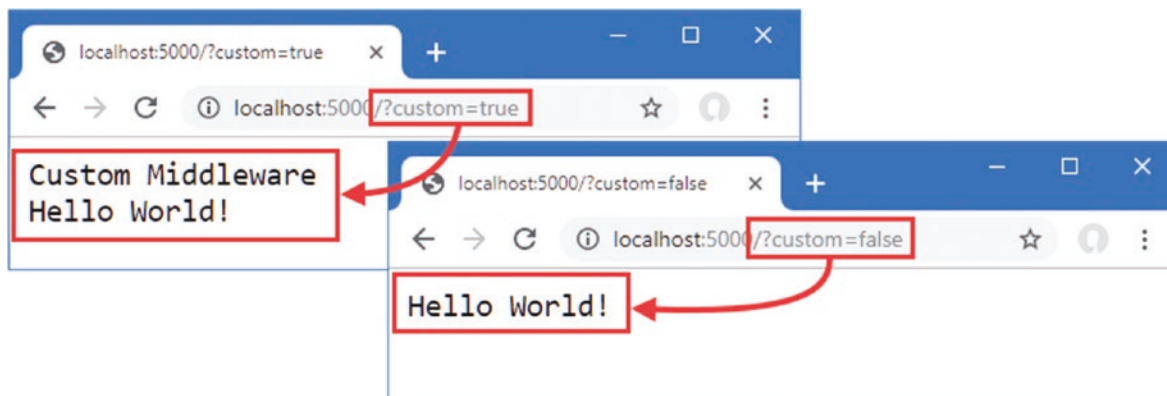


Figure 12-8. Creating custom middleware

Defining Middleware Using a Class

Defining middleware using lambda functions is convenient, but it can lead to a long and complex `Configure` method in the `Startup` class and makes it hard to reuse middleware in different projects. Middleware can also be defined using classes. Add a class file named `Middleware.cs` to the `Platform` folder and add the code shown in Listing 12-5.

Listing 12-5. The Contents of the `Middleware.cs` File in the `Platform` Folder

```
using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;

namespace Platform {

    public class QueryStringMiddleware {
        private RequestDelegate next;

        public QueryStringMiddleware(RequestDelegate nextDelegate) {
            next = nextDelegate;
        }

        public async Task Invoke(HttpContext context) {
            if (context.Request.Method == HttpMethod.Get
                && context.Request.Query["custom"] == "true") {
                await context.Response.WriteAsync("Class-based Middleware \n");
            }
            await next(context);
        }
    }
}
```

Middleware classes receive a `RequestDelegate` as a constructor parameter, which is used to forward the request to the next component in the pipeline. The `Invoke` method is called by ASP.NET Core when a request is received and receives an `HttpContext` object that provides access to the request and response, using the same classes that lambda function middleware receives. The `RequestDelegate` returns a `Task`, which allows it to work asynchronously.

One important difference in class-based middleware is that the `HttpContext` object must be used as an argument when invoking the `RequestDelete` to forward the request, like this:

```
...
await next(context);
...
```

Class-based middleware components are added to the pipeline with the `UseMiddleware` method, which accepts the middleware as a type argument, as shown in Listing 12-6.

Listing 12-6. Adding a Class-Based Middleware Component in the `Startup.cs` File in the `Platform` Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
```

```

namespace Platform {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {

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

            app.Use(async (context, next) => {
                if (context.Request.Method == HttpMethod.Get
                    && context.Request.Query["custom"] == "true") {
                    await context.Response.WriteAsync("Custom Middleware \n");
                }
                await next();
            });

            app.UseMiddleware<QueryStringMiddleware>();

            app.UseRouting();

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

```

When the ASP.NET Core is started, the `QueryStringMiddleware` class will be instantiated, and its `Invoke` method will be called to process requests as they are received.

■ **Caution** A single middleware object is used to handle all requests, which means that the code in the `Invoke` method must be thread-safe.

Select `Start Without Debugging` from the `Debug` menu or restart ASP.NET Core using the `dotnet run` command. Navigate to `http://localhost:5000/?custom=true`, and you will see the output from both middleware components, as shown in Figure 12-9.

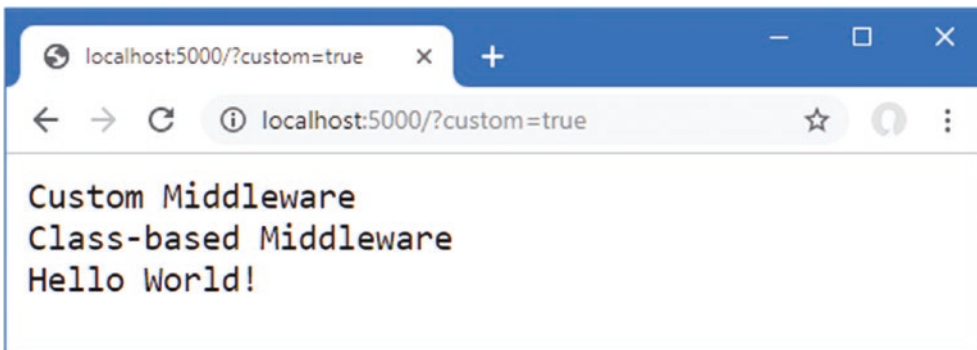


Figure 12-9. Using a class-based middleware component

Understanding the Return Pipeline Path

Middleware components can modify the `HttpResponse` object after the next function has been called, as shown by the new middleware in Listing 12-7.

Listing 12-7. Adding New Middleware in the Startup.cs File in the Platform Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

namespace Platform {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {

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

            app.Use(async (context, next) => {
                await next();
                await context.Response
                    .WriteAsync($"{\nStatus Code: { context.Response.StatusCode}");
            });

            app.Use(async (context, next) => {
                if (context.Request.Method == HttpMethod.Get
                    && context.Request.Query["custom"] == "true") {
                    await context.Response.WriteAsync("Custom Middleware \n");
                }
                await next();
            });

            app.UseMiddleware<QueryStringMiddleware>();

            app.UseRouting();

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


The new middleware immediately calls the next method to pass the request along the pipeline and then uses the `WriteAsync` method to add a string to the response body. This may seem like an odd approach, but it allows middleware to make changes to the response before and after it is passed along the request pipeline by defining statements before and after the next function is invoked, as illustrated by Figure 12-10.

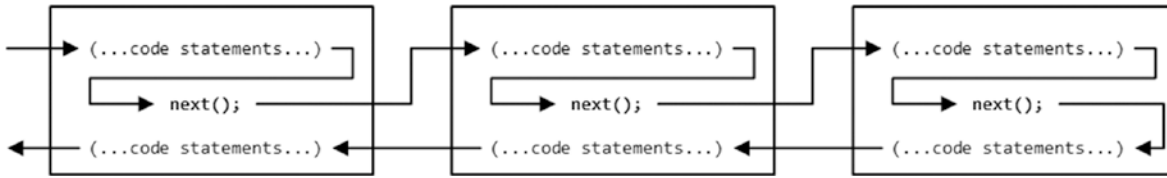


Figure 12-10. Passing request and responses through the ASP.NET Core pipeline

Middleware can operate before the request is passed on, after the request has been processed by other components, or both. The result is that several middleware components collectively contribute to the response that is produced, each providing some aspect of the response or providing some feature or data that is used later in the pipeline.

Select `Start Without Debugging` from the `Debug` menu or use `dotnet` to see the effect of the middleware defined in Listing 12-7, as shown in Figure 12-11. If you are using the command line, start ASP.NET Core using the `dotnet run` command and navigate to `http://localhost:5000`.

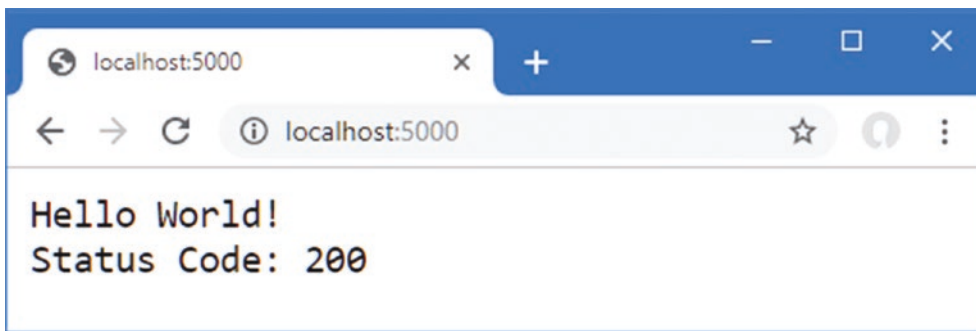


Figure 12-11. Modifying a response in the return path

■ **Note** Middleware components must not make changes to the `HttpResponse` object after ASP.NET Core has started to send the response to the client. The `HasStarted` property, described in Table 12-7, can be checked to avoid exceptions.

Short-Circuiting the Request Pipeline

Components that generate complete responses can choose not to call the next function so that the request isn't passed on. Components that don't pass on requests are said to *short-circuit* the pipeline, which is what the new middleware component shown in Listing 12-8 does for requests that target the `/short` URL.

Listing 12-8. Short-Circuiting the Request Pipeline in the `Startup.cs` File in the Platform Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
```

```

using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

namespace Platform {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {

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

            app.Use(async (context, next) => {
                await next();
                await context.Response
                    .WriteAsync($"Status Code: { context.Response.StatusCode}");
            });

            app.Use(async (context, next) => {
                if (context.Request.Path == "/short") {
                    await context.Response
                    .WriteAsync("Request Short Circuited");
                else {
                    await next();
                }
            });

            app.Use(async (context, next) => {
                if (context.Request.Method == HttpMethod.Get
                    && context.Request.Query["custom"] == "true") {
                    await context.Response.WriteAsync("Custom Middleware \n");
                }
                await next();
            });

            app.UseMiddleware<QueryStringMiddleWare>();

            app.UseRouting();

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

```

The new middleware checks the Path property of the HttpRequest object to see whether the request is for the /short URL; if it is, it calls the WriteAsync method without calling the next function. To see the effect, restart ASP.NET Core and navigate to the `http://localhost:5000/short?custom=true` URL, which will produce the output shown in Figure 12-12.

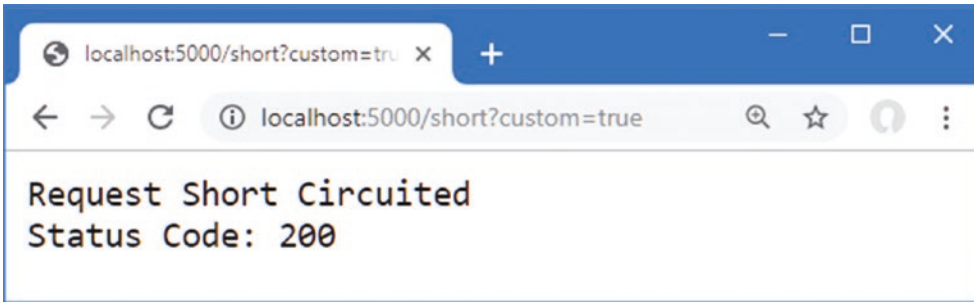


Figure 12-12. Short-circuiting the request pipeline

Even though the URL has the query string parameter that is expected by the next component in the pipeline, the request isn't forwarded, so that middleware doesn't get used. Notice, however, that the previous component in the pipeline has added its message to the response. That's because the short-circuiting only prevents components further along the pipeline from being used and doesn't affect earlier components, as illustrated in Figure 12-13.

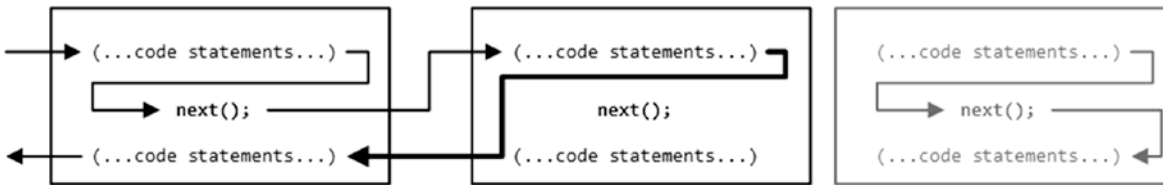


Figure 12-13. Short-circuiting the request pipeline

Creating Pipeline Branches

The Map method is used to create a section of pipeline that is used to process requests for specific URLs, creating a separate sequence of middleware components, as shown in Listing 12-9.

Listing 12-9. Creating a Pipeline Branch

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

namespace Platform {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {

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

```

app.Map("/branch", branch => {
    branch.UseMiddleware<QueryStringMiddleware>();

    branch.Use(async (context, next) => {
        await context.Response.WriteAsync($"Branch Middleware");
    });
});

app.UseMiddleware<QueryStringMiddleware>();

app.UseRouting();

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

```

The first argument to the Map method specifies the string that will be used to match URLs. The second argument is the branch of the pipeline, to which middleware components are added with the Use and UseMiddleware methods. The statements in Listing 12-9 create a branch that is used for URLs that start with /branch and that pass requests through the QueryStringMiddleware class defined in Listing 12-9, and the statements define a middleware lambda expression that adds a message to the response. Figure 12-14 shows the effect of the branch on the request pipeline.

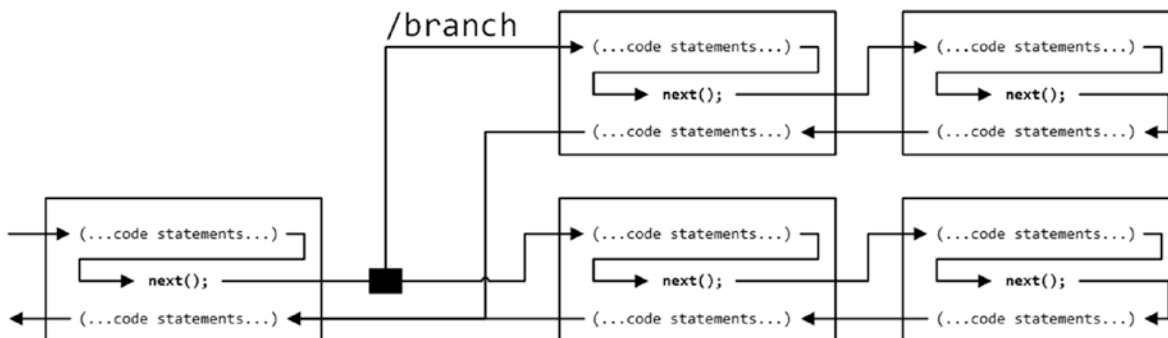


Figure 12-14. Adding a branch to the request pipeline

When a URL is matched by the Map method, it follows the branch and no longer passes through the middleware components on the main path through the pipeline. The same middleware can be used in different parts of the pipeline, which can be seen in Listing 12-9, where the QueryStringMiddleware class is used in both the main part of the pipeline and the branch.

To see the different ways that requests are handled, restart ASP.NET Core and request the `http://localhost:5000?custom=true` URL, which will be handled on the main part of the pipeline and will produce the output shown on the left of Figure 12-15. Navigate to `http://localhost:5000/branch?custom=true`, and the request will be forwarded to the middleware in the branch, producing the output shown on the right in Figure 12-15.

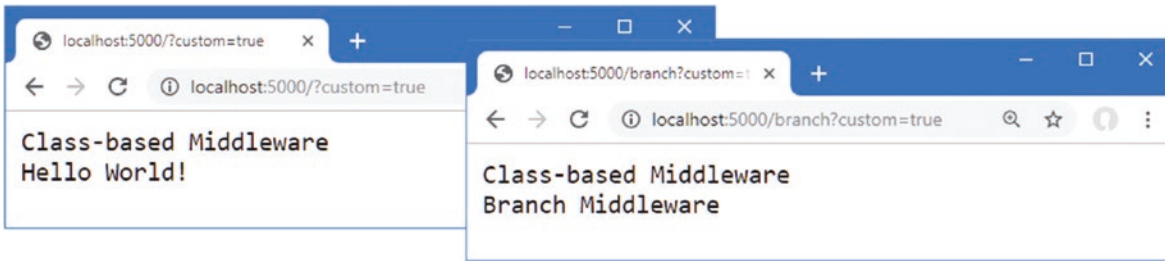


Figure 12-15. The effect of branching the request pipeline

BRANCHING WITH A PREDICATE

ASP.NET Core also supports the `MapWhen` method, which can be used to match requests using a predicate, allowing requests to be selected for a pipeline branch on criteria other than just URLs.

The arguments to the `MapWhen` method are a predicate function that receives an `HttpContext` and that returns `true` for requests that should follow the branch, and a function that receives an `IApplicationBuilder` object representing the pipeline branch, to which middleware is added. Here is an example of using the `MapWhen` method to branch the pipeline:

```
...
app.MapWhen(context => context.Request.Query.Keys.Contains("branch"),
    branch => {
        // ...add middleware components here...
    });
...
```

The predicate function returns `true` to branch for requests whose query string contains a parameter named `branch`.

Creating Terminal Middleware

Terminal middleware never forwards requests to other components and always marks the end of the request pipeline. There is a terminal middleware component in the `Startup` class, as shown here:

```
...
branch.Use(async (context, next) => {
    await context.Response.WriteAsync($"Branch Middleware");
});
...
```

ASP.NET Core supports the `Run` method as a convenience feature for creating terminal middleware, which makes it obvious that a middleware component won't forward requests and that a deliberate decision has been made not to call the next function. In Listing 12-10, I have used the `Run` method for the terminal middleware in the pipeline branch.

Listing 12-10. Using the `Run` Method in the `Startup.cs` File in the Platform Folder

```
...
app.Map("/branch", branch => {

    branch.UseMiddleware<QueryStringMiddleware>();

    branch.Run(async (context) => {
        await context.Response.WriteAsync($"Branch Middleware");
    });
});
...
```

The middleware function passed to the Run method receives only an `HttpContext` object and doesn't have to define a parameter that isn't used. Behind the scenes, the Run method is implemented through the Use method, and this feature is provided only as a convenience.

■ **Caution** Middleware added to the pipeline after a terminal component will never receive requests. ASP.NET Core won't warn you if you add a terminal component before the end of the pipeline.

Class-based components can be written so they can be used as both regular and terminal middleware, as shown in Listing 12-11.

Listing 12-11. Adding Terminal Support in the Middleware.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;

namespace Platform {

    public class QueryStringMiddleware {
        private RequestDelegate next;

        public QueryStringMiddleware() {
            // do nothing
        }

        public QueryStringMiddleware(RequestDelegate nextDelegate) {
            next = nextDelegate;
        }

        public async Task Invoke(HttpContext context) {
            if (context.Request.Method == HttpMethod.Get
                && context.Request.Query["custom"] == "true") {
                await context.Response.WriteAsync("Class-based Middleware \n");
            }
            if (next != null) {
                await next(context);
            }
        }
    }
}
```

The component will forward requests only when the constructor has been provided with a non-null value for the `nextDelegate` parameter. Listing 12-12 shows the application of the component in both standard and terminal forms.

Listing 12-12. Applying Class-Based Middleware in the Startup.cs File in the Platform Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
```

```

namespace Platform {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {

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

            app.Map("/branch", branch => {
                branch.Run(new QueryStringMiddleware().Invoke);
            });

            app.UseMiddleware<QueryStringMiddleware>();

            app.UseRouting();

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

```

There is no equivalent to the `UseMiddleware` method for terminal middleware, so the `Run` method must be used by creating a new instance of the middleware class and selecting its `Invoke` method. Using the `Run` method doesn't alter the output from the middleware, which you can see by restarting ASP.NET Core and navigating to the `http://localhost:5000/branch?custom=true` URL, which produces the content shown in Figure 12-16.

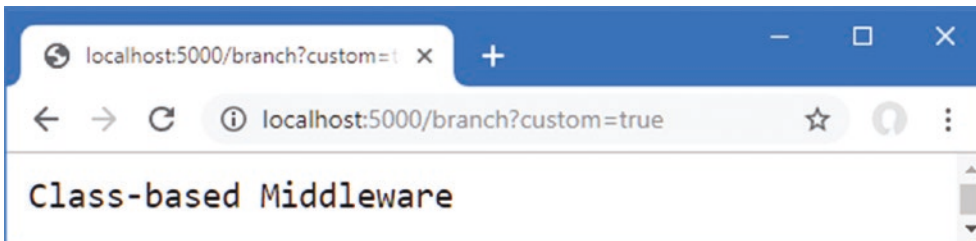


Figure 12-16. Using the `Run` method to create terminal middleware

Configuring Middleware

There is a common pattern for configuring middleware that is known as the *options pattern* and that is used by some of the built-in middleware components described in later chapters.

The starting point is to define a class that contains the configuration options for a middleware component. Add a class file named `MessageOptions.cs` to the `Platform` folder with the code shown in Listing 12-13.

Listing 12-13. The Contents of the MessageOptions.cs File in the Platform Folder

```
namespace Platform {

    public class MessageOptions {

        public string CityName { get; set; } = "New York";
        public string CountryName { get; set; } = "USA";
    }
}
```

The MessageOptions class defines properties that detail a city and a country. In Listing 12-14, I have used the options pattern to create a custom middleware component that relies on the MessageOptions class for its configuration. I have also removed some of the middleware from previous examples for brevity.

Listing 12-14. Using the Options Pattern in the Startup.cs File in the Platform Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Options;

namespace Platform {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.Configure<MessageOptions>(options => {
                options.CityName= "Albany";
            });
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env,
            IOptions<MessageOptions> msgOptions) {

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

            app.Use(async (context, next) => {
                if (context.Request.Path == "/location") {
                    MessageOptions opts = msgOptions.Value;
                    await context.Response
                    .WriteAsync($"{opts.CityName}, {opts.CountryName}");
                } else {
                    await next();
                }
            });

            app.UseRouting();

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



```

    });
  });
}
}
}

```

The options are set up using `IServiceCollection.Configure` in the `ConfigureServices` method. The generic type parameter is used to specify the options class, like this:

```

...
services.Configure<MessageOptions>(options => {
    options.CityName= "Albany";
});
...

```

This statement creates options using the `MessageOptions` class and changes the value of the `CityName` property. When the application starts, the ASP.NET Core platform will create a new instance of the options class and pass it to the function supplied as the argument to the `Configure` method, allowing the default option values to be changed.

The options are accessed by adding an `IOptions<T>` parameter to the `Startup.Configure` method, where the generic type argument specifies the options class, like this:

```

...
public void Configure(IApplicationBuilder app,IWebHostEnvironment env,
    CounterService counter, IOptions<MessageOptions> msgOptions) {
...

```

The `IOptions<T>` interface defines a `Value` property that returns the options object created by the ASP.NET Core platform, allowing middleware components and endpoints to use the options, like this:

```

...
app.Use(async (context, next) => {
    if (context.Request.Path == "/location") {
        MessageOptions opts = msgOptions.Value;
        await context.Response.WriteAsync($"{opts.CityName}, {opts.CountryName}");
    } else {
        await next();
    }
});
...

```

You can see the result by restarting ASP.NET Core and using the browser to navigate to `http://localhost:5000/location`. The middleware component uses the options to produce the output shown in Figure 12-17.

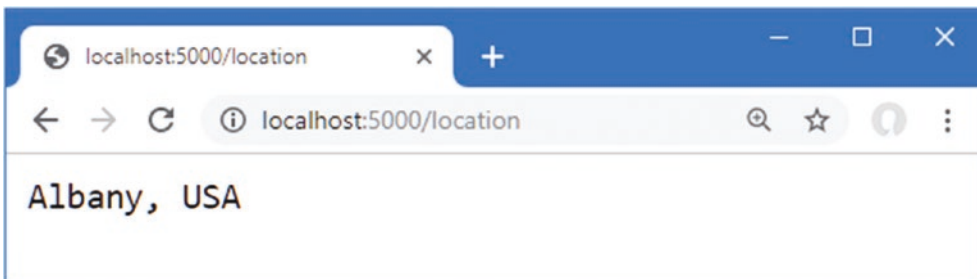


Figure 12-17. Using the options pattern

Using the Options Pattern with Class-Based Middleware

Although the options pattern can be used with lambda function middleware, you will usually see it applied to class-based middleware, such as the built-in features described in Chapters 15 and 16. When used with class-based middleware, the configuration options don't have to be accessed through a `Startup.Configure` method parameter, which produces a more elegant result.

Add the statements shown in Listing 12-15 to the `Middleware.cs` file to define a class-based middleware component that uses the `MessageOptions` class for configuration.

Listing 12-15. Defining a Middleware Component in the `Middleware.cs` File in the Platform Folder

```
using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;
using Microsoft.Extensions.Options;

namespace Platform {

    public class QueryStringMiddleware {
        private RequestDelegate? next;

        // ...statements omitted for brevity...
    }

    public class LocationMiddleware {
        private RequestDelegate next;
        private MessageOptions options;

        public LocationMiddleware(RequestDelegate nextDelegate,
            IOptions<MessageOptions> opts) {
            next = nextDelegate;
            options = opts.Value;
        }

        public async Task Invoke(HttpContext context) {
            if (context.Request.Path == "/location") {
                await context.Response
                    .WriteAsync($"{options.CityName}, {options.CountryName}");
            } else {
                await next(context);
            }
        }
    }
}
```

The `LocationMiddleware` class defines an `IOptions<MessageOptions>` constructor parameter. In Listing 12-16, I have replaced the lambda function middleware component with the class from Listing 12-15 and removed the `IOptions<MessageOptions>` parameter.

Listing 12-16. Using Class-Based Middleware in the `Startup.cs` File in the Platform Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
```

```

using Microsoft.Extensions.Options;

namespace Platform {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.Configure<MessageOptions>(options => {
                options.CityName= "Albany";
            });
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {

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

            app.UseMiddleware<LocationMiddleware>();

            app.UseRouting();

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

```

When the `UseMiddleware` statement is executed, the `LocationMiddleware` constructor is inspected, and its `IOptions<MessageOptions>` parameter will be resolved using the object created in the `ConfigureServices` method. This is done using the dependency injection feature that is described in Chapter 14, but the immediate effect is that the options pattern can be used to easily configure class-based middleware. Restart ASP.NET Core and request `http://localhost:5000/location` to test the new middleware, which will produce the same output as shown in Figure 12-17.

Summary

In this chapter, I focused on the ASP.NET Core platform, introducing the request pipeline, middleware components, and, briefly, services. I described the files added to the project when the Empty template is used and explained the role of the Program and Startup classes. In the next chapter, I describe the ASP.NET Core URL routing feature.

CHAPTER 13



Using URL Routing

The URL routing feature is provided by a pair of middleware components that make it easier to generate responses by consolidating the processing and matching of request URLs. In this chapter, I explain how the ASP.NET Core platform supports URL routing, show its use, and explain why it can be preferable to creating your own custom middleware components. Table 13-1 puts URL routing in context.

■ **Note** This chapter focuses on URL routing for the ASP.NET Core platform. See Part 3 for details of how the higher-level parts of ASP.NET Core build on the features described in this chapter.

Table 13-1. Putting URL Routing in Context

Question	Answer
What is it?	URL routing consolidates the processing and matching of URLs, allowing components known as <i>endpoints</i> to generate responses.
Why is it useful?	URL routing obviates the need for each middleware component to process the URL to see whether the request will be handled or passed along the pipeline. The result is more efficient and easier to maintain.
How is it used?	The URL routing middleware components are added to the request pipeline and configured with a set of routes. Each route contains a URL path and a delegate that will generate a response when a request with the matching path is received.
Are there any pitfalls or limitations?	It can be difficult to define the set of routes matching all the URLs supported by a complex application.
Are there any alternatives?	URL routing is optional, and custom middleware components can be used instead.

Table 13-2 summarizes the chapter.

Table 13-2. Chapter Summary

Problem	Solution	Listing
Handling requests for a specific set of URLs	Define a route with a pattern that matches the required URLs	1-6
Extracting values from URLs	Use segment variables	7-10, 14
Generating URLs	Use the link generator to produce URLs from routes	11-13, 15
Matching URLs with different numbers of segments	Use optional segments or catchall segments in the URL routing pattern	16-18
Restricting matches	Use constraints in the URL routing pattern	19-21, 23-26
Matching requests that are not otherwise handled	Define fallback routes	22
Seeing which endpoint will handle a request	Use the routing context data	27

Preparing for This Chapter

In this chapter, I continue to use the Platform project from Chapter 12. To prepare for this chapter, add a file called `Population.cs` to the Platform folder with the code shown in Listing 13-1.

■ **Tip** You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/apress/pro-asp.net-core-3>. See Chapter 1 for how to get help if you have problems running the examples.

Listing 13-1. The Contents of the Population.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Http;
using System;
using System.Threading.Tasks;

namespace Platform {
    public class Population {
        private RequestDelegate next;

        public Population() { }

        public Population(RequestDelegate nextDelegate) {
            next = nextDelegate;
        }

        public async Task Invoke(HttpContext context) {
            string[] parts = context.Request.Path.ToString()
                .Split("/", StringSplitOptions.RemoveEmptyEntries);
            if (parts.Length == 2 && parts[0] == "population") {
                string city = parts[1];
                int? pop = null;
                switch (city.ToLower()) {
                    case "london":
                        pop = 8_136_000;
                        break;
                    case "paris":
                        pop = 2_141_000;
                        break;
                    case "monaco":
                        pop = 39_000;
                        break;
                }
                if (pop.HasValue) {
                    await context.Response
                        .WriteAsync($"City: {city}, Population: {pop}");
                    return;
                }
            }
            if (next != null) {
                await next(context);
            }
        }
    }
}
```

This middleware component responds to requests for `/population/<city>` where `<city>` is `london`, `paris`, or `monaco`. The middleware component splits up the URL path string, checks that it has the expected length, and uses a `switch` statement to determine if it is a request for a URL that it can respond to. If the URL matches the pattern the middleware is looking for, then a response is generated; otherwise, the request is passed along the pipeline.

Add a class file named `Capital.cs` to the `Platform` folder with the code shown in Listing 13-2.

Listing 13-2. The Contents of the `Capital.cs` File in the `Platform` Folder

```
using Microsoft.AspNetCore.Http;
using System;
using System.Threading.Tasks;

namespace Platform {
    public class Capital {
        private RequestDelegate next;

        public Capital() { }

        public Capital(RequestDelegate nextDelegate) {
            next = nextDelegate;
        }

        public async Task Invoke(HttpContext context) {
            string[] parts = context.Request.Path.ToString()
                .Split("/", StringSplitOptions.RemoveEmptyEntries);
            if (parts.Length == 2 && parts[0] == "capital") {
                string capital = null;
                string country = parts[1];
                switch (country.ToLower()) {
                    case "uk":
                        capital = "London";
                        break;
                    case "france":
                        capital = "Paris";
                        break;
                    case "monaco":
                        context.Response.Redirect($"/population/{country}");
                        return;
                }
                if (capital != null) {
                    await context.Response
                        .WriteAsync($"{capital} is the capital of {country}");
                    return;
                }
            }
            if (next != null) {
                await next(context);
            }
        }
    }
}
```

This middleware component is looking for requests for `/capital/<country>`, where `<country>` is `uk`, `france`, or `monaco`. The capital cities of the United Kingdom and France are displayed, but requests for Monaco, which is a city and a state, are redirected to `/population/monaco`.

Listing 13-3 replaces the middleware examples from the previous chapter and adds the new middleware components to the request pipeline.

Listing 13-3. Replacing the Contents of the Startup.cs File in the Platform Folder

```

using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace Platform {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseMiddleware<Population>();
            app.UseMiddleware<Capital>();
            app.Use(async (context, next) => {
                await context.Response.WriteAsync("Terminal Middleware Reached");
            });
        }
    }
}

```

Start the application by selecting Start Without Debugging or Run Without Debugging from the Debug menu or by opening a new PowerShell command prompt, navigating to the Platform project folder (which contains the Platform.csproj file), and running the command shown in Listing 13-4.

Listing 13-4. Starting the ASP.NET Core Runtime

```
dotnet run
```

Navigate to <http://localhost:5000/population/london>, and you will see the output on the left side of Figure 13-1. Navigate to <http://localhost:5000/capital/france> to see the output from the other middleware component, which is shown on the right side of Figure 13-1.

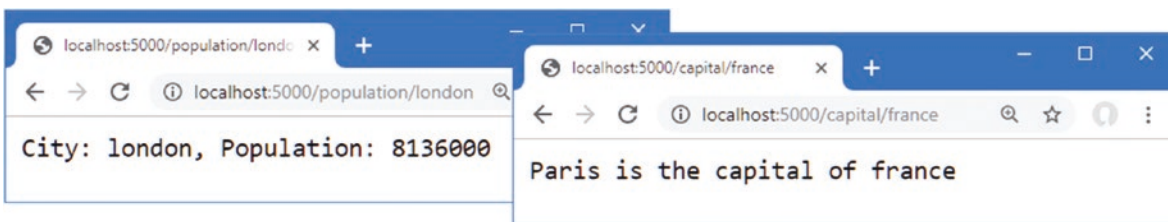


Figure 13-1. Running the example application

Understanding URL Routing

Each middleware component decides whether to act on a request as it passes along the pipeline. Some components are looking for a specific header or query string value, but most components—especially terminal and short-circuiting components—are trying to match URLs.

Each middleware component has to repeat the same set of steps as the request works its way along the pipeline. You can see this in the middleware defined in the previous section, where both components go through the same process: split up the URL, check the number of parts, inspect the first part, and so on.

This approach is inefficient, it is difficult to maintain, and it breaks easily when changed. It is inefficient because the same set of operations is repeated to process the URL. It is difficult to maintain because the URL that each component is looking for is hidden in its code. It breaks easily because changes must be carefully worked through in multiple places. For example, the `Capital` component redirects requests to a URL whose path starts with `/population`, which is handled by the `Population` component. If the `Population` component is revised to support the `/size` URL instead, then this change must also be reflected in the `Capital` component. Real applications can support complex sets of URLs, and working changes fully through individual middleware components can be difficult.

URL routing solves these problems by introducing middleware that takes care of matching request URLs so that components, called *endpoints*, can focus on responses. The mapping between endpoints and the URLs they require is expressed in a *route*. The routing middleware processes the URL, inspects the set of routes and finds the endpoint to handle the request, a process known as *routing*.

Adding the Routing Middleware and Defining an Endpoint

The routing middleware is added using two separate methods: `UseRouting` and `UseEndpoints`. The `UseRouting` method adds the middleware responsible for processing requests to the pipeline. The `UseEndpoints` method is used to define the routes that match URLs to endpoints. URLs are matched using patterns that are compared to the path of request URLs, and each route creates a relationship between one URL pattern and one endpoint. Listing 13-5 shows the use of the routing middleware and contains a simple route.

■ **Tip** I explain why there are two methods for routing in the “Accessing the Endpoint in a Middleware Component” section.

Listing 13-5. Using the Routing Middleware in the `Startup.cs` File in the Platform Folder

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

namespace Platform {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseMiddleware<Population>();
            app.UseMiddleware<Capital>();

            app.UseRouting();

            app.UseEndpoints(endpoints => {
                endpoints.MapGet("routing", async context => {
                    await context.Response.WriteAsync("Request Was Routed");
                });
            });

            app.Use(async (context, next) => {
                await context.Response.WriteAsync("Terminal Middleware Reached");
            });
        }
    }
}
```


There are no arguments to the `UseRouting` method. The `UseEndpoints` method receives a function that accepts an `IEndpointRouteBuilder` object and uses it to create routes using the extension methods described in Table 13-3.

■ **Tip** There are also extension methods that set up endpoints for other parts of ASP.NET Core, such as the MVC Framework, as explained in Part 3.

Table 13-3. The `IEndpointRouteBuilder` Extension Methods

Name	Description
<code>MapGet(pattern, endpoint)</code>	This method routes HTTP GET requests that match the URL pattern to the endpoint.
<code>MapPost(pattern, endpoint)</code>	This method routes HTTP POST requests that match to the URL pattern to the endpoint.
<code>MapPut(pattern, endpoint)</code>	This method routes HTTP PUT requests that match the URL pattern to the endpoint.
<code>MapDelete(pattern, endpoint)</code>	This method routes HTTP DELETE requests that match the URL pattern to the endpoint.
<code>MapMethods(pattern, methods, endpoint)</code>	This method routes requests made with one of the specified HTTP methods that match the URL pattern to the endpoint.
<code>Map(pattern, endpoint)</code>	This method routes all HTTP requests that match the URL pattern to the endpoint.

Endpoints are defined using `RequestDelegate`, which is the same delegate used by conventional middleware, so endpoints are asynchronous methods that receive an `HttpContext` object and use it to generate a response. This means that all the features described in earlier chapters for middleware components can also be used in endpoints.

Restart ASP.NET Core and navigate to `http://localhost:5000/routing` to test the new route. When matching a request, the routing middleware applies the route's URL pattern to the path section of the URL. The path is separated from the hostname by the `/` character, as shown in Figure 13-2.

The diagram shows the URL `http://localhost:5000/routing`. The word `routing` is enclosed in a rectangular box. An upward-pointing arrow originates from the word `Path` centered below the box, pointing directly to the `routing` text.

Figure 13-2. The URL path

The path in the URL matches the pattern specified in the route.

```
...
endpoints.MapGet("routing", async context => {
...

```

URL patterns are expressed without a leading `/` character, which isn't part of the URL path. When the request URL path matches the URL pattern, the request will be forwarded to the endpoint function, which generates the response shown in Figure 13-3.

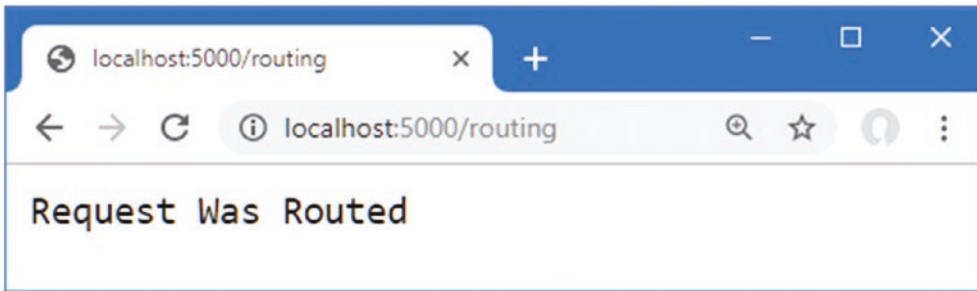


Figure 13-3. Using an endpoint to generate a response

The routing middleware short-circuits the pipeline when a route matches a URL so that the response is generated only by the route's endpoint. The request isn't forwarded to other endpoints or middleware components that appear later in the request pipeline.

If the request URL isn't matched by any route, then the routing middleware passes the request to the next middleware component in the request pipeline. To test this behavior, request the `http://localhost:5000/notrouted` URL, whose path doesn't match the pattern in the route defined in Listing 13-5.

The routing middleware can't match the URL path to a route and forwards the request, which reaches the terminal middleware, producing the response shown in Figure 13-4.

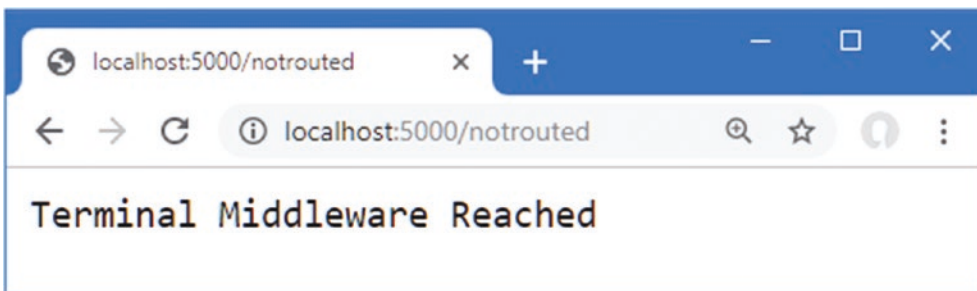


Figure 13-4. Requesting a URL for which there is no matching route

Endpoints generate responses in the same way as the middleware components demonstrated in earlier chapters: they receive an `HttpContext` object that provides access to the request and response through `HttpRequest` and `HttpResponse` objects. This means that any middleware component can also be used as an endpoint. Listing 13-6 adds a route that uses the `Capital` and `Population` middleware components as endpoints.

Listing 13-6. Using Middleware Components as Endpoints in the `Startup.cs` File in the Platform Folder

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

namespace Platform {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
            app.UseDeveloperExceptionPage();
            // app.UseMiddleware<Population>();
            // app.UseMiddleware<Capital>();
        }
    }
}
```

```

app.UseRouting();

app.UseEndpoints(endpoints => {
    endpoints.MapGet("routing", async context => {
        await context.Response.WriteAsync("Request Was Routed");
    });
    endpoints.MapGet("capital/uk", new Capital().Invoke);
    endpoints.MapGet("population/paris", new Population().Invoke);
});

app.Use(async (context, next) => {
    await context.Response.WriteAsync("Terminal Middleware Reached");
});
}
}
}

```

Using middleware components like this is awkward because I need to create new instances of the classes to select the Invoke method as the endpoint. The URL patterns used by the routes support only some of the URLs that the middleware components support, but it is useful to understand that endpoints rely on features that are familiar from earlier chapters. To test the new routes, restart ASP.NET Core and navigate to <http://localhost:5000/capital/uk> and <http://localhost:5000/population/paris>, which will produce the results shown in Figure 13-5.

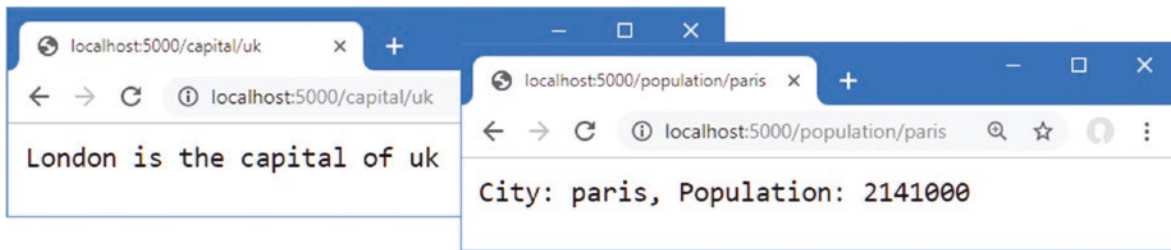


Figure 13-5. Using middleware components as endpoints

Understanding URL Patterns

Using middleware components as endpoints shows that the URL routing feature builds on the standard ASP.NET Core platform. Although the URLs that the application handles can be seen by examining the routes, not all of the URLs understood by the Capital and Population classes are routed, and there have been no efficiency gains since the URL is processed once by the routing middleware to select the route and again by the Capital or Population class in order to extract the data values they require.

Making improvements requires understanding more about how URL patterns are used. When a request arrives, the routing middleware processes the URL to extract the segments from its path, which are the sections of the path separated by the / character, as shown in Figure 13-6.

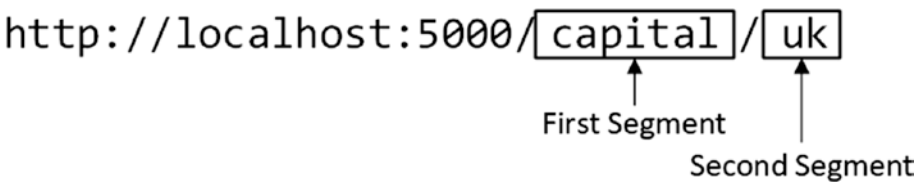


Figure 13-6. The URL segments

The routing middleware also extracts the segments from the URL routing pattern, as shown in Figure 13-7.

```
endpoints.MapGet("capital/uk", new...
```

The diagram shows the code snippet `endpoints.MapGet("capital/uk", new...`. The words `capital` and `uk` are enclosed in rectangular boxes. Below the `capital` box, the text "First Segment" is written with an upward-pointing arrow. Similarly, below the `uk` box, the text "Second Segment" is written with an upward-pointing arrow.

Figure 13-7. The URL pattern segments

To route a request, the segments from the URL pattern are compared to those from the request to see whether they match. The request is routed to the endpoint if its path contains the same number of segments and each segment has the same content as those in the URL pattern, as summarized in Table 13-4.

Table 13-4. Matching URL Segments

URL Path	Description
/capital	No match—too few segments
/capital/europe/uk	No match—too many segments
/name/uk	No match—first segment is not capital
/capital/uk	Matches

Using Segment Variables in URL Patterns

The URL pattern used in Listing 13-6 uses *literal segments*, also known as *static segments*, which match requests using fixed strings. The first segment in the pattern will match only those requests whose path has `capital` as the first segment, for example, and the second segment in the pattern will match only those requests whose second segment is `uk`. Put these together and you can see why the route matches only those requests whose path is `/capital/uk`.

Segment variables, also known as *route parameters*, expand the range of path segments that a pattern segment will match, allowing more flexible routing. Segment variables are given a name and are denoted by curly braces (the `{` and `}` characters), as shown in Listing 13-7.

Listing 13-7. Using Segment Variables in the Startup.cs File in the Platform Folder

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

namespace Platform {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
        }

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

            app.UseEndpoints(endpoints => {
                endpoints.MapGet("{first}/{second}/{third}", async context => {
                    await context.Response.WriteAsync("Request Was Routed\n");
                });
            });
        }
    }
}
```

```

        foreach (var kvp in context.Request.RouteValues) {
            await context.Response
                .WriteAsync($"{kvp.Key}: {kvp.Value}\n");
        }
    });
    endpoints.MapGet("capital/uk", new Capital().Invoke);
    endpoints.MapGet("population/paris", new Population().Invoke);
});

app.Use(async (context, next) => {
    await context.Response.WriteAsync("Terminal Middleware Reached");
});
}
}
}
}
}

```

The URL pattern `{first}/{second}/{third}` matches URLs whose path contains three segments, regardless of what those segments contain. When a segment variable is used, the routing middleware provides the endpoint with the contents of the URL path segment they have matched. This content is available through the `HttpRequest.RouteValues` property, which returns a `RouteValuesDictionary` object. Table 13-5 describes the most useful `RouteValuesDictionary` members.

■ **Tip** There are some reserved words that cannot be used as the names for segment variables: `action`, `area`, `controller`, `handler`, and `page`.

Table 13-5. Useful `RouteValuesDictionary` Members

Name	Description
[key]	The class defines an indexer that allows values to be retrieved by key.
Keys	This property returns the collection of segment variable names.
Values	This property returns the collection of segment variable values.
Count	This property returns the number of segment variables.
ContainsKey(key)	This method returns true if the route data contains a value for the specified key.

The `RouteValuesDictionary` class is enumerable, which means that it can be used in a `foreach` loop to generate a sequence of `KeyValuePair<string, object>` objects, each of which corresponds to the name of a segment variable and the corresponding value extracted from the request URL. The endpoint in Listing 13-7 enumerates the `HttpRequest.RouteValues` property to generate a response that lists the names and value of the segment variables matched by the URL pattern.

The names of the segment variables are `first`, `second`, and `third`, and you can see the values extracted from the URL by restarting ASP.NET Core and requesting any three-segment URL, such as `http://localhost:5000/apples/oranges/cherries`, which produces the response shown in Figure 13-8.

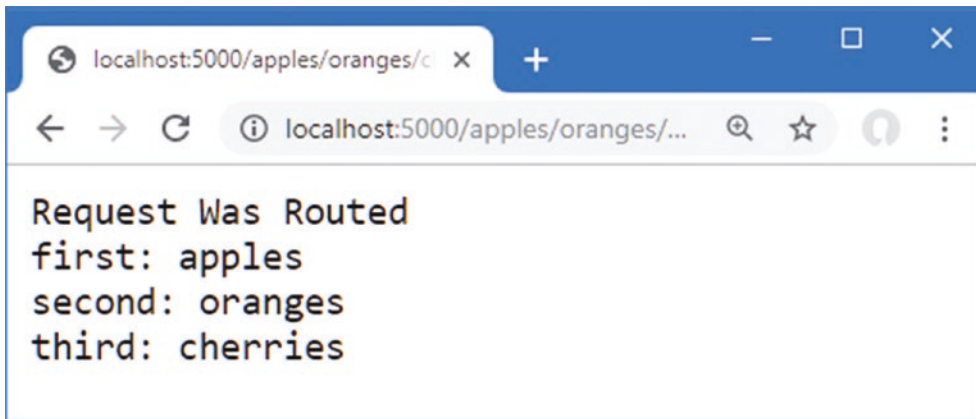


Figure 13-8. Using segment variables

UNDERSTANDING ROUTE SELECTION

When processing a request, the middleware finds all the routes that can match the request and gives each a score, and the route with the lowest score is selected to handle the route. The scoring process is complex, but the effect is that the most specific route receives the request. This means that literal segments are given preference over segment variables and that segment variables with constraints are given preference over those without (constraints are described in the “Constraining Segment Matching” section later in this chapter). The scoring system can produce surprising results, and you should check to make sure that the URLs supported by your application are matched by the routes you expect.

If two routes have the same score, meaning they are equally suited to routing the request, then an exception will be thrown, indicating an ambiguous routing selection. See the “Avoiding Ambiguous Route Exceptions” section later in the chapter for details of how to avoid ambiguous routes.

Refactoring Middleware into an Endpoint

Endpoints usually rely on the routing middleware to provide specific segment variables, rather than enumerating all the segment variables. By relying on the URL pattern to provide a specific value, I can refactor the `Capital` and `Population` classes to depend on the route data, as shown in Listing 13-8.

Listing 13-8. Depending on the Route Data in the `Capital.cs` File in the Platform Folder

```
using Microsoft.AspNetCore.Http;
using System;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Routing;

namespace Platform {
    public static class Capital {

        public static async Task Endpoint(HttpContext context) {
            string capital = null;
            string country = context.Request.RouteValues["country"] as string;
            switch ((country ?? "").ToLower()) {
                case "uk":
                    capital = "London";
                    break;
                case "france":
                    capital = "Paris";
            }
        }
    }
}
```

```

        break;
    case "monaco":
        context.Response.Redirect($"/population/{country}");
        return;
    }
    if (capital != null) {
        await context.Response
            .WriteAsync($"{capital} is the capital of {country}");
    } else {
        context.Response.StatusCode = StatusCodes.Status404NotFound;
    }
}
}
}
}
}

```

Middleware components can be used as endpoints, but the opposite isn't true once there is a dependency on the data provided by the routing middleware. In Listing 13-8, I used the route data to get the value of a segment variable named `country` through the indexer defined by the `RouteValuesDictionary` class.

```

...
string country = context.Request.RouteValues["country"] as string;
...

```

The indexer returns an object value that is cast to a `string` using the `as` keyword. The listing removes the statements that pass the request along the pipeline, which the routing middleware handles on behalf of endpoints.

The use of the segment variable means that requests may be routed to the endpoint with values that are not supported, so I added a statement that returns a 404 status code for countries the endpoint doesn't understand.

I also removed the constructors and replaced the `Invoke` instance method with a static method named `Endpoint`, which better fits with the way that endpoints are used in routes. Listing 13-9 applies the same set of changes to the `Population` class, transforming it from a standard middleware component into an endpoint that depends on the routing middleware to process URLs.

Listing 13-9. Depending on Route Data in the `Population.cs` File in the Platform Folder

```

using Microsoft.AspNetCore.Http;
using System;
using System.Threading.Tasks;

namespace Platform {
    public class Population {

        public static async Task Endpoint(HttpContext context) {
            string city = context.Request.RouteValues["city"] as string;
            int? pop = null;
            switch ((city ?? "").ToLower()) {
                case "london":
                    pop = 8_136_000;
                    break;
                case "paris":
                    pop = 2_141_000;
                    break;
                case "monaco":
                    pop = 39_000;
                    break;
            }
            if (pop.HasValue) {
                await context.Response
                    .WriteAsync($"City: {city}, Population: {pop}");
            } else {

```

```

        context.Response.StatusCode = StatusCodes.Status404NotFound;
    }
}
}
}

```

The change to static methods tidies up the use of the endpoints when defining routes, as shown in Listing 13-10.

Listing 13-10. Updating Routes in the Startup.cs File in the Platform Folder

```

...
app.UseEndpoints(endpoints => {
    endpoints.MapGet("{first}/{second}/{third}", async context => {
        await context.Response.WriteAsync("Request Was Routed\n");
        foreach (var kvp in context.Request.RouteValues) {
            await context.Response.WriteAsync($"{kvp.Key}: {kvp.Value}\n");
        }
    });
    endpoints.MapGet("capital/{country}", Capital.Endpoint);
    endpoints.MapGet("population/{city}", Population.Endpoint);
});
...

```

The new routes match URLs whose path has two segments, the first of which is capital or population. The contents of the second segment are assigned to the segment variables named country and city, allowing the endpoints to support the full set of URLs that were handled at the start of the chapter, without the need to process the URL directly. To test the new routes, restart ASP.NET Core and request `http://localhost:5000/capital/uk` and `http://localhost:5000/population/london`, which will produce the responses shown in Figure 13-9.

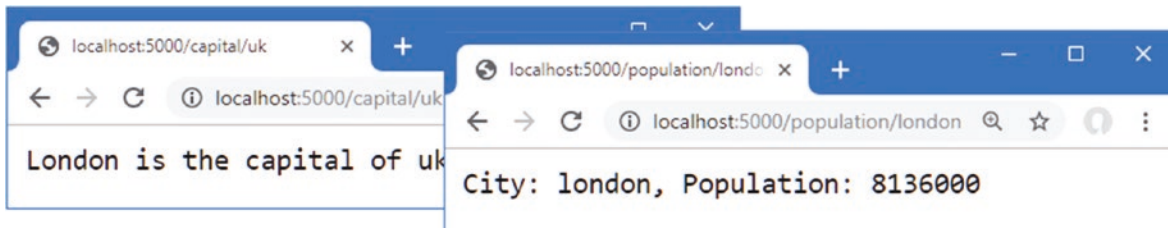


Figure 13-9. Using segment variables in endpoints

These changes address two of the problems I described at the start of the chapter. Efficiency has improved because the URL is processed only once by the routing middleware and not by multiple components. And it is easier to see the URLs that each endpoint supports because the URL patterns show how requests will be matched.

Generating URLs from Routes

The final problem was the difficulty in making changes. The Capital endpoint still has a hardwired dependency on the URL that the Population endpoint supports. To break this dependency, the routing system allows URLs to be generated by supplying data values for segment variables. The first step is to assign a name to the route that will be the target of the URL that is generated, as shown in Listing 13-11.

Listing 13-11. Naming a Route in the Startup.cs File in the Platform Folder

```

using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing;

```



```

namespace Platform {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
        }

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

            app.UseEndpoints(endpoints => {
                endpoints.MapGet("{first}/{second}/{third}", async context => {
                    await context.Response.WriteAsync("Request Was Routed\n");
                    foreach (var kvp in context.Request.RouteValues) {
                        await context.Response
                            .WriteAsync($"{kvp.Key}: {kvp.Value}\n");
                    }
                });
                endpoints.MapGet("capital/{country}", Capital.Endpoint);
                endpoints.MapGet("population/{city}", Population.Endpoint)
                    .WithMetadata(new RouteNameMetadata("population"));
            });

            app.Use(async (context, next) => {
                await context.Response.WriteAsync("Terminal Middleware Reached");
            });
        }
    }
}

```

The `WithMetadata` method is used on the result from the `MapGet` method to assign metadata to the route. The only metadata required for generating URLs is a name, which is assigned by passing a new `RouteNameMetadata` object, whose constructor argument specifies the name that will be used to refer to the route. In Listing 13-11, I have named the route `population`.

■ **Tip** Naming routes helps to avoid links being generated that target a route other than the one you expect, but they can be omitted, in which case the routing system will try to find the best matching route. You can see an example of this approach in Chapter 17.

In Listing 13-12, I have revised the `Capital` endpoint to remove the direct dependency on the `/population` URL and rely on the routing features to generate a URL.

Listing 13-12. Generating a URL in the `Capital.cs` File in the Platform Folder

```

using Microsoft.AspNetCore.Http;
using System;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Routing;
using Microsoft.Extensions.DependencyInjection;

namespace Platform {
    public static class Capital {
        public static async Task Endpoint(HttpContext context) {
            string capital = null;
            string country = context.Request.RouteValues["country"] as string;
            switch ((country ?? "").ToLower()) {
                case "uk":

```

```

        capital= "London";
        break;
    case "france":
        capital = "Paris";
        break;
    case "monaco":
        LinkGenerator generator =
            context.RequestServices.GetService<LinkGenerator>();
        string url = generator.GetPathByRouteValues(context,
            "population", new { city = country });
        context.Response.Redirect(url);
        return;
    }
    if (capital != null) {
        await context.Response
            .WriteAsync($"{capital} is the capital of {country}");
    } else {
        context.Response.StatusCode = StatusCodes.Status404NotFound;
    }
}
}
}
}
}

```

URLs are generated using the `LinkGenerator` class. You can't just create a new `LinkGenerator` instance; one must be obtained using the dependency injection feature that is described in Chapter 14. For the purposes of this chapter, it is enough to know that this statement obtains the `LinkGenerator` object that the endpoint will use:

```

...
LinkGenerator generator = context.RequestServices.GetService<LinkGenerator>();
...

```

The `LinkGenerator` class provides the `GetPathByRouteValues` method, which is used to generate the URL that will be used in the redirection.

```

...
generator.GetPathByRouteValues(context, "population", new { city = country });
...

```

The arguments to the `GetPathByRouteValues` method are the endpoint's `HttpContext` object, the name of the route that will be used to generate the link, and an object that is used to provide values for the segment variables. The `GetPathByRouteValues` method returns a URL that will be routed to the `Population` endpoint, which can be confirmed by restarting ASP.NET Core and requesting the `http://localhost:5000/capital/monaco` URL. The request will be routed to the `Capital` endpoint, which will generate the URL and use it to redirect the browser, producing the result shown in Figure 13-10.



Figure 13-10. Generating a URL

The benefit of this approach is that the URL is generated from the URL pattern in the named route, which means a change in the URL pattern is reflected in the generated URLs, without the need to make changes to endpoints. To demonstrate, Listing 13-13 changes the URL pattern.

Listing 13-13. Changing a URL Pattern in the Startup.cs File in the Platform Folder

```
...
app.UseEndpoints(endpoints => {
    endpoints.MapGet("{first}/{second}/{third}", async context => {
        await context.Response.WriteAsync("Request Was Routed\n");
        foreach (var kvp in context.Request.RouteValues) {
            await context.Response.WriteAsync($"{kvp.Key}: {kvp.Value}\n");
        }
    });
    endpoints.MapGet("capital/{country}", Capital.Endpoint);
    endpoints.MapGet("size/{city}", Population.Endpoint)
        .WithMetadata(new RouteNameMetadata("population"));
});
...
```

The name assigned to the route is unchanged, which ensures that the same endpoint is targeted by the generated URL. To see the effect of the new pattern, restart ASP.NET Core and request the `http://localhost:5000/capital/monaco` URL again. The redirection is to a URL that is matched by the modified pattern, as shown in Figure 13-11. This feature addresses the final problem that I described at the start of the chapter, making it easy to change the URLs that an application supports.

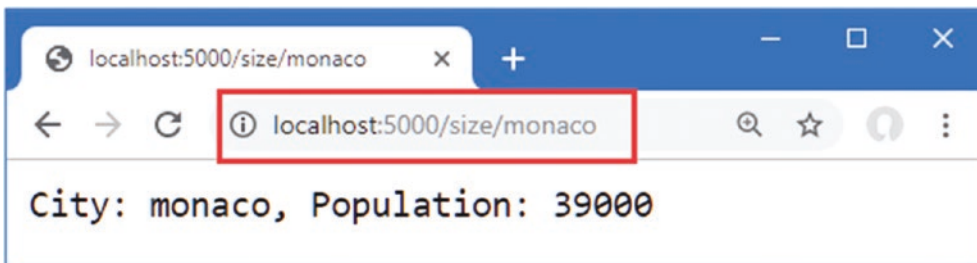


Figure 13-11. Changing the URL pattern

URL ROUTING AND AREAS

The URL routing system supports a feature called *areas*, which allows separate sections of the application to have their own controllers, views, and Razor Pages. I have not described the areas feature in this book because it is not widely used and, when it is used, it tends to cause more problems than it solves. If you want to break up an application, then I recommend creating separate projects.

Managing URL Matching

The previous section introduced the basic URL routing features, but most applications require more work to ensure that URLs are routed correctly, either to increase or to restrict the range of URLs that are matched by a route. In the sections that follow, I show you the different ways that URL patterns can be adjusted to fine-tune the matching process.

Matching Multiple Values from a Single URL Segment

Most segment variables correspond directly to a segment in the URL path, but the routing middleware is able to perform more complex matches, allowing a single segment to be matched to a variable while discarding unwanted characters. Listing 13-14 defines a route that matches only part of a URL segment to a variable.

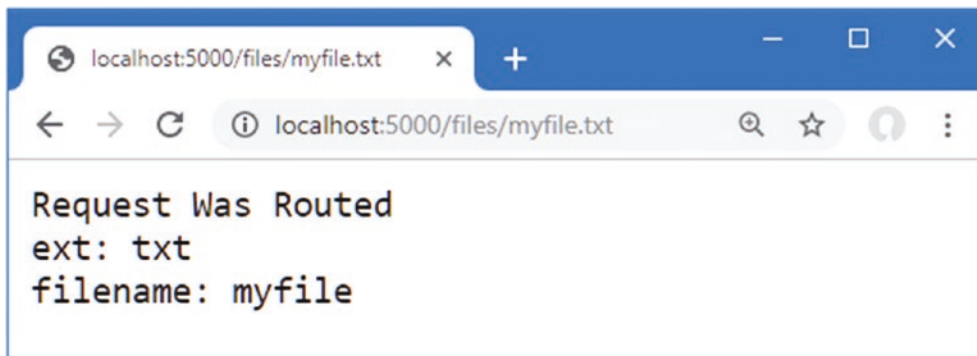
Listing 13-14. Matching Part of a Segment in the Startup.cs File in the Platform Folder

```

...
app.UseEndpoints(endpoints => {
    endpoints.MapGet("files/{filename}.{ext}", async context => {
        await context.Response.WriteAsync("Request Was Routed\n");
        foreach (var kvp in context.Request.RouteValues) {
            await context.Response.WriteAsync($"{kvp.Key}: {kvp.Value}\n");
        }
    });
    endpoints.MapGet("capital/{country}", Capital.Endpoint);
    endpoints.MapGet("size/{city}", Population.Endpoint)
        .WithMetadata(new RouteNameMetadata("population"));
});
...

```

A URL pattern can contain as many segment variables as you need, as long as they are separated by a static string. The requirement for a static separator is so the routing middleware knows where the content for one variable ends and content for the next starts. The pattern in Listing 13-14 matches segment variables named `filename` and `ext`, which are separated by a period; this pattern is often used by process file names. To see how the pattern matches URLs, restart ASP.NET Core and request the `http://localhost:5000/files/myfile.txt` URL, which will produce the response shown in Figure 13-12.

**Figure 13-12.** Matching multiple values from a single path segment

AVOIDING THE COMPLEX PATTERN MISMATCHING PITFALL

The order of the segment variables shown in Figure 13-12 shows that pattern segments that contain multiple variables are matched from right to left. This isn't important most of the time, because endpoints can't rely on a specific key order, but it does show that complex URL patterns are handled differently, which reflects the difficulty in matching them.

In fact, the matching process is so difficult that there can be unexpected matching failures. The specific failures change with each release of ASP.NET Core as the matching process is adjusted to address problems, but the adjustments often introduce new issues. At the time of writing, there is a problem with URL patterns where the content that should be matched by the first variable also appears as a literal string at the start of a segment. This is easier to understand with an example, as shown here:

```

...
endpoints.MapGet("example/red{color}", async context => {
...

```

This pattern has a segment that begins with the literal string `red`, followed by a segment variable named `color`. The routing middleware will correctly match the pattern against the URL path `example/redgreen`, and the value of the `color` route variable will be `green`. However, the URL path `example/redredgreen` won't match because the matching process confuses the position of the literal content with the first part of the content that should be assigned to the `color` variable. This problem may be fixed by the time you read this book, but there will be other issues with complex patterns. It is a good idea to keep URL patterns as simple as possible and make sure you get the matching results you expect.

Using Default Values for Segment Variables

Patterns can be defined with default values that are used when the URL doesn't contain a value for the corresponding segment, increasing the range of URLs that a route can match. Listing 13-15 shows the use of default values in a pattern.

Listing 13-15. Using Default Values in the Startup.cs File in the Platform Folder

```
...
app.UseEndpoints(endpoints => {
    endpoints.MapGet("files/{filename}.{ext}", async context => {
        await context.Response.WriteAsync("Request Was Routed\n");
        foreach (var kvp in context.Request.RouteValues) {
            await context.Response.WriteAsync($"{kvp.Key}: {kvp.Value}\n");
        }
    });
    endpoints.MapGet("capital/{country=France}", Capital.Endpoint);
    endpoints.MapGet("size/{city}", Population.Endpoint)
        .WithMetadata(new RouteNameMetadata("population"));
});
...

```

Default values are defined using an equal sign and the value to be used. The default value in the listing uses the value France when there is no second segment in the URL path. The result is that the range of URLs that can be matched by the route increases, as described in Table 13-6.

Table 13-6. Matching URLs

URL Path	Description
/	No match—too few segments
/city	No match—first segment isn't capital
/capital	Matches, country variable is France
/capital/uk	Matches, country variable is uk
/capital/europe/italy	No match—too many segments

To test the default value, restart ASP.NET Core and navigate to `http://localhost:5000/capital`, which will produce the result shown in Figure 13-13.

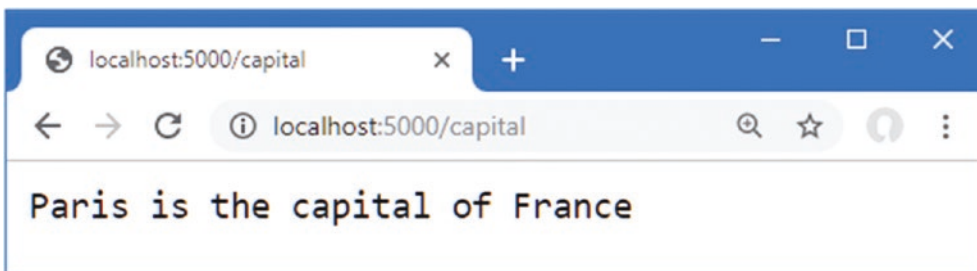


Figure 13-13. Using a default value for a segment variable

Using Optional Segments in a URL Pattern

Default values allow URLs to be matched with fewer segments, but the use of the default value isn't obvious to the endpoint. Some endpoints define their own responses to deal with URLs that omit segments, for which *optional segments* are used. To prepare, Listing 13-16 updates the Population endpoint so that it uses a default value when no city value is available in the routing data.

Listing 13-16. Using a Default Value in the Population.cs File in the Platform Folder

```

using Microsoft.AspNetCore.Http;
using System;
using System.Threading.Tasks;

namespace Platform {
    public class Population {

        public static async Task Endpoint(HttpContext context) {
            string city = context.Request.RouteValues["city"] as string ?? "london";
            int? pop = null;
            switch (city.ToLower()) {
                case "london":
                    pop = 8_136_000;
                    break;
                case "paris":
                    pop = 2_141_000;
                    break;
                case "monaco":
                    pop = 39_000;
                    break;
            }
            if (pop.HasValue) {
                await context.Response
                    .WriteAsync($"City: {city}, Population: {pop}");
            } else {
                context.Response.StatusCode = StatusCodes.Status404NotFound;
            }
        }
    }
}

```

The change uses london as the default value because there is no city segment variable available. Listing 13-17 updates the route for the Population endpoint to make the second segment optional.

Listing 13-17. Using an Optional Segment in the Startup.cs File in the Platform Folder

```

...
app.UseEndpoints(endpoints => {
    endpoints.MapGet("files/{filename}.{ext}", async context => {
        await context.Response.WriteAsync("Request Was Routed\n");
        foreach (var kvp in context.Request.RouteValues) {
            await context.Response.WriteAsync($"{kvp.Key}: {kvp.Value}\n");
        }
    });
    endpoints.MapGet("capital/{country=France}", Capital.Endpoint);
    endpoints.MapGet("size/{city?}", Population.Endpoint)
        .WithMetadata(new RouteNameMetadata("population"));
});
...

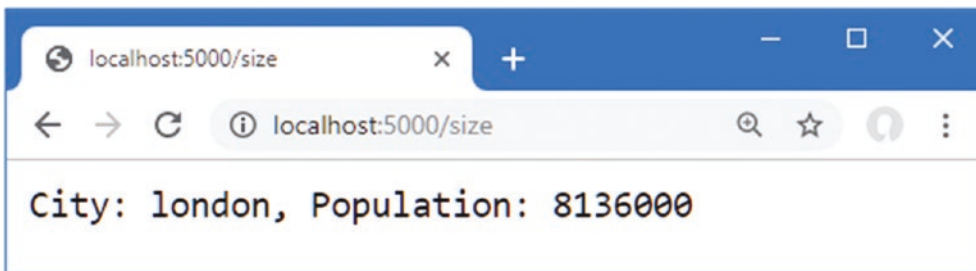
```

Optional segments are denoted with a question mark (the ? character) after the variable name and allow the route to match URLs that don't have a corresponding path segment, as described in Table 13-7.

Table 13-7. Matching URLs

URL Path	Description
/	No match—too few segments.
/city	No match—first segment isn't size.
/size	Matches. No value for the city variable is provided to the endpoint.
/size/paris	Matches, city variable is paris.
/size/europe/italy	No match—too many segments.

To test the optional segment, restart ASP.NET Core and navigate to `http://localhost:5000/size`, which will produce the response shown in Figure 13-14.

**Figure 13-14.** Using an optional segment

Using a catchall Segment Variable

Optional segments allow a pattern to match shorter URL paths. A *catchall* segment does the opposite and allows routes to match URLs that contain more segments than the pattern. A catchall segment is denoted with an asterisk before the variable name, as shown in Listing 13-18.

Listing 13-18. Using a Catchall Segment in the Startup.cs File in the Platform Folder

```

...
app.UseEndpoints(endpoints => {
    endpoints.MapGet("{first}/{second}/*catchall", async context => {
        await context.Response.WriteAsync("Request Was Routed\n");
        foreach (var kvp in context.Request.RouteValues) {
            await context.Response.WriteAsync($"{kvp.Key}: {kvp.Value}\n");
        }
    });
    endpoints.MapGet("capital/{country=France}", Capital.Endpoint);
    endpoints.MapGet("size/{city?}", Population.Endpoint)
        .WithMetadata(new RouteNameMetadata("population"));
});
...

```

The new pattern contains two-segment variables and a catchall, and the result is that the route will match any URL whose path contains two or more segments. There is no upper limit to the number of segments that the URL pattern in this route will match, and the contents of any additional segments are assigned to the segment variable named `catchall`. Restart ASP.NET Core and navigate to `http://localhost:5000/one/two/three/four`, which produces the response shown in Figure 13-15.

■ **Tip** Notice that the segments captured by the catchall are presented in the form `segment/segment/segment` and that the endpoint is responsible for processing the string to break out the individual segments.

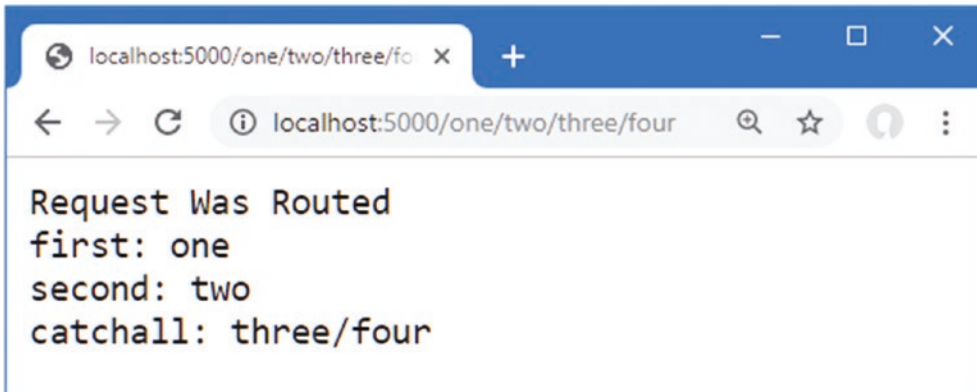


Figure 13-15. Using a catchall segment variable

Constraining Segment Matching

Default values, optional segments, and catchall segments all increase the range of URLs that a route will match. Constraints have the opposite effect and restrict matches. This can be useful if an endpoint can deal only with specific segment contents or if you want to differentiate matching closely related URLs for different endpoints. Constraints are applied by a colon (the `:` character) and a constraint type after a segment variable name, as shown in Listing 13-19.

Listing 13-19. Applying Constraints in the Startup.cs File in the Platform Folder

```
...
app.UseEndpoints(endpoints => {
    endpoints.MapGet("{first:int}/{second:bool}", async context => {
        await context.Response.WriteAsync("Request Was Routed\n");
        foreach (var kvp in context.Request.RouteValues) {
            await context.Response.WriteAsync($"{kvp.Key}: {kvp.Value}\n");
        }
    });
    endpoints.MapGet("capital/{country=France}", Capital.Endpoint);
    endpoints.MapGet("size/{city?}", Population.Endpoint)
        .WithMetadata(new RouteNameMetadata("population"));
});
...
```

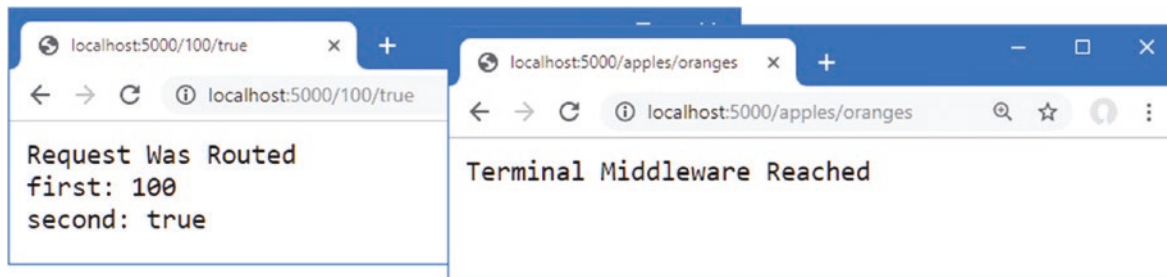
This example constrains the first segment variable so it will match only the path segments that can be parsed to an `int` value, and it constrains the second segment so it will match only the path segments that can be parsed to a `bool`. Values that don't match the constraints won't be matched by the route. Table 13-8 describes the URL pattern constraints.

■ **Note** Some of the constraints match types whose format can differ based on locale. The routing middleware doesn't handle localized formats and will match only those values that are expressed in the invariant culture format.

Table 13-8. *The URL Pattern Constraints*

Constraint	Description
alpha	This constraint matches the letters <i>a</i> to <i>z</i> (and is case-insensitive).
bool	This constraint matches <code>true</code> and <code>false</code> (and is case-insensitive).
datetime	This constraint matches <code>DateTime</code> values, expressed in the nonlocalized invariant culture format.
decimal	This constraint matches decimal values, formatted in the nonlocalized invariant culture.
double	This constraint matches double values, formatted in the nonlocalized invariant culture.
file	This constraint matches segments whose content represents a file name, in the form <code>name.ext</code> . The existence of the file is not validated.
float	This constraint matches float values, formatted in the nonlocalized invariant culture.
guid	This constraint matches GUID values.
int	This constraint matches <code>int</code> values.
length(len)	This constraint matches path segments that have the specified number of characters.
length(min, max)	This constraint matches path segments whose length falls between the lower and upper values specified.
long	This constraint matches long values.
max(val)	This constraint matches path segments that can be parsed to an <code>int</code> value that is less than or equal to the specified value.
maxlength(len)	This constraint matches path segments whose length is equal to or less than the specified value.
min(val)	This constraint matches path segments that can be parsed to an <code>int</code> value that is more than or equal to the specified value.
minlength(len)	This constraint matches path segments whose length is equal to or more than the specified value.
nonfile	This constraint matches segments that do not represent a file name, i.e., values that would not be matched by the <code>file</code> constraint.
range(min, max)	This constraint matches path segments that can be parsed to an <code>int</code> value that falls between the inclusive range specified.
regex(expression)	This constraint applies a regular expression to match path segments.

To test the constraints, restart ASP.NET Core and request `http://localhost:5000/100/true`, which is a URL whose path segments conform to the constraints in Listing 13-19 and that produces the result shown on the left side of Figure 13-16. Request `http://localhost:5000/apples/oranges`, which has the right number of segments but contains values that don't conform to the constraints. None of the routes matches the request, which is forwarded to the terminal middleware, as shown on the right of Figure 13-16.

**Figure 13-16.** *Testing constraints*

Constraints can be combined to further restrict matching, as shown in Listing 13-20.

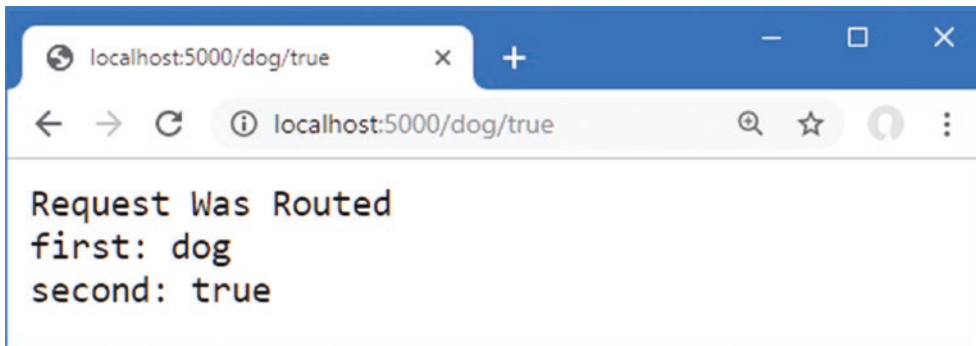
Listing 13-20. Combining URL Pattern Constraints in the Startup.cs File in the Platform Folder

```

...
app.UseEndpoints(endpoints => {
    endpoints.MapGet("{first:alpha:length(3)}/{second:bool}", async context => {
        await context.Response.WriteAsync("Request Was Routed\n");
        foreach (var kvp in context.Request.RouteValues) {
            await context.Response.WriteAsync($"{kvp.Key}: {kvp.Value}\n");
        }
    });
    endpoints.MapGet("capital/{country=France}", Capital.Endpoint);
    endpoints.MapGet("size/{city?}", Population.Endpoint)
        .WithMetadata(new RouteNameMetadata("population"));
});
...

```

The constraints are combined, and only path segments that can satisfy all the constraints will be matched. The combination in Listing 13-20 constrains the URL pattern so that the first segment will match only three alphabetic characters. To test the pattern, restart ASP.NET Core and request `http://localhost:5000/dog/true`, which will produce the output shown in Figure 13-17. Requesting the URL `http://localhost:5000/dogs/true` won't match the route because the first segment contains four characters.

**Figure 13-17.** Combining constraints

Constraining Matching to a Specific Set of Values

The regex constraint applies a regular expression, which provides the basis for one of the most commonly required restrictions: matching only a specific set of values. In Listing 13-21, I have applied the regex constraint to the routes for the Capital endpoint, so it will receive requests only for the values it is able to process.

Listing 13-21. Matching Specific Values in the Startup.cs File in the Platform Folder

```

...
app.UseEndpoints(endpoints => {
    endpoints.MapGet("{first:alpha:length(3)}/{second:bool}", async context => {
        await context.Response.WriteAsync("Request Was Routed\n");
        foreach (var kvp in context.Request.RouteValues) {
            await context.Response.WriteAsync($"{kvp.Key}: {kvp.Value}\n");
        }
    });

    endpoints.MapGet("capital/{country:regex(^uk|france|monaco$)}",
        Capital.Endpoint);
    endpoints.MapGet("size/{city?}", Population.Endpoint)
        .WithMetadata(new RouteNameMetadata("population"));
});
...

```

The route will match only those URLs with two segments. The first segment must be `capital`, and the second segment must be `uk`, `france`, or `monaco`. Regular expressions are case-insensitive, which you can confirm by restarting ASP.NET Core and requesting `http://localhost:5000/capital/UK`, which will produce the result shown in Figure 13-18.

■ **Tip** You may find that your browser requests `/capital/uk`, with a lowercase `uk`. If this happens, clear your browser history and try again.

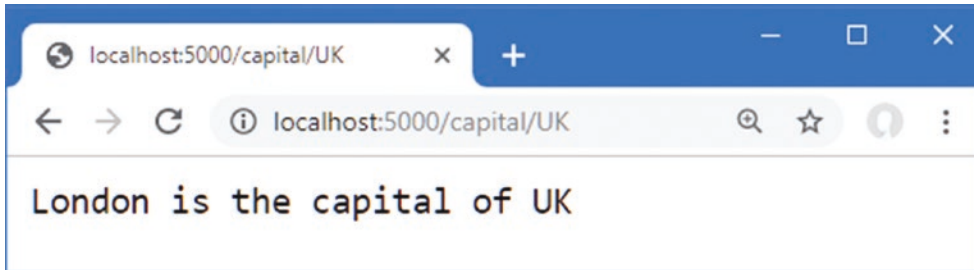


Figure 13-18. Matching specific values with a regular expression

Defining Fallback Routes

Fallback routes direct a request to an endpoint only when no other route matches a request. Fallback routes prevent requests from being passed further along the request pipeline by ensuring that the routing system will always generate a response, as shown in Listing 13-22.

Listing 13-22. Using a Fallback Route in the Startup.cs File in the Platform Folder

```
...
app.UseEndpoints(endpoints => {
    endpoints.MapGet("{first:alpha:length(3)}/{second:bool}", async context => {
        await context.Response.WriteAsync("Request Was Routed\n");
        foreach (var kvp in context.Request.RouteValues) {
            await context.Response.WriteAsync($"{kvp.Key}: {kvp.Value}\n");
        }
    });

    endpoints.MapGet("capital/{country:regex(^uk|france|monaco$})",
        Capital.Endpoint);
    endpoints.MapGet("size/{city?}", Population.Endpoint)
        .WithMetadata(new RouteNameMetadata("population"));

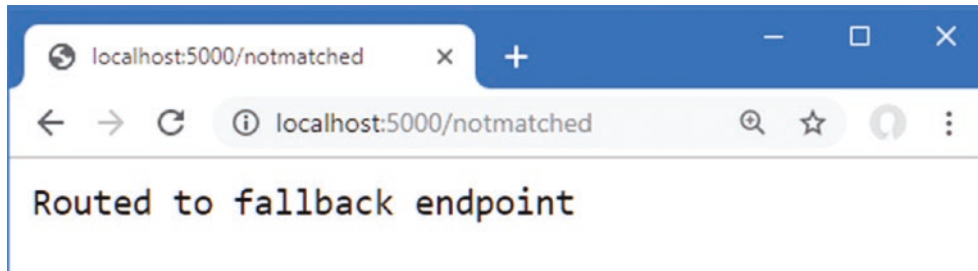
    endpoints.MapFallback(async context => {
        await context.Response.WriteAsync("Routed to fallback endpoint");
    });
});
...
```

The `MapFallback` method creates a route that will be used as a last resort and that will match any request. The methods for creating fallback routes are described in Table 13-9. (There are also methods for creating fallback routes that are specific to other parts of ASP.NET Core and that are described in Part 3.)

Table 13-9. The Methods for Creating Fallback Routes

Name	Description
MapFallback(endpoint)	This method creates a fallback that routes requests to an endpoint.
MapFallbackToFile(path)	This method creates a fallback that routes requests to a file.

With the addition of the route in Listing 13-22, the routing middleware will handle all requests, including those that match none of the regular routes. Restart ASP.NET Core and navigate to a URL that won't be matched by any of the routes, such as `http://localhost:5000/notmatched`, and you will see the response shown in Figure 13-19.

**Figure 13-19.** Using a fallback route

There is no magic to fallback routes. The URL pattern used by fallbacks is `{path:nofile}`, and they rely on the `Order` property to ensure that the route is used only if there are no other suitable routes, which is a feature described in the “Avoiding Ambiguous Route Exceptions” section.

Advanced Routing Features

The routing features described in the previous sections address the needs of most projects, especially since they are usually accessed through higher-level features such as the MVC Framework, described in Part 3. There are some advanced features for projects that have unusual routing requirements, which I describe in the following sections.

Creating Custom Constraints

If the constraints described in Table 13-8 are not sufficient, you can define your own custom constraints by implementing the `IRouteConstraint` interface. To create a custom constraint, add a file named `CountryRouteConstraint.cs` to the `Platform` folder and add the code shown in Listing 13-23.

Listing 13-23. The Contents of the `CountryRouteConstraint.cs` File in the `Platform` Folder

```
using System;
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Routing;

namespace Platform {

    public class CountryRouteConstraint: IRouteConstraint {
        private static string[] countries = { "uk", "france", "monaco" };

        public bool Match(HttpContext httpContext, IRouter route, string routeKey,
            RouteValueDictionary values, RouteDirection routeDirection) {
            string segmentValue = values[routeKey] as string ?? "";
            return Array.IndexOf(countries, segmentValue.ToLower()) > -1;
        }
    }
}
```

The `IRouteConstraint` interface defines the `Match` method, which is called to allow a constraint to decide whether a request should be matched by the route. The parameters for the `Match` method provide the `HttpContext` object for the request, the route, the name of the segment, the segment variables extracted from the URL, and whether the request is to check for an incoming or outgoing URL. The `Match` method returns `true` if the constraint is satisfied by the request and `false` if it is not. The constraint in Listing 13-23 defines a set of countries that are compared to the value of the segment variable to which the constraint has been applied. The constraint is satisfied if the segment matches one of the countries. Custom constraints are set up using the options pattern, as shown in Listing 13-24. (The options pattern is described in Chapter 12.)

Listing 13-24. Using a Custom Constraint in the Startup.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing;

namespace Platform {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.Configure<RouteOptions>(opts => {
                opts.ConstraintMap.Add("countryName",
                    typeof(CountryRouteConstraint));
            });
        }

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

            app.UseEndpoints(endpoints => {
                endpoints.MapGet("{first:alpha:length(3)}/{second:bool}",
                    async context => {
                        await context.Response.WriteAsync("Request Was Routed\n");
                        foreach (var kvp in context.Request.RouteValues) {
                            await context.Response
                                .WriteAsync($"{kvp.Key}: {kvp.Value}\n");
                        }
                    });

                endpoints.MapGet("capital/{country:countryName}", Capital.Endpoint);

                endpoints.MapGet("size/{city?}", Population.Endpoint)
                    .WithMetadata(new RouteNameMetadata("population"));

                endpoints.MapFallback(async context => {
                    await context.Response.WriteAsync("Routed to fallback endpoint");
                });
            });

            app.Use(async (context, next) => {
                await context.Response.WriteAsync("Terminal Middleware Reached");
            });
        }
    }
}
```

The options pattern is applied to the `RouteOptions` class, which defines the `ConstraintMap` property. Each constraint is registered with a key that allows it to be applied in URL patterns. In Listing 13-24, the key for the `CountryRouteConstraint` class is `countryName`, which allows me to constraint a route like this:

```
...
endpoints.MapGet("capital/{country:countryName}", Capital.Endpoint);
...
```

Requests will be matched by this route only when the first segment of the URL is `capital` and the second segment is one of the countries defined in Listing 13-23.

Avoiding Ambiguous Route Exceptions

When trying to route a request, the routing middleware assigns each route a score. As explained earlier in the chapter, precedence is given to more specific routes, and route selection is usually a straightforward process that behaves predictably, albeit with the occasional surprise if you don't think through and test the full range of URLs the application will support.

If two routes have the same score, the routing system can't choose between them and throws an exception, indicating that the routes are ambiguous. In most cases, the best approach is to modify the ambiguous routes to increase specificity by introducing literal segments or a constraint. There are some situations where that won't be possible, and some extra work is required to get the routing system to work as intended. Listing 13-25 replaces the routes from the previous example with two new routes that are ambiguous, but only for some requests.

Listing 13-25. Defining Ambiguous Routes in the `Startup.cs` File in the Platform Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing;

namespace Platform {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.Configure<RouteOptions>(opts => {
                opts.ConstraintMap.Add("countryName",
                    typeof(CountryRouteConstraint));
            });
        }

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

            app.UseEndpoints(endpoints => {
                endpoints.Map("{number:int}", async context => {
                    await context.Response.WriteAsync("Routed to the int endpoint");
                });
                endpoints.Map("{number:double}", async context => {
                    await context.Response
                        .WriteAsync("Routed to the double endpoint");
                });
            });

            app.Use(async (context, next) => {
                await context.Response.WriteAsync("Terminal Middleware Reached");
            });
        }
    }
}
```

These routes are ambiguous only for some values. Only one route matches URLs where the first path segment can be parsed to a double, but both routes match for where the segment can be parsed as an int or a double. To see the issue, restart ASP.NET Core and request `http://localhost:5000/23.5`. The path segment `23.5` can be parsed to a double and produces the response shown on the left side of Figure 13-20. Request `http://localhost:5000/23`, and you will see the exception shown on the right of Figure 13-20. The segment `23` can be parsed as both an int and a double, which means that the routing system cannot identify a single route to handle the request.

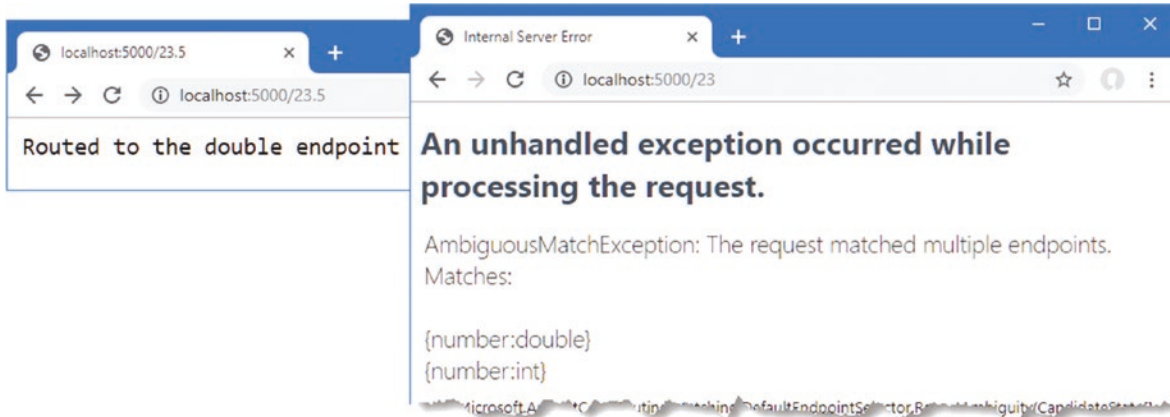


Figure 13-20. An occasionally ambiguous routing configuration

For these situations, preference can be given to a route by defining its order relative to other matching routes, as shown in Listing 13-26.

Listing 13-26. Breaking Route Ambiguity in the Startup.cs File in the Platform Folder

```

...
app.UseEndpoints(endpoints => {
    endpoints.Map("{number:int}", async context => {
        await context.Response.WriteAsync("Routed to the int endpoint");
    }).Add(b => ((RouteEndpointBuilder)b).Order = 1);

    endpoints.Map("{number:double}", async context => {
        await context.Response.WriteAsync("Routed to the double endpoint");
    }).Add(b => ((RouteEndpointBuilder)b).Order = 2);
});
...

```

The process is awkward and requires a call to the `Add` method, casting to a `RouteEndpointBuilder` and setting the value of the `Order` property. Precedence is given to the route with the lowest `Order` value, which means that the changes in Listing 13-26 tell the routing system to use the first route for URLs that both routes can handle. Restart ASP.NET Core and request the `http://localhost:5000/23` URL again, and you will see that the first route handles the request, as shown in Figure 13-21.

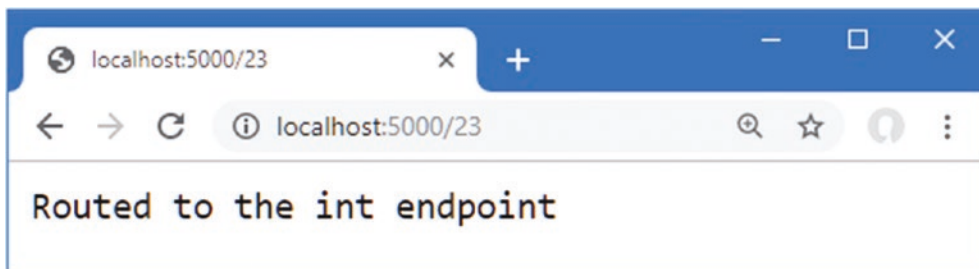


Figure 13-21. Avoiding ambiguous routes

Accessing the Endpoint in a Middleware Component

As earlier chapters demonstrated, not all middleware generates responses. Some components provide features used later in the request pipeline, such as the session middleware, or enhance the response in some way, such as status code middleware.

One limitation of the normal request pipeline is that a middleware component at the start of the pipeline can't tell which of the later components will generate a response. The routing middleware does something different. Although routes are registered in the `UseEndpoints` method, the selection of a route is done in the `UseRouting` method, and the endpoint is executed to generate a response in the `UseEndpoints` method. Any middleware component that is added to the request pipeline between the `UseRouting` method and the `UseEndpoints` method can see which endpoint has been selected before the response is generated and alter its behavior accordingly.

In Listing 13-27, I have added a middleware component that adds different messages to the response based on the route that has been selected to handle the request.

Listing 13-27. Adding a Middleware Component in the Startup.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing;

namespace Platform {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.Configure<RouteOptions>(opts => {
                opts.ConstraintMap.Add("countryName",
                    typeof(CountryRouteConstraint));
            });
        }

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

            app.Use(async (context, next) => {
                Endpoint end = context.GetEndpoint();
                if (end != null) {
                    await context.Response
                        .WriteAsync($"{end.DisplayName} Selected \n");
                } else {
                    await context.Response.WriteAsync("No Endpoint Selected \n");
                }
                await next();
            });

            app.UseEndpoints(endpoints => {
                endpoints.Map("{number:int}", async context => {
                    await context.Response.WriteAsync("Routed to the int endpoint");
                })
                    .WithDisplayName("Int Endpoint")
                    .Add(b => ((RouteEndpointBuilder)b).Order = 1);

                endpoints.Map("{number:double}", async context => {
                    await context.Response
                        .WriteAsync("Routed to the double endpoint");
                })
                    .WithDisplayName("Double Endpoint")
            });
        }
    }
}
```



```

        .Add(b => ((RouteEndpointBuilder)b).Order = 2);
    });

    app.Use(async (context, next) => {
        await context.Response.WriteAsync("Terminal Middleware Reached");
    });
}
}
}
}
}

```

The `GetEndpoint` extension method on the `HttpContext` class returns the endpoint that has been selected to handle the request, described through an `Endpoint` object. The `Endpoint` class defines the properties described in Table 13-10.

■ **Caution** There is also a `SetEndpoint` method that allows the endpoint chosen by the routing middleware to be changed before the response is generated. This should be used with caution and only when there is a compelling need to interfere with the normal route selection process.

Table 13-10. *The Properties Defined by the Endpoint Class*

Name	Description
<code>DisplayName</code>	This property returns the display name associated with the endpoint, which can be set using the <code>WithDisplayName</code> method when creating a route.
<code>Metadata</code>	This property returns the collection of metadata associated with the endpoint.
<code>RequestDelegate</code>	This property returns the delegate that will be used to generate the response.

To make it easier to identify the endpoint that the routing middleware has selected, I used the `WithDisplayName` method to assign names to the routes in Listing 13-27. The new middleware component adds a message to the response reporting the endpoint that has been selected. Restart ASP.NET Core and request the `http://localhost:5000/23` URL to see the output from the middleware that shows the endpoint has been selected between the two methods that add the routing middleware to the request pipeline, as shown in Figure 13-22.

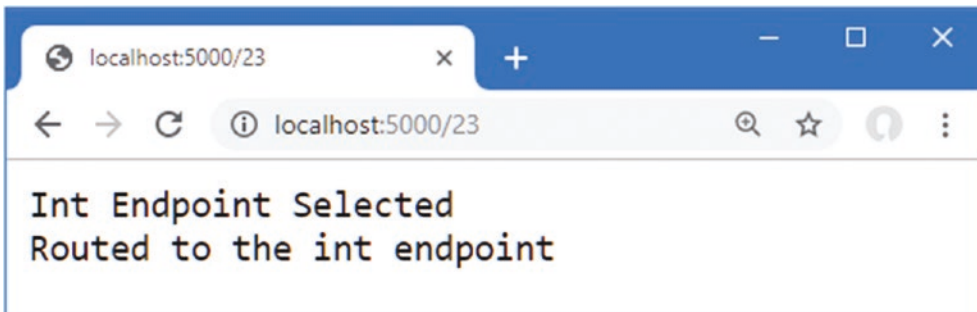


Figure 13-22. *Determining the endpoint*

Summary

In this chapter, I introduced the endpoint routing system and explained how it deals with some common problems arising in regular middleware. I showed you how to define routes, how to match and generate URLs, and how to use constraints to restrict the use of routes. I also showed you some of the advanced uses of the routing system, including custom constraints and avoiding route ambiguity. In the next chapter, I explain how ASP.NET Core services work.



Using Dependency Injection

Services are objects that are shared between middleware components and endpoints. There are no restrictions on the features that services can provide, but they are usually used for tasks that are needed in multiple parts of the application, such as logging or database access.

The ASP.NET Core *dependency injection* feature is used to create and consume services. This is a topic that causes confusion and can be difficult to understand. In this chapter, I describe the problems that dependency injection solves and explain how dependency injection is supported by the ASP.NET Core platform. Table 14-1 puts dependency injection in context.

Table 14-1. *Putting Dependency Injection in Context*

Question	Answer
What is it?	Dependency Injection makes it easy to create loosely coupled components, which typically means that components consume functionality defined by interfaces without having any firsthand knowledge of which implementation classes are being used.
Why is it useful?	Dependency injection makes it easier to change the behavior of an application by changing the components that implement the interfaces that define application features. It also results in components that are easier to isolate for unit testing.
How is it used?	The Startup class is used to specify which implementation classes are used to deliver the functionality specified by the interfaces used by the application. Services can be explicitly requested through the <i>IServiceProvider</i> interface or by declaring constructor or method parameters.
Are there any pitfalls or limitations?	There are some differences in the way that middleware components and endpoints are handled and the way that services with different lifecycles are accessed.
Are there any alternatives?	You don't have to use dependency injection in your own code, but it is helpful to know how it works because it is used by the ASP.NET Core platform to provide features to developers.

Table 14-2 summarizes the chapter.

Table 14-2. *Chapter Summary*

Problem	Solution	Listing
Obtaining a service in the Startup class	Add a parameter to the Configure method	13
Obtaining a service in a middleware component	Define a constructor parameter	14, 33-35
Obtaining a service in an endpoint	Get an <i>IServiceProvider</i> object through the context objects	15-18
Instantiating a class that has constructor dependencies	Use the <i>ActivatorUtilities</i> class	19-21
Defining services that are instantiated for every dependency	Define transient services	22-27
Defining services that are instantiated for every request	Define scoped services	28-32
Accessing services in the Startup.ConfigureServices method	Define a Startup constructor parameter and assign the value to the property	36

Table 14-2. (continued)

Problem	Solution	Listing
Managing service instantiation	Use a service factory	37, 38
Defining multiple implementations for a service	Define multiple services with the same scope and consume them through the <code>GetServices</code> method	39–41
Using services that support generic type parameters	Use a service with an unbound type	42

Preparing for This Chapter

In this chapter, I continue to use the Platform project from Chapter 13. New classes are required to prepare for this chapter. Start by creating the `Platform/Services` folder and add to it a class file named `IResponseFormatter.cs`, with the code shown in Listing 14-1.

■ **Tip** You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/apress/pro-asp.net-core-3>. See Chapter 1 for how to get help if you have problems running the examples.

Listing 14-1. The Contents of the `IResponseFormatter.cs` File in the Services Folder

```
using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;

namespace Platform.Services {
    public interface IResponseFormatter {

        Task Format(HttpContext context, string content);
    }
}
```

The `IResponseFormatter` interface defines a single method that receives an `HttpContext` object and a string. To create an implementation of the interface, add a class called `TextResponseFormatter.cs` to the `Platform/Services` folder with the code shown in Listing 14-2.

Listing 14-2. The Contents of the `TextResponseFormatter.cs` File in the Services Folder

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;

namespace Platform.Services {
    public class TextResponseFormatter : IResponseFormatter {
        private int responseCounter = 0;

        public async Task Format(HttpContext context, string content) {
            await context.Response.
                WriteAsync($"Response {++responseCounter}:\n{content}");
        }
    }
}
```

The `TextResponseFormatter` class implements the interface and writes the content to the response as a simple string with a prefix to make it obvious when the class is used.

Creating a Middleware Component and an Endpoint

Some of the examples in this chapter show how features are applied differently when using middleware and endpoints. Add a file called `WeatherMiddleware.cs` to the `Platform` folder with the code shown in Listing 14-3.

Listing 14-3. The Contents of the `WeatherMiddleware.cs` File in the `Platform` Folder

```
using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;

namespace Platform {
    public class WeatherMiddleware {
        private RequestDelegate next;

        public WeatherMiddleware(RequestDelegate nextDelegate) {
            next = nextDelegate;
        }

        public async Task Invoke(HttpContext context) {
            if (context.Request.Path == "/middleware/class") {
                await context.Response
                    .WriteAsync("Middleware Class: It is raining in London");
            } else {
                await next(context);
            }
        }
    }
}
```

To create an endpoint that produces a similar result to the middleware component, add a file called `WeatherEndpoint.cs` to the `Platform` folder with the code shown in Listing 14-4.

Listing 14-4. The Contents of the `WeatherEndpoint.cs` File in the `Platform` Folder

```
using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;

namespace Platform {
    public class WeatherEndpoint {

        public static async Task Endpoint(HttpContext context) {
            await context.Response
                .WriteAsync("Endpoint Class: It is cloudy in Milan");
        }
    }
}
```

Configuring the Request Pipeline

Replace the contents of the `Startup.cs` file with those shown in Listing 14-5. The classes defined in the previous section are applied alongside lambda functions that produce similar results.

Listing 14-5. Replacing the Contents of the `Startup.cs` File in the `Platform` Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing;
```

```

using Platform.Services;

namespace Platform {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseRouting();
            app.UseMiddleware<WeatherMiddleware>();

            IResponseFormatter formatter = new TextResponseFormatter();
            app.Use(async (context, next) => {
                if (context.Request.Path == "/middleware/function") {
                    await formatter.Format(context,
                        "Middleware Function: It is snowing in Chicago");
                } else {
                    await next();
                }
            });

            app.UseEndpoints(endpoints => {

                endpoints.MapGet("/endpoint/class", WeatherEndpoint.Endpoint);

                endpoints.MapGet("/endpoint/function", async context => {
                    await context.Response
                        .WriteAsync("Endpoint Function: It is sunny in LA");
                });
            });
        }
    }
}

```

Start the application by selecting Start Without Debugging or Run Without Debugging from the Debug menu or by opening a new PowerShell command prompt, navigating to the Platform project folder (which contains the Platform.csproj file, and running the command shown in Listing 14-6.

Listing 14-6. Starting the ASP.NET Core Runtime

```
dotnet run
```

Use a browser to request `http://localhost:5000/middleware/function`, and you will see the response shown in Figure 14-1. Each time you reload the browser, the counter shown in the response will be incremented.

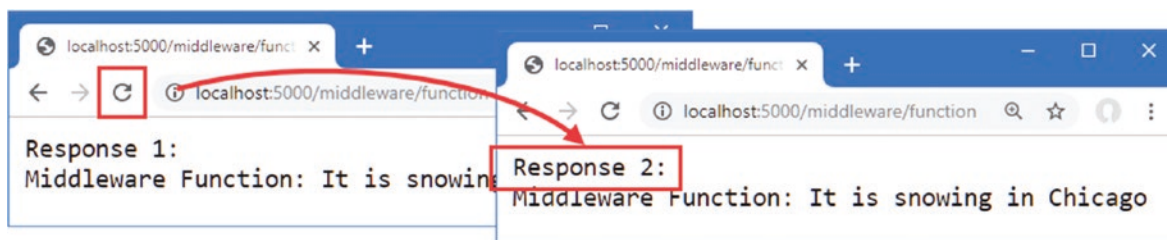


Figure 14-1. Running the example application

Understanding Service Location and Tight Coupling

To understand dependency injection, it is important to start with the two problems it solves. In the sections that follow, I describe both problems addressed by dependency injection.

TAKING A VIEW ON DEPENDENCY INJECTION

Dependency injection is one of the topics that readers contact me about most often. About half of the emails complain that I am “forcing” DI upon them. Oddly, the other half are complaints that I did not emphasize the benefits of DI strongly enough and other readers may not have realized how useful it can be.

Dependency injection can be a difficult topic to understand, and its value is contentious. DI can be a useful tool, but not everyone likes it—or needs it.

DI offers limited benefit if you are not doing unit testing or if you are working on a small, self-contained and stable project. It is still helpful to understand how DI works because DI is used to access some important ASP.NET Core features described in earlier chapters, but you don’t always need to embrace DI in the custom classes you write. There are alternative ways of creating shared features—two of which I describe in the following sections—and using these is perfectly acceptable if you don’t like DI.

I rely on DI in my own applications because I find that projects often go in unexpected directions, and being able to easily replace a component with a new implementation can save me a lot of tedious and error-prone changes. I’d rather put in some effort at the start of the project than do a complex set of edits later.

But I am not dogmatic about dependency injection and nor should you be. Dependency injection solves a problem that doesn’t arise in every project, and only you can determine whether you need DI for your project.

Understanding the Service Location Problem

Most projects have features that need to be used in different parts of the application, which are known as *services*. Common examples include logging tools and configuration settings but can extend to any shared feature, including the `TextResponseFormatter` class that is defined in Listing 14-2 and used by one of the middleware components.

Each `TextResponseFormatter` object maintains a counter that is included in the response sent to the browser, and if I want to incorporate the same counter into the responses generated by other endpoints, I need to have a way to make a single `TextResponseFormatter` object available in such a way that it can be easily found and consumed at every point where responses are generated.

There are many ways to make services locatable, but there are two main approaches, aside from the one that is the main topic of this chapter. The first approach is to create an object and use it as a constructor or method argument to pass it to the part of the application where it is required. The other approach is to add a static property to the service class that provides direct access to the shared instance, as shown in Listing 14-7. This is known as the *singleton pattern*, and it was a common approach before the widespread use of dependency injection.

Listing 14-7. Creating a Singleton in the `TextResponseFormatter.cs` File in the Services Folder

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;

namespace Platform.Services {
    public class TextResponseFormatter : IResponseFormatter {
        private int responseCounter = 0;
        private static TextResponseFormatter shared;

        public async Task Format(HttpContext context, string content) {
            await context.Response.
                WriteAsync($"Response {++responseCounter}: \n{content}");
        }
    }
}
```

```

    public static TextResponseFormatter Singleton {
        get {
            if (shared == null) {
                shared = new TextResponseFormatter();
            }
            return shared;
        }
    }
}

```

This is a basic implementation of the singleton pattern, and there are many variations that pay closer attention to issues such as safe concurrent access. What's important for this chapter is that the changes in Listing 14-7 rely on the consumers of the `TextResponseFormatter` service obtaining a shared object through the static `Singleton` property, as shown in Listing 14-8.

Listing 14-8. Using a Service in the Startup.cs File in the Platform Folder

```

using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing;
using Platform.Services;

namespace Platform {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseRouting();
            app.UseMiddleware<WeatherMiddleware>();

            app.Use(async (context, next) => {
                if (context.Request.Path == "/middleware/function") {
                    await TextResponseFormatter.Singleton.Format(context,
                        "Middleware Function: It is snowing in Chicago");
                } else {
                    await next();
                }
            });

            app.UseEndpoints(endpoints => {

                endpoints.MapGet("/endpoint/class", WeatherEndpoint.Endpoint);

                endpoints.MapGet("/endpoint/function", async context => {
                    await TextResponseFormatter.Singleton.Format(context,
                        "Endpoint Function: It is sunny in LA");
                });
            });
        }
    }
}

```

The singleton pattern allows me to share a single `TextResponseFormatter` so it is used by a middleware component and an endpoint, with the effect that a single counter is incremented by requests for two different URLs. To see the effect of the singleton pattern, restart ASP.NET Core and request the `http://localhost:5000/middleware/function` and `http://localhost:5000/endpoint/function` URLs. A single counter is updated for both URLs, as shown in Figure 14-2.

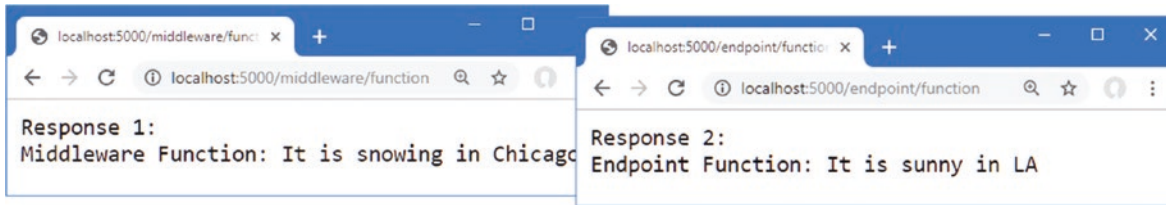


Figure 14-2. Implementing the singleton pattern to create a shared service

The singleton pattern is simple to understand and easy to use, but the knowledge of how services are located is spread throughout the application, and every service class and every service consumer needs to understand how to access the shared object. This can lead to variations in the singleton pattern as new services are created and creates many points in the code that must be updated when there is a change. This pattern can also be rigid and doesn't allow any flexibility in how services are managed because every consumer always shares a single service object.

Understanding the Tightly Coupled Components Problem

Although I defined an interface in Listing 14-1, the way that I have used the singleton pattern means that consumers are always aware of the implementation class they are using because that's the class whose static property is used to get the shared object. If I want to switch to a different implementation of the `IResponseFormatter` interface, I must locate every use of the service and replace the existing implementation class with the new one. There are patterns to solve this problem, too, such as the *type broker* pattern, in which a class provides access to singleton objects through their interfaces. Add a class file called `TypeBroker.cs` to the `Platform/Services` folder and use it to define the code shown in Listing 14-9.

Listing 14-9. The Contents of the `TypeBroker.cs` File in the `Services` Folder

```
namespace Platform.Services {
    public static class TypeBroker {
        private static IResponseFormatter formatter = new TextResponseFormatter();

        public static IResponseFormatter Formatter => formatter;
    }
}
```

The `Formatter` property provides access to a shared service object that implements the `IResponseFormatter` interface. Consumers of the service need to know that the `TypeBroker` class is responsible for selecting the implementation that will be used, but this pattern means that service consumers can work through interfaces rather than concrete classes, as shown in Listing 14-10.

Listing 14-10. Using a `TypeBroker` in the `Startup.cs` File in the `Platform` Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing;
using Platform.Services;

namespace Platform {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
        }
    }
}
```



```

public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
    app.UseDeveloperExceptionPage();
    app.UseRouting();
    app.UseMiddleware<WeatherMiddleware>();

    app.Use(async (context, next) => {
        if (context.Request.Path == "/middleware/function") {
            await TypeBroker.Formatter.Format(context,
                "Middleware Function: It is snowing in Chicago");
        } else {
            await next();
        }
    });

    app.UseEndpoints(endpoints => {

        endpoints.MapGet("/endpoint/class", WeatherEndpoint.Endpoint);

        endpoints.MapGet("/endpoint/function", async context => {
            await TypeBroker.Formatter.Format(context,
                "Endpoint Function: It is sunny in LA");
        });
    });
}
}
}
}
}

```

This approach makes it easy to switch to a different implementation class by altering just the `TypeBroker` class and prevents service consumers from creating dependencies on a specific implementation. It also means that service classes can focus on the features they provide without having to deal with how those features will be located. To demonstrate, add a class file called `HtmlResponseFormatter.cs` to the `Platform/Services` folder with the code shown in Listing 14-11.

Listing 14-11. The Contents of the `HtmlResponseFormatter.cs` File in the `Services` Folder

```

using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;

namespace Platform.Services {
    public class HtmlResponseFormatter : IResponseFormatter {

        public async Task Format(HttpContext context, string content) {
            context.Response.ContentType = "text/html";
            await context.Response.WriteAsync($"@"
                <!DOCTYPE html>
                <html lang=""en"">
                <head><title>Response</title></head>
                <body>
                    <h2>Formatted Response</h2>
                    <span>{content}</span>
                </body>
                </html>");
        }
    }
}

```

This implementation of the `IResponseFormatter` sets the `ContentType` property of the `HttpResponse` object and inserts the content into an HTML template string. To use the new formatter class, I only need to change the `TypeBroker`, as shown in Listing 14-12.

Listing 14-12. Changing Implementation in the TypeBroker.cs File in the Platform/Services Folder

```
namespace Platform.Services {
    public static class TypeBroker {
        private static IResponseFormatter formatter = new HtmlResponseFormatter();

        public static IResponseFormatter Formatter => formatter;
    }
}
```

To make sure the new formatter works, restart ASP.NET Core and request `http://localhost:5000/endpoint/function`, which will produce the result shown in Figure 14-3.

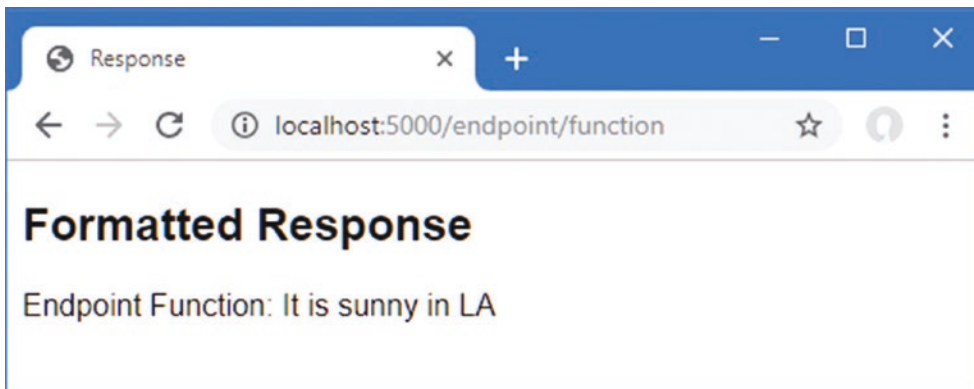


Figure 14-3. Using a different service implementation class

Using Dependency Injection

Dependency injection provides an alternative approach to providing services that tidy up the rough edges that arise in the singleton and type broker pattern, and it is integrated with other ASP.NET Core features. Listing 14-13 shows the use of ASP.NET Core dependency injection to replace the type broker from the previous section.

Listing 14-13. Using Dependency Injection in the Startup.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing;
using Platform.Services;

namespace Platform {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.AddSingleton<IResponseFormatter, HtmlResponseFormatter>();
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env,
            IResponseFormatter formatter) {

            app.UseDeveloperExceptionPage();
            app.UseRouting();
        }
    }
}
```

```

app.UseMiddleware<WeatherMiddleware>();

app.Use(async (context, next) => {
    if (context.Request.Path == "/middleware/function") {
        await formatter.Format(context,
            "Middleware Function: It is snowing in Chicago");
    } else {
        await next();
    }
});

app.UseEndpoints(endpoints => {
    endpoints.MapGet("/endpoint/class", WeatherEndpoint.Endpoint);

    endpoints.MapGet("/endpoint/function", async context => {
        await formatter.Format(context,
            "Endpoint Function: It is sunny in LA");
    });
});
}
}
}
}
}

```

Services are registered in the `ConfigureServices` method of the `Startup` class, using extension methods on the `IServiceCollection` parameter. In Listing 14-13, I used an extension method to create a service for the `IResponseFormatter` interface.

```

...
public void ConfigureServices(IServiceCollection services) {
    services.AddSingleton<IResponseFormatter, HtmlResponseFormatter>();
}
...

```

The `AddSingleton` method is one of the extension methods available for services and tells ASP.NET Core that a single object should be used to satisfy demands for the service (the other extension methods are described in the “Using Service Lifecycles” section). The interface and the implementation class are specified as generic type arguments. To consume the service, I added a parameter to the `Configure` method.

```

...
public void Configure(IApplicationBuilder app, IWebHostEnvironment env,
    IResponseFormatter formatter) {
...

```

The new parameter declares a dependency on the `IResponseFormatter` interface, and the method is said to depend on the interface. Before the `Configure` method is invoked, its parameters are inspected, the dependency is detected, and the application’s services are inspected to determine whether it is possible to resolve the dependency. The statement in the `ConfigureServices` method tells the dependency injection system that a dependency on the `IResponseFormatter` interface can be resolved with an `HtmlResponseFormatter` object. The object is created and used as an argument to invoke the method. Because the object that resolves the dependency is provided from outside the class or function that uses it, it is said to have been injected, which is why the process is known as *dependency injection*.

Using a Service in a Middleware Class

Defining a service in the `ConfigureServices` method and consuming it in the `Configure` method may not seem impressive, but once a service is defined, it can be used almost anywhere in an ASP.NET Core application. Listing 14-14 declares a dependency on the `IResponseFormatter` interface in the middleware class defined at the start of the chapter.

Listing 14-14. Declaring a Dependency in the WeatherMiddleware.cs File in the Services Folder

```

using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;
using Platform.Services;

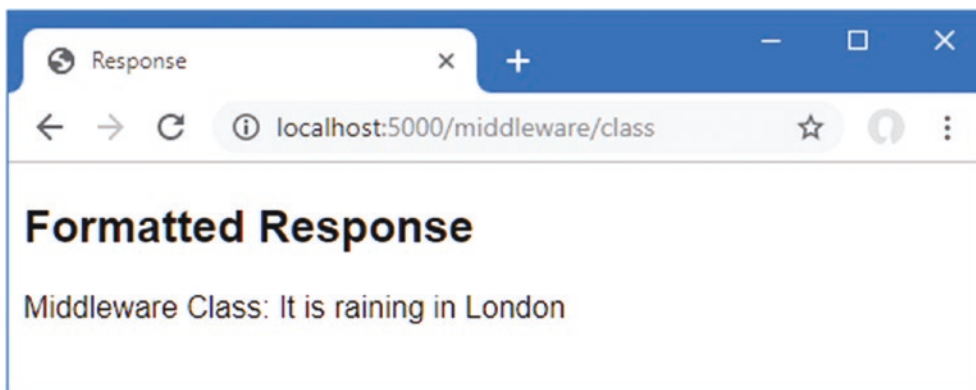
namespace Platform {
    public class WeatherMiddleware {
        private RequestDelegate next;
        private IResponseFormatter formatter;

        public WeatherMiddleware(RequestDelegate nextDelegate,
            IResponseFormatter respFormatter) {
            next = nextDelegate;
            formatter = respFormatter;
        }

        public async Task Invoke(HttpContext context) {
            if (context.Request.Path == "/middleware/class") {
                await formatter.Format(context,
                    "Middleware Class: It is raining in London");
            } else {
                await next(context);
            }
        }
    }
}

```

To declare the dependency, I added a constructor parameter. To see the result, restart ASP.NET Core and request the `http://localhost:5000/middleware/class` URL, which will produce the response shown in Figure 14-4.

**Figure 14-4.** Declaring a dependency in a middleware class

When the request pipeline is being set up, the ASP.NET Core platform reaches the statement in the `Configure` method that adds the `WeatherMiddleware` class as a component.

```

...
app.UseMiddleware<WeatherMiddleware>();
...

```

The platform understands it needs to create an instance of the `WeatherMiddleware` class and inspects the constructor. The dependency on the `IResponseFormatter` interface is detected, the services are inspected to see if the dependency can be resolved, and the shared service object is used when the constructor is invoked.

There are two important points to understand about this example. The first is that `WeatherMiddleware` doesn't know which implementation class will be used to resolve its dependency on the `IResponseFormatter` interface—it just knows that it will receive an object that conforms to the interface through its constructor parameter. Second, the `WeatherMiddleware` class doesn't know how the dependency is resolved—it just declares a constructor parameter. This is a more elegant approach than my implementations of the singleton and type broker patterns earlier in the chapter, and I can change the implementation class used to resolve the service by changing the generic type parameters used in the `Startup.ConfigureServices` method.

Using a Service in an Endpoint

The situation is more complicated in the `WeatherEndpoint` class, which is static and doesn't have a constructor through which dependencies can be declared. There are several approaches available to resolve dependencies for an endpoint class, which are described in the sections that follow.

Getting Services from the `HttpContext` Object

Services can be accessed through the `HttpContext` object that is received when a request is routed to an endpoint, as shown in Listing 14-15.

Listing 14-15. Using a Service in the `WeatherEndpoint.cs` File in the Platform Folder

```
using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;
using Platform.Services;
using Microsoft.Extensions.DependencyInjection;

namespace Platform {
    public class WeatherEndpoint {

        public static async Task Endpoint(HttpContext context) {
            IResponseFormatter formatter =
                context.RequestServices.GetRequiredService<IResponseFormatter>();
            await formatter.Format(context, "Endpoint Class: It is cloudy in Milan");
        }
    }
}
```

The `HttpContext.RequestServices` property returns an object that implements the `IServiceProvider` interfaces, which provides access to the services that have been configured in the application's `Start.ConfigureServices` method. The `Microsoft.Extensions.DependencyInjection` namespace used in Listing 14-15 contains extension methods for the `IServiceProvider` interface that allow individual services to be obtained, as described in Table 14-3.

Table 14-3. The `IServiceProvider` Extension Methods for Obtaining Services

Name	Description
<code>GetService<T>()</code>	This method returns a service for the type specified by the generic type parameter or null if no such service has been defined.
<code>GetService(type)</code>	This method returns a service for the type specified or null if no such service has been defined.
<code>GetRequiredService<T>()</code>	This method returns a service specified by the generic type parameter and throws an exception if a service isn't available.
<code>GetRequiredService(type)</code>	This method returns a service for the type specified and throws an exception if a service isn't available.

When the Endpoint method is invoked, the `GetRequiredService<T>` method is used to obtain an `IResponseFormatter` object, which is used to format the response. To see the effect, restart ASP.NET Core and use the browser to request `http://localhost:5000/endpoint/class`, which will produce the formatted response shown in Figure 14-5.

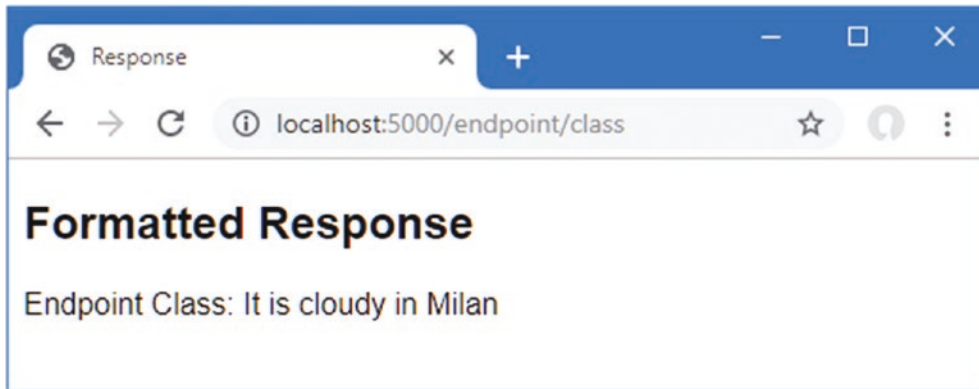


Figure 14-5. Using a service in an endpoint class

Using an Adapter Function

The drawback of using the `HttpContext.RequestServices` method is that the service must be resolved for every request that is routed to the endpoints. As you will learn later in the chapter, there are some services for which this is required because they provide features that are specific to a single request or response. This isn't the case for the `IResponseFormatter` service, where a single object can be used to format multiple responses.

A more elegant approach is to get the service when the endpoint's route is created and not for each request. Listing 14-16 changes the static Endpoint method so that it declares a dependency on the `IResponseFormatter` interface.

Listing 14-16. Defining an Adapter Function in the `WeatherEndpoint.cs` File in the Platform Folder

```
using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;
using Platform.Services;

namespace Platform {
    public class WeatherEndpoint {

        public static async Task Endpoint(HttpContext context,
            IResponseFormatter formatter) {
            await formatter.Format(context, "Endpoint Class: It is cloudy in Milan");
        }
    }
}
```

Add a file called `EndPointExtensions.cs` to the `Platform/Services` folder and add the code shown in Listing 14-17.

Listing 14-17. The Contents of the `EndPointExtensions.cs` File in the Services Folder

```
using Microsoft.AspNetCore.Routing;
using Microsoft.Extensions.DependencyInjection;
using Platform.Services;

namespace Microsoft.AspNetCore.Builder {

    public static class EndPointExtensions {
        public static void MapWeather(this IEndpointRouteBuilder app, string path) {
```

```

        IResponseFormatter formatter =
            app.ServiceProvider.GetService<IResponseFormatter>();
        app.MapGet(path, context => Platform.WeatherEndpoint
            .Endpoint(context, formatter));
    }
}

```

The new file creates an extension method for the `IEndpointRouterBuilder` interface, which is used to create routes in the `Startup` class. The interface defines a `ServiceProvider` property that returns an `IServiceProvider` object through which services can be obtained. The extension method gets the service and uses the `MapGet` method to register a `RequestDelegate` that passes on the `HttpContext` object and the `IResponseFormatter` object to the `WeatherEndpoint.Endpoint` method. In Listing 14-18, I have used the extension method to create the route for the endpoint.

Listing 14-18. Creating a Route in the `Startup.cs` File in the Platform Folder

```

...
app.UseEndpoints(endpoints => {

    //endpoints.MapGet("/endpoint/class", WeatherEndpoint.Endpoint);
    endpoints.MapWeather("/endpoint/class");

    endpoints.MapGet("/endpoint/function", async context => {
        await formatter.Format(context,
            "Endpoint Function: It is sunny in LA");
    });
});
...

```

The `MapWeather` extension method sets up the route and creates the adapter around the endpoint class. To see the result, restart ASP.NET Core and request the `http://localhost:5000/endpoint/class` URL, which will produce the same result as the previous example, shown in Figure 14-5.

Using the Activation Utility Class

I used static methods for endpoint classes in Chapter 13 because it makes them easier to use when creating routes. But for endpoints that require services, it can often be easier to use a class that can be instantiated because it allows for a more generalized approach to handling services. Listing 14-19 revises the endpoint with a constructor and removes the `static` keyword from the `Endpoint` method.

Listing 14-19. Revising the Endpoint in the `WeatherEndpoint.cs` File in the Platform Folder

```

using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;
using Platform.Services;

namespace Platform {
    public class WeatherEndpoint {
        private IResponseFormatter formatter;

        public WeatherEndpoint(IResponseFormatter responseFormatter) {
            formatter = responseFormatter;
        }

        public async Task Endpoint(HttpContext context) {
            await formatter.Format(context, "Endpoint Class: It is cloudy in Milan");
        }
    }
}

```

The most common use of dependency injection in ASP.NET Core applications is in class constructors. Injection through methods, such as performed for middleware classes, is a complex process to re-create, but there are some useful built-in tools that take care of inspecting constructors and resolving dependencies using services, as shown in Listing 14-20.

Listing 14-20. Resolving Dependencies in the EndpointExtensions.cs File in the Services Folder

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Routing;
using Microsoft.Extensions.DependencyInjection;
using Platform.Services;
using System.Reflection;
using System.Threading.Tasks;

namespace Microsoft.AspNetCore.Builder {

    public static class EndpointExtensions {

        public static void MapEndpoint<T> (this IEndpointRouteBuilder app,
            string path, string methodName = "Endpoint") {

            MethodInfo methodInfo = typeof(T).GetMethod(methodName);
            if (methodInfo == null || methodInfo.ReturnType != typeof(Task)) {
                throw new System.Exception("Method cannot be used");
            }
            T endpointInstance =
                ActivatorUtilities.CreateInstance<T>(app.ServiceProvider);
            app.MapGet(path, (RequestDelegate)methodInfo
                .CreateDelegate(typeof(RequestDelegate), endpointInstance));
        }
    }
}
```

The extension method accepts a generic type parameter that specifies the endpoint class that will be used. The other arguments are the path that will be used to create the route and the name of the endpoint class method that processes requests.

A new instance of the endpoint class is created, and a delegate to the specified method is used to create a route. Like any code that uses .NET reflection, the extension method in Listing 14-20 can be difficult to read, but the key statement for the purposes of this chapter is this one:

```
...
T endpointInstance = ActivatorUtilities.CreateInstance<T>(app.ServiceProvider);
...
```

The `ActivatorUtilities` class, defined in the `Microsoft.Extensions.DependencyInjection` namespace, provides methods for instantiating classes that have dependencies declared through their constructor. Table 14-4 shows the most useful `ActivatorUtilities` methods.

Table 14-4. The `ActivatorUtilities` Methods

Name	Description
<code>CreateInstance<T>(services, args)</code>	This method creates a new instance of the class specified by the type parameter, resolving dependencies using the services and additional (optional) arguments.
<code>CreateInstance(services, type, args)</code>	This method creates a new instance of the class specified by the parameter, resolving dependencies using the services and additional (optional) arguments.
<code>GetServiceOrCreateInstance<T>(services, args)</code>	This method returns a service of the specified type, if one is available, or creates a new instance if there is no service.
<code>GetServiceOrCreateInstance(services, type, args)</code>	This method returns a service of the specified type, if one is available, or creates a new instance if there is no service.

Both methods resolve constructor dependencies using services through an `IServiceProvider` object and an optional array of arguments that are used for dependencies that are not services. These methods make it easy to apply dependency injection to custom classes, and the use of the `CreateInstance` method results in an extension method that can create routes with endpoint classes that consume services. Listing 14-21 uses the new extension method to create a route.

Listing 14-21. Creating a Route Using an Extension Method in the `Startup.cs` File in the Platform Folder

```
...
app.UseEndpoints(endpoints => {
    endpoints.MapEndpoint<WeatherEndpoint>("/endpoint/class");

    endpoints.MapGet("/endpoint/function", async context => {
        await formatter.Format(context,
            "Endpoint Function: It is sunny in LA");
    });
});
...
```

This type of extension method makes it easy to work with endpoint classes and provides a similar experience to the `UseMiddleware` method described in Chapter 12. To make sure that requests are routed to the endpoint, restart ASP.NET Core and request the `http://localhost:5000/endpoint/class` URL, which should produce the same response as shown in Figure 14-5.

Using Service Lifecycles

When I created the service in the previous section, I used the `AddSingleton` extension method, like this:

```
...
public void ConfigureServices(IServiceCollection services) {
    services.AddSingleton<IResponseFormatter, HtmlResponseFormatter>();
}
...
```

The `AddSingleton` method produces a service that is instantiated the first time it is used to resolve a dependency and is then reused for each subsequent dependency. This means that any dependency on the `IResponseFormatter` object will be resolved using the same `HtmlResponseFormatter` object.

Singletons are a good way to get started with services, but there are some problems for which they are not suited, so ASP.NET Core supports *scoped services*, which give a lifecycle for the objects that are created to resolve dependencies. Table 14-5 describes the set of methods used to create services. There are versions of these methods that accept types as conventional arguments, as demonstrated in the “Using Unbound Types in Services” section, later in this chapter.

There are versions of the methods in Table 14-5 that have a single type argument, which allows a service to be created that solves the service location problem without addressing the tightly coupled issue. You can see an example of this type of service in Chapter 24, where I share a simple data source that isn’t accessed through an interface.

Table 14-5. The Extension Methods for Creating Services

Name	Description
<code>AddSingleton<T, U>()</code>	This method creates a single object of type <code>U</code> that is used to resolve all dependencies on type <code>T</code> .
<code>AddTransient<T, U>()</code>	This method creates a new object of type <code>U</code> to resolve each dependency on type <code>T</code> .
<code>AddScoped<T, U>()</code>	This method creates a new object of type <code>U</code> that is used to resolve dependencies on <code>T</code> within a single scope, such as request.

Creating Transient Services

The `AddTransient` method does the opposite of the `AddSingleton` method and creates a new instance of the implementation class for every dependency that is resolved. To create a service that will demonstrate the use of service lifecycles, add a file called `GuidService.cs` to the `Platform/Services` folder with the code shown in Listing 14-22.

Listing 14-22. The Contents of the `GuidService.cs` File in the `Services` Folder

```
using System;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;

namespace Platform.Services {

    public class GuidService : IResponseFormatter {
        private Guid guid = Guid.NewGuid();

        public async Task Format(HttpContext context, string content) {
            await context.Response.WriteAsync($"Guid: {guid}\n{content}");
        }
    }
}
```

The `Guid` struct generates a unique identifier, which will make it obvious when a different instance is used to resolve a dependency on the `IResponseFormatter` interface. In Listing 14-23, I have changed the statement that creates the `IResponseFormatter` service to use the `AddTransient` method and the `GuidService` implementation class.

Listing 14-23. Creating a Transient Service in the `Startup.cs` File in the `Platform` Folder

```
...
public void ConfigureServices(IServiceCollection services) {
    services.AddTransient<IResponseFormatter, GuidService>();
}
...
```

If you restart ASP.NET Core and request the `http://localhost:5000/endpoint/class` and `http://localhost:5000/middleware/class` URLs, you will receive the responses shown in Figure 14-6. Each response will be shown with a different GUID value, confirming that transient service objects have been used to resolve the dependencies on the `IResponseFormatter` service for the endpoint and the middleware component. (The nature of GUIDs means you will see different values in your responses. What is important is that you don't see the same value used for both responses.)

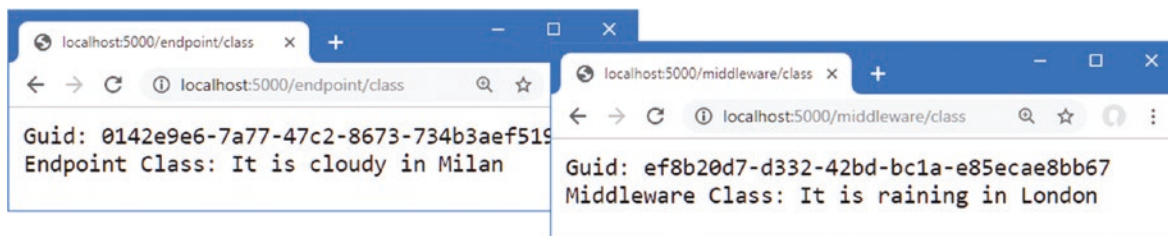


Figure 14-6. Using transient services

Avoiding the Transient Service Reuse Pitfall

The previous example demonstrated that when different service objects are created, the effect is not quite as you might expect, which you can see by clicking the reload buttons. Rather than seeing new GUID values, responses contain the same value, as shown in Figure 14-7.

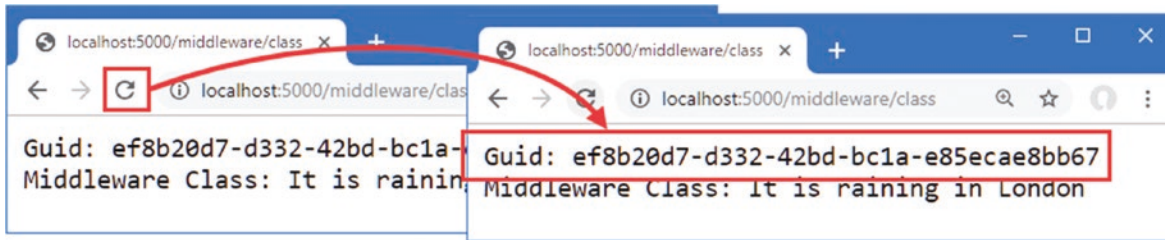


Figure 14-7. The same GUID values appearing in responses

New service objects are created only when dependencies are resolved, not when services are used. The components and endpoints in the example application have their dependencies resolved only when the application starts and the `Startup.Configure` method is invoked. Each receives a separate service object, which is then reused for every request that is processed.

To solve this problem for the middleware component, the dependency for the service can be moved to the `Invoke` method, as shown in Listing 14-24.

Listing 14-24. Moving a Dependency in the `WeatherMiddleware.cs` File in the Platform Folder

```
using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;
using Platform.Services;

namespace Platform {
    public class WeatherMiddleware {
        private RequestDelegate next;
        //private IResponseFormatter formatter;

        public WeatherMiddleware(RequestDelegate nextDelegate) {
            next = nextDelegate;
            //formatter = respFormatter;
        }

        public async Task Invoke(HttpContext context, IResponseFormatter formatter) {
            if (context.Request.Path == "/middleware/class") {
                await formatter.Format(context,
                    "Middleware Class: It is raining in London");
            } else {
                await next(context);
            }
        }
    }
}
```

The ASP.NET Core platform will resolve dependencies declared by the `Invoke` method every time a request is processed, which ensures that a new transient service object is created.

The `ActivatorUtilities` class doesn't deal with resolving dependencies for methods, and ASP.NET Core includes this feature only for middleware components. The simplest way of solving this issue for endpoints is to explicitly request services when each request is handled, which is the approach I used earlier when showing how services are used. It is also possible to enhance the extension method to request services on behalf of an endpoint, as shown in Listing 14-25.

Listing 14-25. Requesting Services in the `EndpointExtensions.cs` File in the Services Folder

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Routing;
using Microsoft.Extensions.DependencyInjection;
using Platform.Services;
```

```

using System.Reflection;
using System.Threading.Tasks;
using System.Linq;

namespace Microsoft.AspNetCore.Builder {

    public static class EndpointExtensions {

        public static void MapEndpoint<T>(this IEndpointRouteBuilder app,
            string path, string methodName = "Endpoint") {

            MethodInfo methodInfo = typeof(T).GetMethod(methodName);
            if (methodInfo == null || methodInfo.ReturnType != typeof(Task)) {
                throw new System.Exception("Method cannot be used");
            }

            T endpointInstance =
                ActivatorUtilities.CreateInstance<T>(app.ServiceProvider);

            ParameterInfo[] methodParams = methodInfo.GetParameters();
            app.MapGet(path, context => (Task)methodInfo.Invoke(endpointInstance,
                methodParams.Select(p => p.ParameterType == typeof(HttpContext)
                ? context
                : app.ServiceProvider.GetService(p.ParameterType)).ToArray());
        }
    }
}

```

The code in Listing 14-25 isn't as efficient as the approach taken by the ASP.NET Core platform for middleware components. All the parameters defined by the method that handles requests are treated as services to be resolved, except for the `HttpContext` parameter. A route is created with a delegate that resolves the services for every request and invokes the method that handles the request. Listing 14-26 revises the `WeatherEndpoint` class to move the dependency on `IResponseFormatter` to the `Endpoint` method so that a new service object will be received for every request.

Listing 14-26. Moving the Dependency in the `WeatherEndpoint.cs` File in the Platform Folder

```

using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;
using Platform.Services;

namespace Platform {
    public class WeatherEndpoint {
        //private IResponseFormatter formatter;

        //public WeatherEndpoint(IResponseFormatter responseFormatter) {
        //    formatter = responseFormatter;
        //}

        public async Task Endpoint(HttpContext context,
            IResponseFormatter formatter) {
            await formatter.Format(context, "Endpoint Class: It is cloudy in Milan");
        }
    }
}

```

The changes in Listing 14-24 to Listing 14-26 ensure that the transient service is resolved for every request, which means that a new `GuidService` object is created and every response contains a unique ID.

For the middleware and endpoint defined as lambda expressions, the service must be obtained as each request is handled because the dependency declared by the `Configure` method parameter is resolved only once, when the request pipeline is configured. Listing 14-27 shows the changes required to get a new service object.

Listing 14-27. Using a Transient Service in the Startup.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing;
using Platform.Services;

namespace Platform {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.AddTransient<IResponseFormatter, GuidService>();
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseRouting();
            app.UseMiddleware<WeatherMiddleware>();

            app.Use(async (context, next) => {
                if (context.Request.Path == "/middleware/function") {
                    IResponseFormatter formatter
                    = app.ApplicationServices.GetService<IResponseFormatter>();
                    await formatter.Format(context,
                        "Middleware Function: It is snowing in Chicago");
                } else {
                    await next();
                }
            });

            app.UseEndpoints(endpoints => {

                endpoints.MapEndpoint<WeatherEndpoint>("/endpoint/class");

                endpoints.MapGet("/endpoint/function", async context => {
                    IResponseFormatter formatter
                    = app.ApplicationServices.GetService<IResponseFormatter>();
                    await formatter.Format(context,
                        "Endpoint Function: It is sunny in LA");
                });
            });
        }
    }
}
```

Restart ASP.NET Core, navigate to any of the four URLs supported by the application (`http://localhost:5000/middleware/class`, `/middleware/function`, `/endpoint/class`, and `/endpoint/function`), and click the browser's reload button. Each time you reload, a new request is sent to ASP.NET Core, and the component or endpoint that handles the request receives a new service object, such that a different GUID is shown in each response, as shown in Figure 14-8.

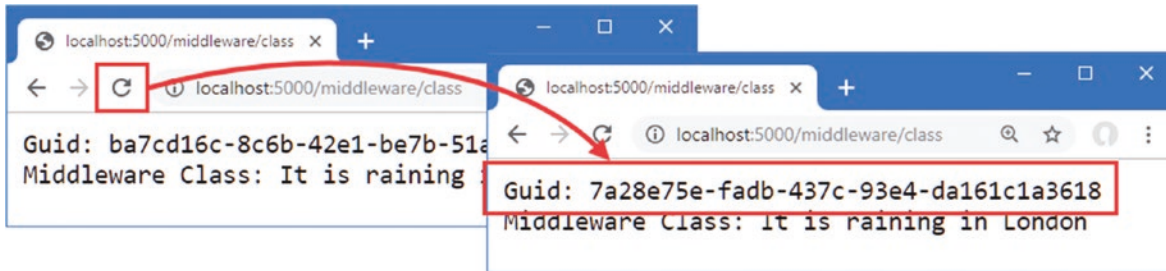


Figure 14-8. Using a transient service

Using Scoped Services

Scoped services strike a balance between singleton and transient services. Within a scope, dependencies are resolved with the same object. A new scope is started for each HTTP request, which means that a service object will be shared by all the components that handle that request. To prepare for a scoped service, Listing 14-28 changes the `WeatherMiddleware` class to declare three dependencies on the same service.

Listing 14-28. Adding Dependencies in the `WeatherMiddleware.cs` File in the Platform Folder

```
using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;
using Platform.Services;

namespace Platform {
    public class WeatherMiddleware {
        private RequestDelegate next;

        public WeatherMiddleware(RequestDelegate nextDelegate) {
            next = nextDelegate;
        }

        public async Task Invoke(HttpContext context, IResponseFormatter formatter1,
            IResponseFormatter formatter2, IResponseFormatter formatter3) {
            if (context.Request.Path == "/middleware/class") {
                await formatter1.Format(context, string.Empty);
                await formatter2.Format(context, string.Empty);
                await formatter3.Format(context, string.Empty);
            } else {
                await next(context);
            }
        }
    }
}
```

Declaring several dependencies on the same service isn't required in real projects, but it is useful for this example because each dependency is resolved independently. Since the `IResponseFormatter` service was created with the `AddTransient` method, each dependency is resolved with a different object. Restart ASP.NET Core and request `http://localhost:5000/middleware/class`, and you will see that a different GUID is used for each of the three messages written to the response, as shown in Figure 14-9. When you reload the browser, a new set of three GUIDs is displayed.

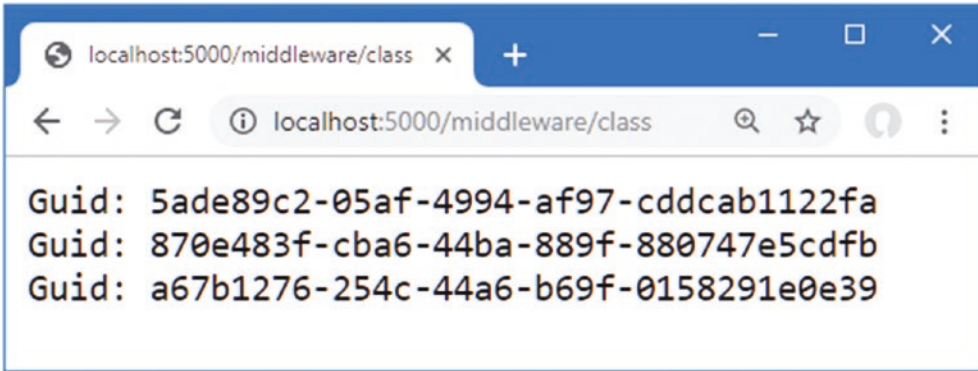


Figure 14-9. Resolving dependencies on a transient service

Listing 14-29 changes the `IResponseFormatter` service to use the scoped lifecycle with the `AddScoped` method.

Tip You can create your own scopes through the `CreateScope` extension method for the `IServiceProvider` interface. The result is an `IServiceProvider` that is associated with a new scope and that will have its own implementation objects for scoped services.

Listing 14-29. Using a Scoped Service in the Startup.cs File in the Platform Folder

```
...
public void ConfigureServices(IServiceCollection services) {
    services.AddScoped<IResponseFormatter, GuidService>();
}
...
```

Restart ASP.NET Core and request `http://localhost:5000/middleware/class` again, and you will see that the same GUID is used to resolve all three dependencies declared by the middleware component, as shown in Figure 14-10. When the browser is reloaded, the HTTP request sent to ASP.NET Core creates a new scope and a new service object.

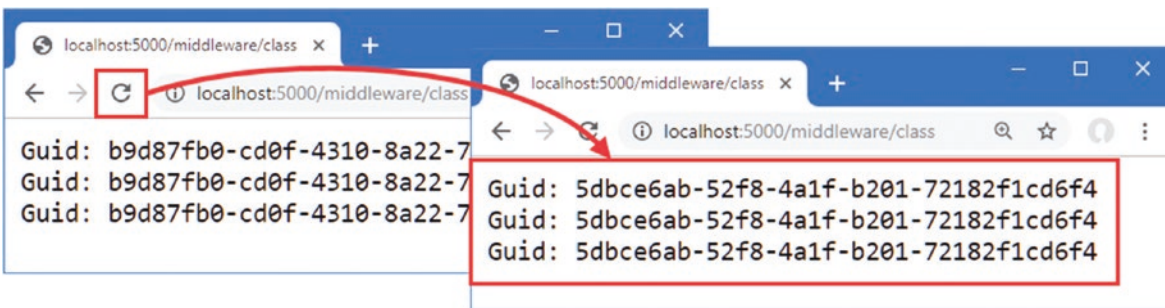


Figure 14-10. Using a scoped service

Avoiding the Scoped Service Validation Pitfall

Service consumers are unaware of the lifecycle that has been selected for singleton and transient services: they declare a dependency or request a service and get the object they require.

Scoped services can be used only within a scope. A new scope is created automatically for each request that was received. Requesting a scoped service outside of a scope causes an exception. To see the problem, request `http://localhost:5000/endpoint/class`, which will generate the exception response shown in Figure 14-11.

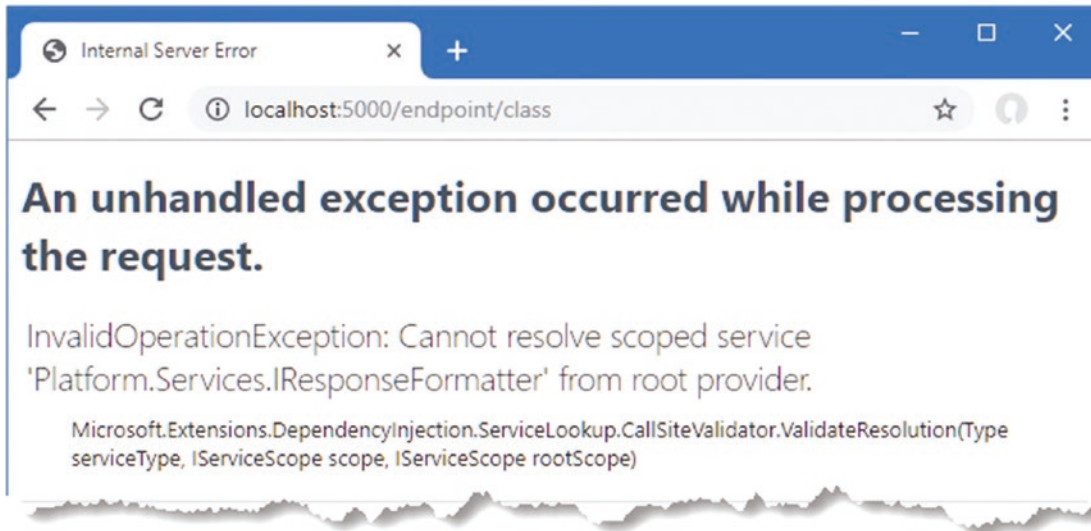


Figure 14-11. Requesting a scoped service

The extension method that configures the endpoint resolves services through an `IServiceProvider` object obtained from the routing middleware, like this:

```
...
app.MapGet(path, context => (Task)methodInfo.Invoke(endpointInstance,
    methodParams.Select(p => p.ParameterType == typeof(HttpContext)
        ? context : app.ServiceProvider.GetService(p.ParameterType)).ToArray()));
...
```

This is known as the *root provider*, and it does not provide access to scoped services to prevent the accidental use of services that are not intended for use outside of a scope.

Accessing Scoped Services Through the Context Object

The `HttpContext` class defines a `RequestServices` property that returns an `IServiceProvider` object that allows access to scoped services, as well as singleton and transient services. This fits well with the most common use of scoped services, which is to use a single service object for each HTTP request. Listing 14-30 revises the endpoint extension method so that dependencies are resolved using the services provided through the `HttpContext`.

Listing 14-30. Using a Scoped Service in the `EndpointExtensions.cs` File in the Services Folder

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Routing;
using Microsoft.Extensions.DependencyInjection;
using Platform.Services;
using System.Reflection;
using System.Threading.Tasks;
using System.Linq;

namespace Microsoft.AspNetCore.Builder {

    public static class EndpointExtensions {

        public static void MapEndpoint<T>(this IEndpointRouteBuilder app,
            string path, string methodName = "Endpoint") {
```



```

        MethodInfo methodInfo = typeof(T).GetMethod(methodName);
        if (methodInfo == null || methodInfo.ReturnType != typeof(Task)) {
            throw new System.Exception("Method cannot be used");
        }

        T endpointInstance =
            ActivatorUtilities.CreateInstance<T>(app.ServiceProvider);

        ParameterInfo[] methodParams = methodInfo.GetParameters();
        app.MapGet(path, context => (Task)methodInfo.Invoke(endpointInstance,
            methodParams.Select(p => p.ParameterType == typeof(HttpContext)
                ? context
                : context.RequestServices.GetService(p.ParameterType).ToArray()));
    }
}

```

Only dependencies that are declared by the method that handles the request are resolved using the `HttpContext`. `RequestServices` property. Services that are declared by an endpoint class constructor are still resolved using the `IEndpointRouteBuilder.ServiceProvider` property, which ensures that endpoints don't use scoped services inappropriately.

Creating New Handlers for Each Request

The problem with the extension method is that it requires endpoint classes to know the lifecycles for the services they depend on. The `WeatherEndpoint` class depends on the `IResponseFormatter` service and must know that a dependency can be declared only through the `Endpoint` method and not the constructor.

To remove the need for this knowledge, a new instance of the endpoint class can be created to handle each request, as shown in Listing 14-31, which allows constructor and method dependencies to be resolved without needing to know which services are scoped.

Listing 14-31. Instantiating Endpoints in the `EndpointExtensions.cs` File in the `Services` Folder

```

using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Routing;
using Microsoft.Extensions.DependencyInjection;
using Platform.Services;
using System.Reflection;
using System.Threading.Tasks;
using System.Linq;

namespace Microsoft.AspNetCore.Builder {

    public static class EndpointExtensions {

        public static void MapEndpoint<T>(this IEndpointRouteBuilder app,
            string path, string methodName = "Endpoint") {

            MethodInfo methodInfo = typeof(T).GetMethod(methodName);
            if (methodInfo == null || methodInfo.ReturnType != typeof(Task)) {
                throw new System.Exception("Method cannot be used");
            }

            ParameterInfo[] methodParams = methodInfo.GetParameters();
            app.MapGet(path, context => {
                T endpointInstance =
                    ActivatorUtilities.CreateInstance<T>(context.RequestServices);
            });
        }
    }
}

```



```

    });
  });
}
}
}

```

Restart ASP.NET Core and request the URLs that target the lambda functions: `http://localhost:5000/endpoint/function` and `/middleware/function`. The scoped service will be obtained without an exception, producing the responses shown in Figure 14-12.

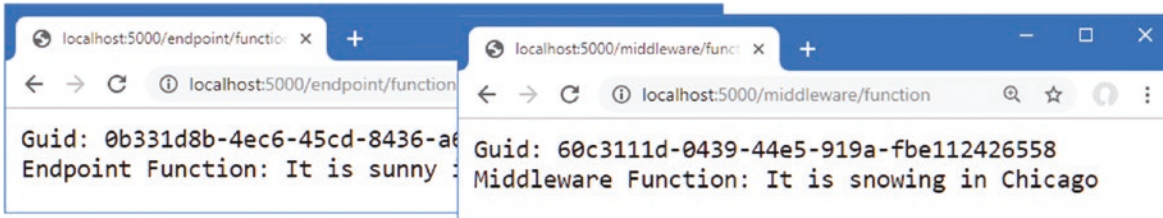


Figure 14-12. Using scoped services in lambda functions

Other Dependency Injection Features

In the sections that follow, I describe some additional features available when using dependency injection. These are not required for all projects, but they are worth understanding because they provide context for how dependency injection works and can be helpful when the standard features are not quite what a project requires.

Creating Dependency Chains

When a class is instantiated to resolve a service dependency, its constructor is inspected, and any dependencies on services are resolved. This allows one service to declare a dependency on another service, creating a chain that is resolved automatically. To demonstrate, add a class file called `TimeStamping.cs` to the `Platform/Services` folder with the code shown in Listing 14-33.

Listing 14-33. The Contents of the `TimeStamping.cs` File in the Services Folder

```

using System;

namespace Platform.Services {

    public interface ITimeStamper {
        string TimeStamp { get; }
    }

    public class DefaultTimeStamper : ITimeStamper {

        public string TimeStamp {
            get => DateTime.Now.ToShortTimeString();
        }
    }
}

```

The class file defines an interface named `ITimeStamper` and an implementation class named `DefaultTimeStamper`. Next, add a file called `TimeResponseFormatter.cs` to the `Platform/Services` folder with the code shown in Listing 14-34.

Listing 14-34. The Contents of the TimeResponseFormatter.cs File in the Services Folder

```

using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;

namespace Platform.Services {
    public class TimeResponseFormatter : IResponseFormatter {
        private ITimeStamper stamper;

        public TimeResponseFormatter(ITimeStamper timeStamper) {
            stamper = timeStamper;
        }

        public async Task Format(HttpContext context, string content) {
            await context.Response.WriteAsync($"{stamper.TimeStamp}: {content}");
        }
    }
}

```

The `TimeResponseFormatter` class is an implementation of the `IResponseFormatter` interface that declares a dependency on the `ITimeStamper` interface with a constructor parameter. Listing 14-35 defines services for both interfaces in the `ConfigureServices` method of the `Startup` class.

■ **Note** Services don't need to have the same lifecycle as their dependencies, but you can end up with odd effects if you mix lifecycles. Lifecycles are applied only when a dependency is resolved, which means that if a scoped service depends on a transient service, for example, then the transient object will behave as though it was assigned the scoped lifecycle.

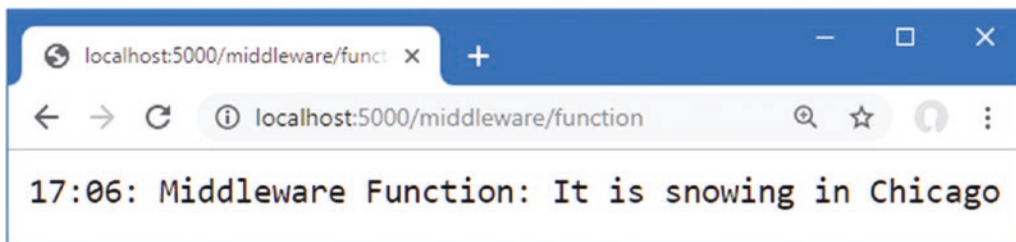
Listing 14-35. Configuring Services in the Startup.cs File in the Platform Folder

```

...
public void ConfigureServices(IServiceCollection services) {
    services.AddScoped<IResponseFormatter, TimeResponseFormatter>();
    services.AddScoped<ITimeStamper, DefaultTimeStamper>();
}
...

```

When a dependency on the `IResponseFormatter` service is resolved, the `TimeResponseFormatter` constructor will be inspected, and its dependency on the `ITimeStamper` service will be detected. A `DefaultTimeStamper` object will be created and injected into the `TimeResponseFormatter` constructor, which allows the original dependency to be resolved. To see the dependency chain in action, restart ASP.NET Core and request `http://localhost:5000/middleware/function`, and you will see the timestamp generated by the `DefaultTimeStamper` class included in the response produced by the `TimeResponseFormatter` class, as shown in Figure 14-13.

**Figure 14-13.** Creating a chain of dependencies

Accessing Services in the ConfigureServices Method

The dependency injection feature is set up by the Platform class before the Startup class is instantiated and the ConfigureServices method is invoked. During the setup process, the services required by the ASP.NET Core platform are created, as well as the basic services provided to applications, such as configuration data, logging, and the environment setting, all of which are described in earlier chapters.

The Startup class can declare dependencies on these services by defining a constructor. When the Startup class is instantiated during application startup, the constructor is inspected, and the dependencies it declares are resolved. The objects injected into the constructor can be assigned to properties and accessed in the ConfigureServices method, which adds the services that are specific to the application. The combined set of services is then available to the Configure method and to the middleware components and endpoints that process requests.

■ **Note** The methods defined by the Startup class are not treated the same way. The Configure method can declare dependencies using parameters, but you will receive an exception if you add parameters to the ConfigureServices method. The technique described in this section is the only way to declare dependencies on services for use in the ConfigureServices method.

The most common use of the Startup constructor is to declare a dependency on the IConfiguration service, which provides access to the application's configuration data, as shown in Listing 14-36. The configuration data is described in detail in Chapter 15.

Listing 14-36. Declaring a Dependency in the Startup.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Routing;
using Microsoft.Extensions.DependencyInjection;
using Platform.Services;
using Microsoft.Extensions.Configuration;

namespace Platform {

    public class Startup {

        public Startup(IConfiguration config) {
            Configuration = config;
        }

        private IConfiguration Configuration;

        public void ConfigureServices(IServiceCollection services) {
            services.AddScoped<IResponseFormatter, TimeResponseFormatter>();
            services.AddScoped<ITimeStamper, DefaultTimeStamper>();
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {

            // ...statements omitted for brevity...
        }
    }
}
```

The IConfiguration service is received through the constructor and assigned to a property named Configuration, which can then be used by the ConfigureServices method. This example doesn't change the responses produced by the application, but you can see how configuration data can be used to configure services in the next section.

Using Service Factory Functions

Factory functions allow you to take control of the way that service implementation objects are created, rather than relying on ASP.NET Core to create instances for you. There are factory versions of the `AddSingleton`, `AddTransient`, and `AddScoped` methods, all of which are used with a function that receives an `IServiceProvider` object and returns an implementation object for the service.

One use for factory functions is to define the implementation class for a service as a configuration setting, which is read through the `IConfiguration` service. This requires the pattern described in the previous section so that the configuration data is accessible in the `ConfigureServices` method. Listing 14-37 adds a factory function for the `IResponseFormatter` service that gets the implementation class from the configuration data.

Listing 14-37. Using a Factory Function in the `Startup.cs` File in the Platform Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing;
using Platform.Services;
using Microsoft.Extensions.Configuration;
using System;

namespace Platform {
    public class Startup {

        public Startup(IConfiguration config) {
            Configuration = config;
        }

        private IConfiguration Configuration;

        public void ConfigureServices(IServiceCollection services) {
            services.AddScoped<IResponseFormatter>(serviceProvider => {
                string typeName = Configuration["services:IResponseFormatter"];
                return (IResponseFormatter)ActivatorUtilities
                    .CreateInstance(serviceProvider, typeName == null
                        ? typeof(GuidService) : Type.GetType(typeName, true));
            });
            services.AddScoped<ITimeStamper, DefaultTimeStamper>();
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {

            // ...statements omitted for brevity...
        }
    }
}
```

The factory function reads a value from the configuration data, which is converted into a type and passed to the `ActivatorUtilities.CreateInstance` method. Listing 14-38 adds a configuration setting to the `appsettings.Development.json` file that selects the `HtmlResponseFormatter` class as the implementation for the `IResponseFormatter` service. The JSON configuration file is described in detail in Chapter 15.

Listing 14-38. Defining a Setting in the appsettings.Development.json File in the Platform Folder

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "services": {
    "IResponseFormatter": "Platform.Services.HtmlResponseFormatter"
  }
}
```

When a dependency on the `IResponseFormatter` service is resolved, the factory function creates an instance of the type specified in the configuration file. Restart ASP.NET Core and request the `http://localhost:5000/middleware/function` URL, which will produce the response shown in Figure 14-14.

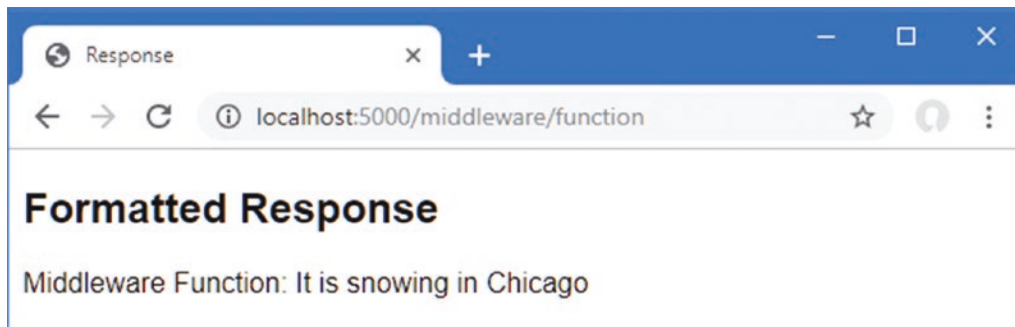


Figure 14-14. Using a service factory

Creating Services with Multiple Implementations

Services can be defined with multiple implementations, which allows a consumer to select an implementation that best suits a specific problem. This is a feature that works best when the service interface provides insight into the capabilities of each implementation class. To provide information about the capabilities of the `IResponseFormatter` implementation classes, add the default property shown in Listing 14-39 to the interface.

Listing 14-39. Adding a Property in the `IResponseFormatter.cs` File in the Services Folder

```
using Microsoft.AspNetCore.Http;
using System.Threading.Tasks;

namespace Platform.Services {
    public interface IResponseFormatter {

        Task Format(HttpContext context, string content);

        public bool RichOutput => false;
    }
}
```

This `RichOutput` property will be `false` for implementation classes that don't override the default value. To ensure there is one implementation that returns `true`, add the property shown in Listing 14-40 to the `HtmlResponseFormatter` class.

Listing 14-40. Overriding a Property in the `HtmlResponseFormatter.cs` File in the Services Folder

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Http;

namespace Platform.Services {
    public class HtmlResponseFormatter : IResponseFormatter {

        public async Task Format(HttpContext context, string content) {
            context.Response.ContentType = "text/html";
            await context.Response.WriteAsync($"@
            <!DOCTYPE html>
            <html lang="en">
            <head><title>Response</title></head>
            <body>
                <h2>Formatted Response</h2>
                <span>{content}</span>
            </body>
            </html>");
        }

        public bool RichOutput => true;
    }
}
```

Listing 14-41 registers multiple implementations for the `IResponseFormatter` service, which is done by making repeated calls to the `Add<lifecycle>` method. The listing also replaces the existing request pipeline with two routes that demonstrate how the service can be used.

Listing 14-41. Defining and Using a Service in the `Startup.cs` File in the Platform Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing;
using Platform.Services;
using Microsoft.Extensions.Configuration;
using System;
using System.Linq;

namespace Platform {
    public class Startup {

        public Startup(IConfiguration config) {
            Configuration = config;
        }

        private IConfiguration Configuration;

        public void ConfigureServices(IServiceCollection services) {
            services.AddScoped<ITimeStamper, DefaultTimeStamper>();
            services.AddScoped<IResponseFormatter, TextResponseFormatter>();
            services.AddScoped<IResponseFormatter, HtmlResponseFormatter>();
            services.AddScoped<IResponseFormatter, GuidService>();
        }
    }
}
```



```

public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
    app.UseDeveloperExceptionPage();
    app.UseRouting();
    app.UseEndpoints(endpoints => {
        endpoints.MapGet("/single", async context => {
            IResponseFormatter formatter = context.RequestServices
                .GetService<IResponseFormatter>();
            await formatter.Format(context, "Single service");
        });

        endpoints.MapGet("/", async context => {
            IResponseFormatter formatter = context.RequestServices
                .GetServices<IResponseFormatter>().First(f => f.RichOutput);
            await formatter.Format(context, "Multiple services");
        });
    });
}

```

The `AddScoped` statements register three services for the `IResponseFormatter` interface, each with a different implementation class. The route for the `/single` URL uses the `IServiceProvider.GetService<T>` method to request a service, like this:

```

...
context.RequestServices.GetService<IResponseFormatter>();
...

```

This is a service consumer that is unaware that there are multiple implementations available. The service is resolved using the most recently registered implementation, which is the `GuidService` class. Restart ASP.NET Core and request `http://localhost:5000/single`, and you will see the output on the left side of Figure 14-15.

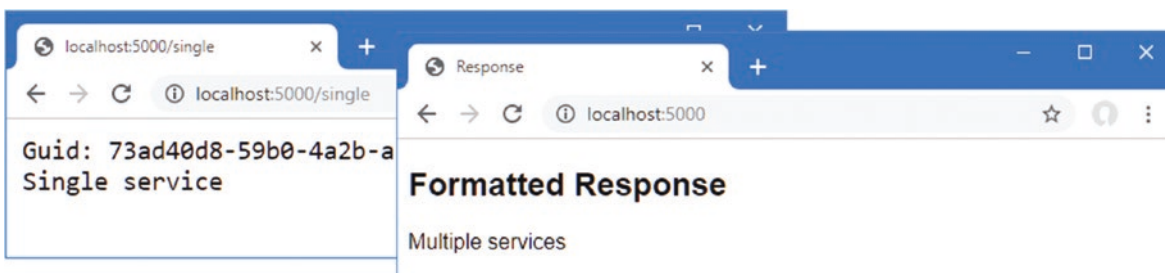


Figure 14-15. Using multiple service implementations

The other endpoint is a service consumer that is aware that multiple implementations may be available and that requests the service using the `IServiceProvider.GetServices<T>` method.

```

...
context.RequestServices.GetServices<IResponseFormatter>().First(f => f.RichOutput);
...

```

This method returns an `IEnumerable<IResponseFormatter>` that enumerates the implementations that are available. These are filtered using the LINQ `First` method to select an implementation whose `RichOutput` property returns `true`. If you request `http://localhost:5000`, you will see the output on the right of Figure 14-15, showing that the endpoint has selected the service implementation that best suits its needs.

Using Unbound Types in Services

Services can be defined with generic type parameters that are bound to specific types when the service is requested, as shown in Listing 14-42.

Listing 14-42. Using an Unbound Type in the Startup.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Routing;
using Platform.Services;
using Microsoft.Extensions.Configuration;
using System;
using System.Linq;
using System.Collections.Generic;

namespace Platform {
    public class Startup {

        public Startup(IConfiguration config) {
            Configuration = config;
        }

        private IConfiguration Configuration;

        public void ConfigureServices(IServiceCollection services) {
            services.AddSingleton(typeof(ICollection<>), typeof(List<>));
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseRouting();
            app.UseEndpoints(endpoints => {
                endpoints.MapGet("/string", async context => {
                    ICollection<string> collection
                    = context.RequestServices.GetService<ICollection<string>>();
                    collection.Add($"Request: { DateTime.Now.ToLongTimeString() }");
                    foreach (string str in collection) {
                        await context.Response.WriteAsync($"String: {str}\n");
                    }
                });

                endpoints.MapGet("/int", async context => {
                    ICollection<int> collection
                    = context.RequestServices.GetService<ICollection<int>>();
                    collection.Add(collection.Count() + 1);
                    foreach (int val in collection) {
                        await context.Response.WriteAsync($"Int: {val}\n");
                    }
                });
            });
        }
    }
}
```

This feature relies on the versions of the `AddSingleton`, `AddScoped`, and `AddTransient` methods that accept types as conventional arguments and cannot be performed using generic type arguments. The service in Listing 14-42 is created with unbound types, like this:

```
...
services.AddSingleton(typeof(ICollection<>), typeof(List<>));
...
```

When a dependency on an `ICollection<T>` service is resolved, a `List<T>` object will be created so that a dependency on `ICollection<string>`, for example, will be resolved using a `List<string>` object. Rather than require separate services for each type, the unbound service allows mappings for all generic types to be created.

The two endpoints in Listing 14-42 request `ICollection<string>` and `ICollection<int>` services, each of which will be resolved with a different `List<T>` object. To target the endpoints, restart ASP.NET Core and request `http://localhost:5000/string` and `http://localhost:5000/int`. The service has been defined as a singleton, which means that the same `List<string>` and `List<int>` objects will be used to resolve all requests for `ICollection<string>` and `ICollection<int>`. Each request adds a new item to the collection, which you can see by reloading the web browser, as shown in Figure 14-16.

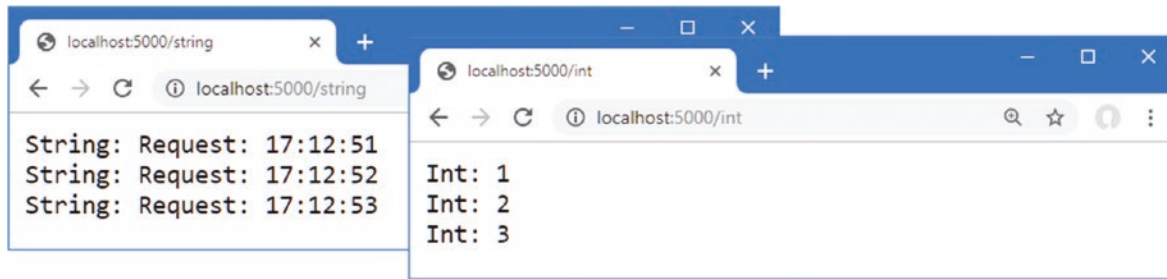


Figure 14-16. Using a singleton service with an unbound type

Summary

In this chapter, I described dependency injection, which is used to define services that are easy to use, easy to change, and easy to consume. I showed you the different ways that services are consumed, explained the different lifecycles that services can be given, and explained some of the less frequently used features such as dependency chains and unbound service types. In the next chapter, I describe the built-in features provided by the ASP.NET Core platform.

CHAPTER 15



Using the Platform Features, Part 1

ASP.NET Core includes a set of built-in services and middleware components that provide features that are commonly required by web applications. In this chapter, I describe three of the most important and widely used features: application configuration, logging, and serving static content. In Chapter 16, I continue to describe the platform features, focusing on the more advanced built-in services and middleware. Table 15-1 puts the chapter in context.

Table 15-1. *Putting Platform Features in Context*

Question	Answer
What are they?	The platform features deal with common web application requirements, such as configuration, logging, static files, sessions, authentication, and database access.
Why are they useful?	Using these features means you don't have to re-create their functionality in your own projects.
How are they used?	The built-in middleware components are added to the request pipeline in the Startup.Configure method using extension methods whose name starts with Use. Services are set up in the Startup.ConfigureServices method using methods that start with Add.
Are there any pitfalls or limitations?	The most common problems relate to the order in which middleware components are added to the request pipeline. Remember that middleware components form a chain along which requests pass, as described in Chapter 12.
Are there any alternatives?	You don't have to use any of the services or middleware components that ASP.NET Core provides.

Table 15-2 summarizes the chapter.

Table 15-2. *Chapter Summary*

Problem	Solution	Listing
Accessing the configuration data	Use the IConfiguration service	4-7
Setting the application environment	Use the launch settings file	11
Determining the application environment	Use the IWebHostEnvironment service	12
Keeping sensitive data outside of the project	Create user secrets	13-19
Logging messages	Use the ILogger<T> service	20-22
Delivering static content	Enable the static content middleware	23-26
Delivering client-side packages	Install the package with LibMan and deliver it with the static content middleware	27-30

Preparing for This Chapter

In this chapter, I continue to use the Platform project created in Chapter 14. To prepare for this chapter, update the Startup class to remove middleware and services, as shown in Listing 15-1.

Listing 15-1. Removing Middleware and Services in the Startup.cs File in the Platform Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Options;

namespace Platform {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
            app.UseDeveloperExceptionPage();
            app.UseRouting();
            app.UseEndpoints(endpoints => {
                endpoints.MapGet("/", async context => {
                    await context.Response.WriteAsync("Hello World!");
                });
            });
        }
    }
}
```

One of the main topics in this chapter is configuration data. Replace the contents of the `appsettings.Development.json` file with the contents of Listing 15-2 to remove the setting added in Chapter 14.

Listing 15-2. Replacing the Contents of the `appsettings.Development.json` File in the Platform Folder

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    }
  }
}
```

Start the application by selecting Start Without Debugging or Run Without Debugging from the Debug menu or by opening a new PowerShell command prompt, navigating to the Platform project folder (which contains the `Platform.csproj` file), and running the command shown in Listing 15-3.

■ **Tip** You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/apress/pro-asp.net-core-3>. See Chapter 1 for how to get help if you have problems running the examples.

Listing 15-3. Starting the ASP.NET Core Runtime

```
dotnet run
```

If the application was started using Visual Studio or Visual Studio Code, a new browser window will open and display the content shown in Figure 15-1. If the application was started from the command line, open a new browser tab and navigate to `http://localhost:5000`; you will see the content shown in Figure 15-1.

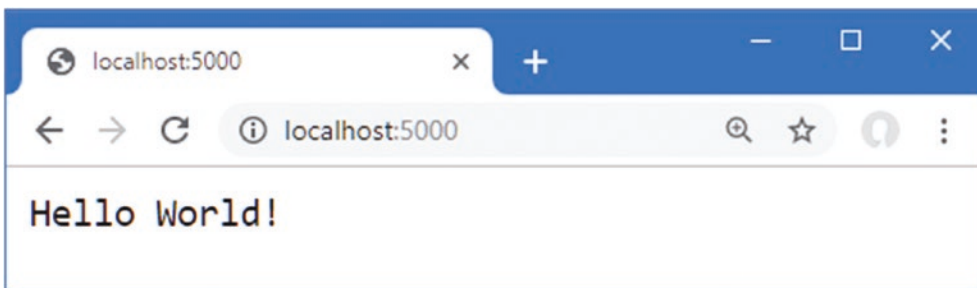


Figure 15-1. Running the example application

Using the Configuration Service

One of the built-in features provided by ASP.NET Core is access to the application's configuration settings, which is then presented as a service.

The main source of configuration data is the `appsettings.json` file. The `appsettings.json` file created by the Empty template contains the following settings:

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

The configuration service will process the JSON configuration file and create nested configuration sections that contain individual settings. For the `appsettings.json` file in the example application, the configuration service will create a `Logging` configuration section that contains a `LogLevel` section. The `LogLevel` section will contain settings for `Default`, `Microsoft`, and `Microsoft.Hosting.Lifetime` settings. There will also be an `AllowedHosts` setting that isn't part of a configuration section and whose value is an asterisk (the `*` character).

The configuration service doesn't understand the meaning of the configuration sections or settings in the `appsettings.json` file and is just responsible for processing the JSON data file and merging the configuration settings with the values obtained from other sources, such as environment variables or command-line arguments. The result is a hierarchical set of configuration properties, as shown in Figure 15-2.

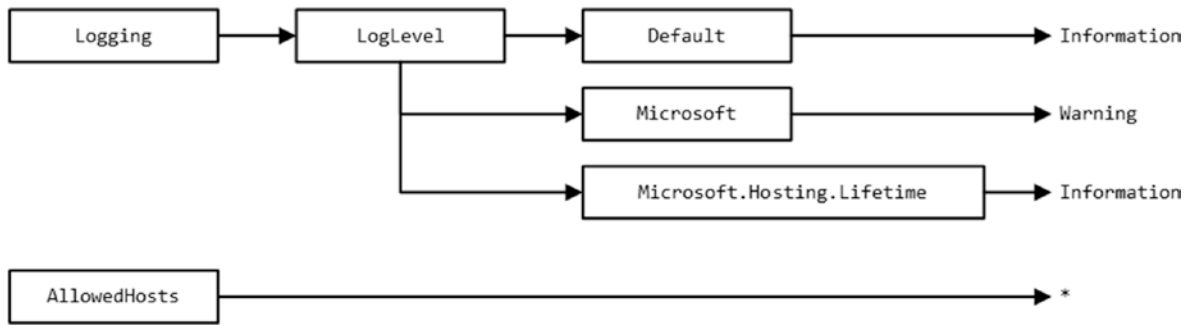


Figure 15-2. The hierarchy of configuration properties in the `appsettings.json` File

Understanding the Environment-Specific Configuration File

Most projects contain more than one JSON configuration file, allowing different settings to be defined for different parts of the development cycle. There are three predefined environments, named `Development`, `Staging`, and `Production`, each of which corresponds to a commonly used phase of development. During startup, the configuration service looks for a JSON file whose name includes the current environment. The default environment is `Development`, which means the configuration service will load the `appsettings.Development.json` file and use its contents to supplement the contents of the main `appsettings.json` file.

■ **Note** The Visual Studio Solution Explorer nests the `appsettings.Development.json` file in the `appsettings.json` item. You can expand the `appsettings.json` file to see and edit the nested entries or click the button at the top of the Solution Explorer that disables the nesting feature.

Here are the configuration settings added to the `appsettings.Development.json` file in Listing 15-2:

```

{
  "Logging": {
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    }
  }
}

```

Where the same setting is defined in both files, the value in the `appsettings.Development.json` file will replace the one in the `appsettings.json` file, which means that the contents of the two JSON files will produce the hierarchy of configuration settings shown in Figure 15-3.

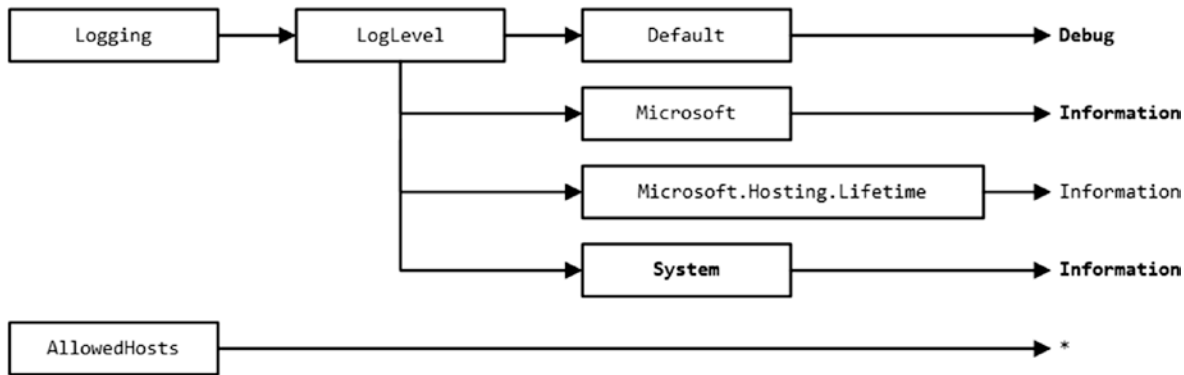


Figure 15-3. Merging JSON configuration settings

The effect of the additional configuration settings is to increase the detail level of logging messages, which I describe in more detail in the “Using the Logging Service” section.

Accessing Configuration Settings

The configuration data is accessed through a service. If you only require the configuration data to configure middleware, then the dependency on the configuration service can be declared using a parameter of the `Configure` method, as shown in Listing 15-4.

Listing 15-4. Accessing Configuration Data in the `Startup.cs` File in the Platform Folder

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Options;
using Microsoft.Extensions.Configuration;

namespace Platform {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env,
            IConfiguration config ) {

            app.UseDeveloperExceptionPage();
            app.UseRouting();

            app.Use(async (context, next) => {
                string defaultDebug = config["Logging:LogLevel:Default"];
                await context.Response
                    .WriteAsync($"The config setting is: {defaultDebug}");
            });
        }
    }
}
  
```



```

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

```

Configuration data is provided through the `IConfiguration` interface; this interface is defined in the `Microsoft.Extensions.Configuration` namespace and provides an API for navigating through the configuration hierarchy and reading configuration settings. To receive the configuration data in the `Startup` class, an `IConfiguration` parameter is added to the `Configure` method. Configuration settings can be read by specifying the path through the configuration sections, like this:

```

...
string defaultDebug = config["Logging:LogLevel:Default"];
...

```

This statement reads the value of the `Default` setting, which is defined in the `LogLevel` section of the `Logging` part of the configuration. The names of the configuration sections and the configuration settings are separated by colons (the `:` character).

The value of the configuration setting read in Listing 15-4 is used to provide a result for a middleware component that handles the `/config` URL. Restart the ASP.NET Core platform by selecting `Degug ▶ Start Without Debugging` or by using `Control+C` at the command prompt and running the command shown in Listing 15-5 in the `Platform` folder.

Listing 15-5. Starting the ASP.NET Core Platform

```
dotnet run
```

Once the runtime has restarted, navigate to the `http://localhost:5000/config` URL, and you will see the value of the configuration setting displayed in the browser tab, as shown in Figure 15-4.

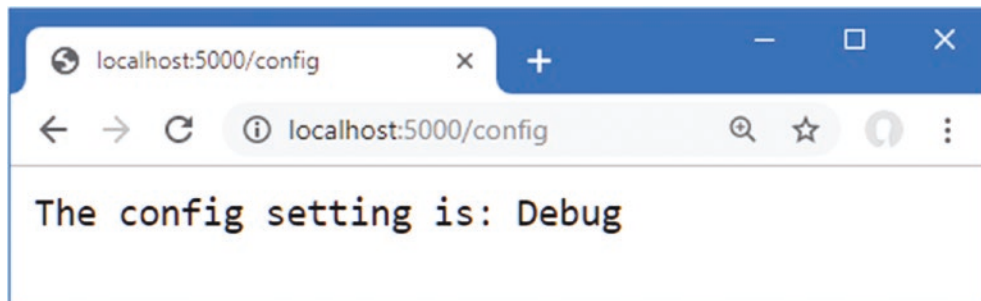


Figure 15-4. Reading configuration data

Using the Configuration Data in Services

A slightly different approach is required to access configuration data in the `ConfigureServices` method, where adding parameters to receive the `IConfiguration` object isn't supported. As explained in Chapter 14, some services are created before the `Startup` class is instantiated, which allows dependencies on them to be declared using a constructor parameter. Listing 15-6 adds a constructor parameter to the `Startup` class that declares a dependency on the `IConfiguration` service and assigns the object used to resolve the dependency to a property that can be accessed in both the `ConfigureServices` and `Configure` methods.

Listing 15-6. Using Configuration Data in the Startup.cs File in the Platform Folder

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Options;
using Microsoft.Extensions.Configuration;

namespace Platform {
    public class Startup {

        public Startup(IConfiguration configService) {
            Configuration = configService;
        }

        private IConfiguration Configuration { get; set; }

        public void ConfigureServices(IServiceCollection services) {

            // configuration data can be accessed here
        }

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

            app.Use(async (context, next) => {
                string defaultDebug = Configuration["Logging:LogLevel:Default"];
                await context.Response
                    .WriteAsync($"The config setting is: {defaultDebug}");
            });

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

```

The constructor assigns the object received through the `IConfiguration` parameter to the property named `Configuration`, which allows the access to the configuration data in both the `Configure` and `ConfigureServices` methods.

Using Configuration Data with the Options Pattern

Configuration data in the `ConfigureServices` method is used with the options pattern that I described in Chapter 14. In that chapter, I showed you how default option values can be changed using a lambda function. An alternative approach is to use configuration settings to set options.

To prepare, add the configuration settings shown in Listing 15-7 to the `appsettings.json` file.

Listing 15-7. Adding Configuration Data in the appsettings.json File in the Platform Folder

```
{
  "Location": {
    "CityName": "Buffalo"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  },
  "AllowedHosts": "*"
}
```

In Listing 15-8, I have used the options pattern to configure the LocationMiddleware component created in Chapter 14, using the configuration data defined in Listing 15-7.

Listing 15-8. Using Configuration Data in the Startup.cs File in the Platform Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Options;
using Microsoft.Extensions.Configuration;

namespace Platform {
    public class Startup {

        public Startup(IConfiguration configService) {
            Configuration = configService;
        }

        private IConfiguration Configuration { get; set; }

        public void ConfigureServices(IServiceCollection services) {
            services.Configure<MessageOptions>(Configuration.GetSection("Location"));
        }

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

            app.UseMiddleware<LocationMiddleware>();

            app.Use(async (context, next) => {
                string defaultDebug = Configuration["Logging:LogLevel:Default"];
                await context.Response
                    .WriteAsync($"The config setting is: {defaultDebug}");
            });
        }
    }
}
```

```

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

```

The section of the configuration data is obtained using the `GetSection` method and passed to the `Configure` method when the options are created. The configuration values in the selected section are inspected and used to replace the default values with the same names in the options class. To see the effect, restart ASP.NET Core and use the browser to navigate to the `http://localhost:5000/location` URL. You will see the results shown in Figure 15-5, where the `CityName` option is taken from the configuration data and the `CountryName` option is taken from the default value in the options class.

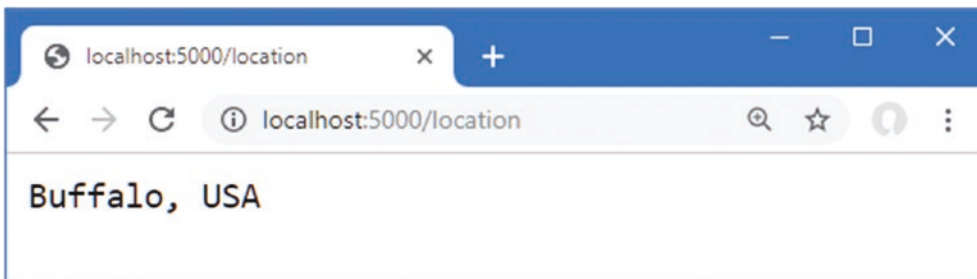


Figure 15-5. Using configuration data in the options pattern

Understanding the Launch Settings File

The `launchSettings.json` file in the `Properties` folder contains the configuration settings for starting the ASP.NET Core platform, including the TCP ports that are used to listen for HTTP and HTTPS requests and the environment used to select the additional JSON configuration files.

■ **Tip** Visual Studio often hides the `Properties` folder by default. Click the `Show All Files` button at the top of the `Solution Explorer` to reveal the folder and the `launchSettings.json` file.

Here is the content added to the `launchSettings.json` file when the `Empty` template is used to create a project:

```

{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:5000",
      "sslPort": 0
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}

```

```

    "Platform": {
      "commandName": "Project",
      "launchBrowser": true,
      "applicationUrl": "http://localhost:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}

```

The `iisSettings` section is used to configure the HTTP and HTTPS ports used when the ASP.NET Core platform is started through IIS Express, which happens when Visual Studio is used.

These settings are supplemented by the `IISExpress` section, which specifies whether a new browser window should be opened when the application starts and which contains `environmentVariables`, which is used to define environment variables that are added to the application's configuration data.

The `Platform` section contains the configuration that is used when the application is started using Visual Studio Code or directly using the `dotnet run` command at the command prompt, and the section specifies the settings for the ASP.NET Core Kestrel HTTP server.

The most important part of both sections is the `environmentVariables` section, which defines the `ASPNETCORE_ENVIRONMENT` setting. During startup, the value of the `ASPNETCORE_ENVIRONMENT` setting is used to select the additional JSON configuration file so that a value of `Development`, for example, will cause the `appsettings.Development.json` file to be loaded.

If you are using Visual Studio Code, `ASPNETCORE_ENVIRONMENT` is set in a different file. Select **Debug** ► **Open Configurations** to open the `launch.json` file in the `.vscode` folder, which is created when a project is edited with Visual Studio Code. Here is the default configuration for the example project, showing the current `ASPNETCORE_ENVIRONMENT` value:

```

{
  "version": "0.2.0",
  "configurations": [
    {
      "name": ".NET Core Launch (web)",
      "type": "coreclr",
      "request": "launch",
      "preLaunchTask": "build",
      "program": "${workspaceFolder}/bin/Debug/netcoreapp3.0/Platform.dll",
      "args": [],
      "cwd": "${workspaceFolder}",
      "stopAtEntry": false,
      "serverReadyAction": {
        "action": "openExternally",
        "pattern": "^\\s*Now listening on:\\s+(https?://\\S+)"
      },
      "env": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      },
      "sourceFileMap": {
        "/Views": "${workspaceFolder}/Views"
      }
    },
    {
      "name": ".NET Core Attach",
      "type": "coreclr",
      "request": "attach",
      "processId": "${command:pickProcess}"
    }
  ]
}

```

To display the value of the `ASPNETCORE_ENVIRONMENT` setting, add the statements to the middleware component that responds to the `/config` URL, as shown in Listing 15-9.

Listing 15-9. Displaying the Configuration Setting in the Startup.cs File in the Platform Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Options;
using Microsoft.Extensions.Configuration;

namespace Platform {
    public class Startup {

        public Startup(IConfiguration configService) {
            Configuration = configService;
        }

        private IConfiguration Configuration { get; set; }

        public void ConfigureServices(IServiceCollection services) {
            services.Configure<MessageOptions>(Configuration.GetSection("Location"));
        }

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

            app.UseMiddleware<LocationMiddleware>();

            app.Use(async (context, next) => {
                string defaultDebug = Configuration["Logging:LogLevel:Default"];
                await context.Response
                    .WriteAsync($"The config setting is: {defaultDebug}");
                string environ = Configuration["ASPNETCORE_ENVIRONMENT"];
                await context.Response
                    .WriteAsync($"The env setting is: {environ}");
            });

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

Restart ASP.NET Core and navigate to `http://localhost:5000/config`, and you will see the value of the `ASPNETCORE_ENVIRONMENT` setting, as shown in Figure 15-6.



Figure 15-6. Displaying the environment configuration setting

To see the effect that the `ASPNETCORE_ENVIRONMENT` setting has on the overall configuration, change the value in the `launchSettings.json` file, as shown in Listing 15-10.

Listing 15-10. Changing the Environment in the `launchSettings.json` File in the Platform/Properties Folder

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:5000",
      "sslPort": 0
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Production"
      }
    },
    "Platform": {
      "commandName": "Project",
      "launchBrowser": true,
      "applicationUrl": "http://localhost:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Production"
      }
    }
  }
}
```

If you are using Visual Studio, you can change the environment variables by selecting **Project** ► **Platform Properties** and navigating to the **Debug** tab. Double-click the value for the `ASPNETCORE_ENVIRONMENT` variable, and you will be able to change the value to **Production**, as shown in Figure 15-7.

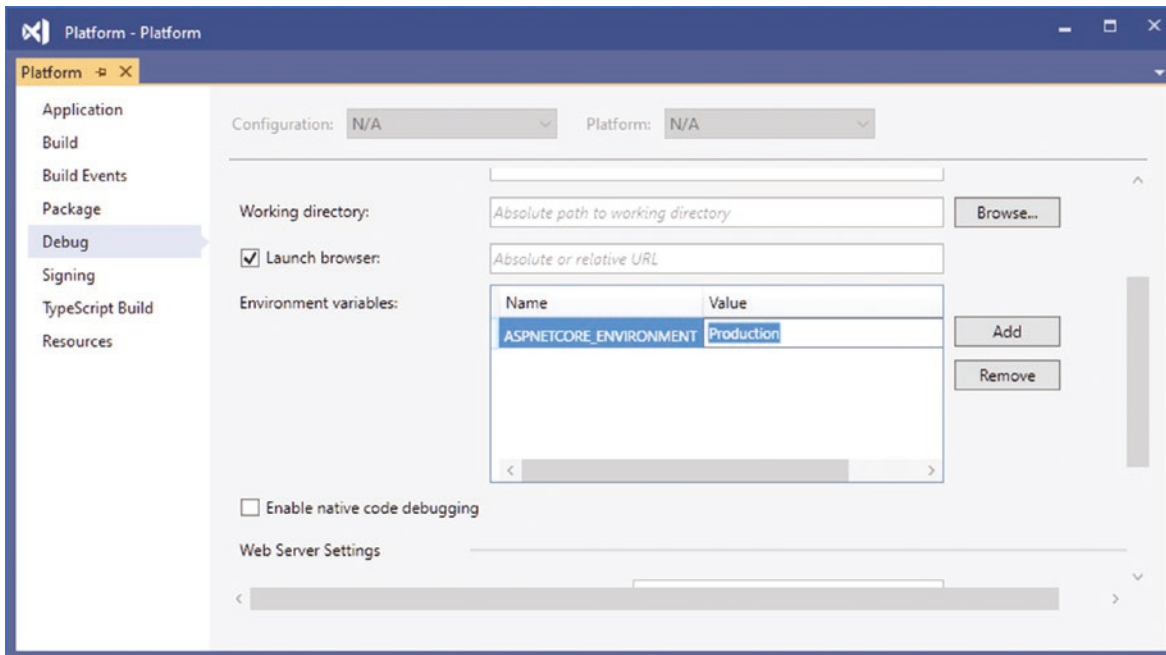


Figure 15-7. Changing an environment variable using Visual Studio

If you are using Visual Studio Code, select **Debug** ► **Open Configurations** and change the value in the `env` section, as shown in Listing 15-11.

Listing 15-11. Changing the Environment in the `launch.json` File in the `Platform/.vscode` Folder

```
{
  "version": "0.2.0",
  "configurations": [
    {
      "name": ".NET Core Launch (web)",
      "type": "coreclr",
      "request": "launch",
      "preLaunchTask": "build",
      "program": "${workspaceFolder}/bin/Debug/netcoreapp3.0/Platform.dll",
      "args": [],
      "cwd": "${workspaceFolder}",
      "stopAtEntry": false,
      "serverReadyAction": {
        "action": "openExternally",
        "pattern": "\\s*Now listening on:\\s+(https?://\\S+)"
      },
      "env": {
        "ASPNETCORE_ENVIRONMENT": "Production"
      },
      "sourceFileMap": {
        "/Views": "${workspaceFolder}/Views"
      }
    }
  ],
}
```



```

    {
      "name": ".NET Core Attach",
      "type": "coreclr",
      "request": "attach",
      "processId": "${command:pickProcess}"
    }
  ]
}

```

Save the changes to the property page or configuration file and restart ASP.NET Core. Navigate to `http://localhost:5000/config`, and you will see the effect of the environment change, as shown in Figure 15-8.

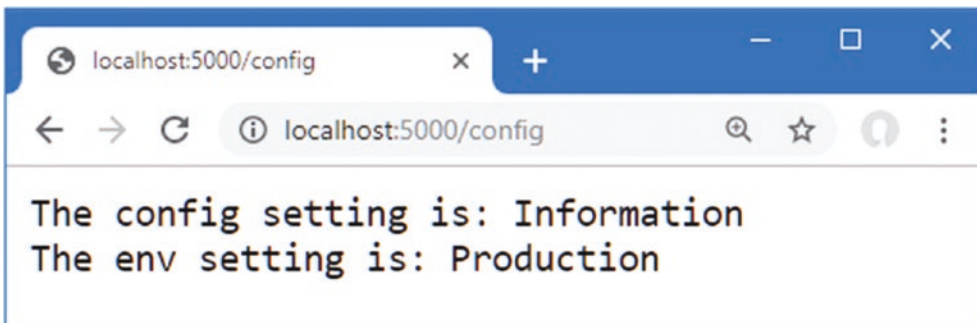


Figure 15-8. The effect of changing the environment configuration setting

Notice that both configuration values displayed in the browser have changed. The `appsettings.Development.json` file is no longer loaded, and there is no `appsettings.Production.json` file in the project, so only the configuration settings in the `appsettings.json` file are used.

Determining the Environment in the Startup Class

The ASP.NET Core platform provides the `IWebHostEnvironment` service for determining the current environment, which avoids the need to get the configuration setting manually. The `IWebHostEnvironment` service defines the methods shown in Table 15-3. These methods are extension methods that are defined in the `Microsoft.Extensions.Hosting` namespace, which must be imported with a `using` statement before they can be used.

Table 15-3. The `IWebHostEnvironment` Extension Methods

Name	Description
<code>IsDevelopment()</code>	This method returns <code>true</code> when the Development environment has been selected.
<code>IsStaging()</code>	This method returns <code>true</code> when the Staging environment has been selected.
<code>IsProduction()</code>	This method returns <code>true</code> when the Production environment has been selected.
<code>IsEnvironment(env)</code>	This method returns <code>true</code> when the environment specified by the argument has been selected.

The `Configure` method in the Startup class already has an `IWebHostEnvironment` parameter, which was added when the project was created using the Empty template in Chapter 14. This is often used with the methods described in Table 15-3 to select the middleware components that are added to the request pipeline. The most common example is to determine whether the `UseDeveloperExceptionPage` method is used, because `UseDeveloperExceptionPage` adds middleware that generates responses that should be used only in development. To ensure that the middleware is used only during development, Listing 15-12 uses the `IWebHostEnvironment` service and the `IsDevelopment` method.

Listing 15-12. Selecting Middleware in the Startup.cs File in the Platform Folder

```

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

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

    app.UseMiddleware<LocationMiddleware>();

    app.Use(async (context, next) => {
        string defaultDebug = Configuration["Logging:LogLevel:Default"];
        await context.Response
            .WriteAsync($"The config setting is: {defaultDebug}");
        string environ = Configuration["ASPNETCORE_ENVIRONMENT"];
        await context.Response
            .WriteAsync($"\\n\\nThe env setting is: {environ}");
    });

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

```

This middleware catches exceptions thrown when a request is processed and displays a detailed stack trace, which is something that should not be seen once the application has been deployed. I demonstrate the error handling middleware in Chapter 16.

■ **Tip** The `IWebHostEnvironment` service also defines properties that can be used to get details of the file location for the application, as demonstrated in Chapter 16.

Storing User Secrets

During development, it is often necessary to use sensitive data to work with the services that an application depends on. This data can include API keys, database connection passwords, or default administration accounts, and it is used both to access services and to reinitialize them to test application changes with a fresh database or user configuration.

If the sensitive data is included in the C# classes or JSON configuration files, it will be checked into the source code version control repository and become visible to all developers and to anyone else who can see the code—which may mean visible to the world for projects that have open repositories or repositories that are poorly secured.

The user secrets service allows sensitive data to be stored in a file that isn't part of the project and won't be checked in to version control, allowing each developer to have their own sensitive data that won't be accidentally exposed through a version control check-in.

Storing User Secrets

The first step is to prepare the file that will be used to store sensitive data. Open a new PowerShell command prompt and run the commands shown in Listing 15-13 in the Platform folder (the folder that contains the Platform.csproj file).

Listing 15-13. Installing the User Secrets Tool Package

```
dotnet tool uninstall --global dotnet-user-secrets
dotnet tool install --global dotnet-user-secrets --version 3.0.0-preview-18579-0056
```

These commands remove the package if it has been installed previously and install the version required by this chapter. Next, run the command shown in Listing 15-14 in the Platform folder.

■ **Note** If you have problems running the commands in this section, then close the PowerShell prompt and open a new one after installing the global tools package.

Listing 15-14. Initializing User Secrets

```
dotnet user-secrets init
```

This command adds an element to the Platform.csproj project file that contains a unique ID for the project that will be associated with the secrets on each developer machine. Next, run the commands shown in Listing 15-15 in the Platform folder.

Listing 15-15. Storing a User Secret

```
dotnet user-secrets set "WebService:Id" "MyAccount"
dotnet user-secrets set "WebService:Key" "MySecret123$"
```

Each secret has a key and a value, and related secrets can be grouped together by using a common prefix, followed by a colon (the : character), followed by the secret name. The commands in Listing 15-15 create related Id and Key secrets that have the WebService prefix.

After each command, you will see a message confirming that a secret has been added to the secret store. To check the secrets for the project, use the command prompt to run the command shown in Listing 15-16 in the Platform folder.

Listing 15-16. Listing the User Secrets

```
dotnet user-secrets list
```

This command produces the following output:

```
WebService:Key = MySecret123$
WebService:Id = MyAccount
```

Behind the scenes, a JSON file has been created in the %APPDATA%\Microsoft\UserSecrets folder (or the ~/.microsoft/usersecrets folder for Linux) to store the secrets. Each project its own folder (whose name corresponds to the unique ID created by the init command in Listing 15-14).

■ **Tip** If you are using Visual Studio, you can create and edit the JSON file directly by right-clicking the project in the Solution Explorer and selecting Manage User Secrets from the popup menu.

Reading User Secrets

User secrets are merged with the normal configuration settings and accessed in the same way. In Listing 15-17, I have added a statement that displays the secrets to the middleware component that handles the /config URL.

Listing 15-17. Using User Secrets in the Startup.cs File in the Platform Folder

```
...
app.Use(async (context, next) => {
    string defaultDebug = Configuration["Logging:LogLevel:Default"];
    await context.Response
        .WriteAsync($"The config setting is: {defaultDebug}");
    string environ = Configuration["ASPNETCORE_ENVIRONMENT"];
    await context.Response
        .WriteAsync($"\\n\\nThe env setting is: {environ}");
    string wsID = Configuration["WebService:Id"];
    string wsKey = Configuration["WebService:Key"];
    await context.Response.WriteAsync($"\\n\\nThe secret ID is: {wsID}");
    await context.Response.WriteAsync($"\\n\\nThe secret Key is: {wsKey}");
});
...
```

User secrets are loaded only when the application is set to the Development environment. Edit the launchSettings.json file to change the environment to Development, as shown in Listing 15-18. (If you prefer, you can use the Visual Studio interface by selecting Project ► Platform Properties.)

Listing 15-18. Changing the Environment in the launchSettings.json File in the Platform/Properties Folder

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:5000",
      "sslPort": 0
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "Platform": {
      "commandName": "Project",
      "launchBrowser": true,
      "applicationUrl": "http://localhost:5000",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}
```

If you are launching the application from within Visual Studio Code, you must also edit the environment by selecting Debug ► Open Configurations and changing the value in the env section, as shown in Listing 15-19.

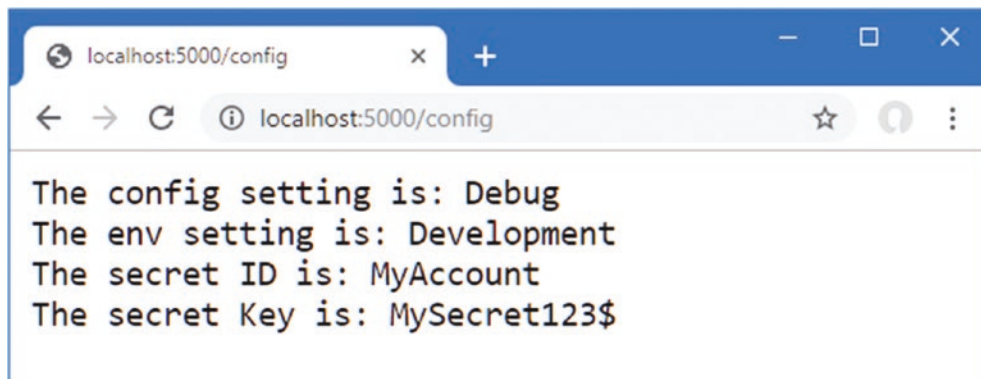
Listing 15-19. Changing the Environment in the launch.json File in the .vscode Folder

```

{
  "version": "0.2.0",
  "configurations": [
    {
      "name": ".NET Core Launch (web)",
      "type": "coreclr",
      "request": "launch",
      "preLaunchTask": "build",
      "program": "${workspaceFolder}/bin/Debug/netcoreapp3.0/Platform.dll",
      "args": [],
      "cwd": "${workspaceFolder}",
      "stopAtEntry": false,
      "serverReadyAction": {
        "action": "openExternally",
        "pattern": "^\\s*Now listening on:\\s+(https?://\\S+)"
      },
      "env": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      },
      "sourceFileMap": {
        "/Views": "${workspaceFolder}/Views"
      }
    },
    {
      "name": ".NET Core Attach",
      "type": "coreclr",
      "request": "attach",
      "processId": "${command:pickProcess}"
    }
  ]
}

```

Save the changes, restart the ASP.NET Core runtime, and request the `http://localhost:5000/config` URL to see the user secrets, as shown in Figure 15-9.

**Figure 15-9.** Displaying user secrets

Using the Logging Service

ASP.NET Core provides a logging service that can be used to record messages that describe the state of the application to track errors, monitor performance, and help diagnose problems.

Log messages are sent to logging providers, which are responsible for forwarding messages to where they can be seen, stored, and processed. There are built-in providers for basic logging, and there is a range of third-party providers available for feeding messages into logging frameworks that allow messages to be collated and analyzed.

Three of the built-in providers are enabled by default: the console provider, the debug provider, and the EventSource provider. The debug provider forwards messages so they can be processed through the `System.Diagnostics.Debug` class, and the EventSource provider forwards messages for event tracing tools, such as PerfView (<https://github.com/Microsoft/perfview>). I use the console provider in this chapter because it is simple and doesn't require any additional configuration to display logging messages.

■ **Tip** You can see the list of providers available and instructions for enabling them at <https://docs.microsoft.com/en-gb/aspnet/core/fundamentals/logging>.

Generating Logging Messages

Logging messages are generated using the unbounded `ILogger<T>` service. Listing 15-20 declares a dependency on the service in the `Startup.Configure` method and removes the middleware from the previous examples for brevity.

Listing 15-20. Generating a Logging Message in the `Startup.cs` File in the Platform Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Options;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging;

namespace Platform {
    public class Startup {

        public Startup(IConfiguration configService) {
            Configuration = configService;
        }

        private IConfiguration Configuration { get; set; }

        public void ConfigureServices(IServiceCollection services) {
            services.Configure<MessageOptions>(Configuration.GetSection("Location"));
        }

        public void Configure(IApplicationBuilder app,
            IWebHostEnvironment env, ILogger<Startup> logger) {

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

```

    app.UseRouting();

    app.UseMiddleware<LocationMiddleware>();

    app.UseEndpoints(endpoints => {
        endpoints.MapGet("/", async context => {
            logger.LogDebug("Response for / started");
            await context.Response.WriteAsync("Hello World!");
            logger.LogDebug("Response for / completed");
        });
    });
}
}
}

```

The logging service groups log messages together based on the category assigned to messages. Log messages are written using the `ILogger<T>` interface, where the generic parameter `T` is used to specify the category. The convention is to use the type of the class that generates the messages as the category type, which is why Listing 15-20 declares a dependency on the unbound service using `Startup` for the type argument, like this:

```

...
public void Configure(IApplicationBuilder app,
    IWebHostEnvironment env, ILogger<Startup> logger) {
...

```

This ensures that log messages generated in the `Configure` method will be assigned the category `Startup`. Log messages are created using the extension methods shown in Table 15-4.

Table 15-4. The `ILogger<T>` Extension Methods

Name	Description
<code>LogTrace</code>	This method generates a Trace-level message, used for low-level debugging during development.
<code>LogDebug</code>	This method generates a Debug-level message, used for low-level debugging during development or production problem resolution.
<code>LogInformation</code>	This method generates an Information-level message, used to provide information about the general state of the application.
<code>LogError</code>	This method generates an Error-level message, used to record exceptions or errors that are not handled by the application.
<code>LogCritical</code>	This method generates a Critical-level message, used to record serious failures.

Log messages are assigned a level that reflects their importance and detail. The levels range from `Trace`, for detailed diagnostics, to `Critical`, for the most important information that requires an immediate response. There are overloaded versions of each method that allow log messages to be generated using strings or exceptions. In Listing 15-20, I used the `LogDebug` method to generate logging messages when a request is handled.

```

...
logger.LogDebug("Response for / started");
...

```

The result is log messages at the `Debug` level that are generated when the response is started and completed. To see the log messages, select `Debug` ► `Start Without Debugging`. If you are using the command line, use `Control+C` to stop the ASP.NET Core runtime and use the command shown in Listing 15-21 in the Platform folder to start it again.

Listing 15-21. Starting the Example Application

```
dotnet run
```

Once the application has started, use a browser tab to request the `http://localhost:5000` URL. If you are using the command line, you will see the logging messages displayed in the output from the application, like this:

```
...
info: Microsoft.AspNetCore.Hosting.Diagnostics[1]
      Request starting HTTP/1.1 GET http://localhost:5000/
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[0]
      Executing endpoint '/' HTTP: GET'
debug: Platform.Startup[0]
      Response for / started
debug: Platform.Startup[0]
      Response for / completed
info: Microsoft.AspNetCore.Routing.EndpointMiddleware[1]
      Executed endpoint '/' HTTP: GET'
info: Microsoft.AspNetCore.Hosting.Diagnostics[2]
      Request finished in 23.02210000000002ms 200
...
```

If you are using Visual Studio, the output from the application is shown in the Output window, which can be selected from the View menu. Select ASP.NET Core Web Server from the drop-down menu to see the output, as shown in Figure 15-10.

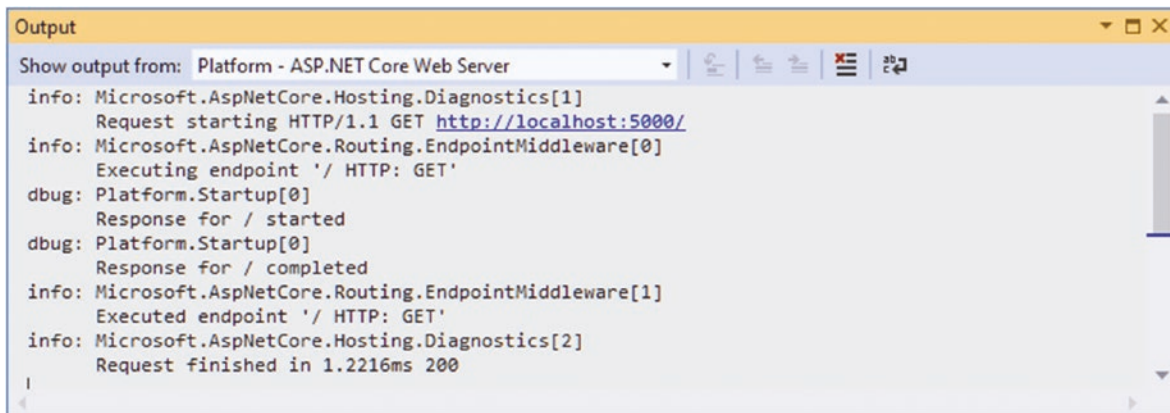


Figure 15-10. Selecting the logging messages in the Visual Studio Output window

If you are using Visual Studio Code, the log messages will be displayed in the Debug Console window, as shown in Figure 15-11, which can be selected from the View menu.

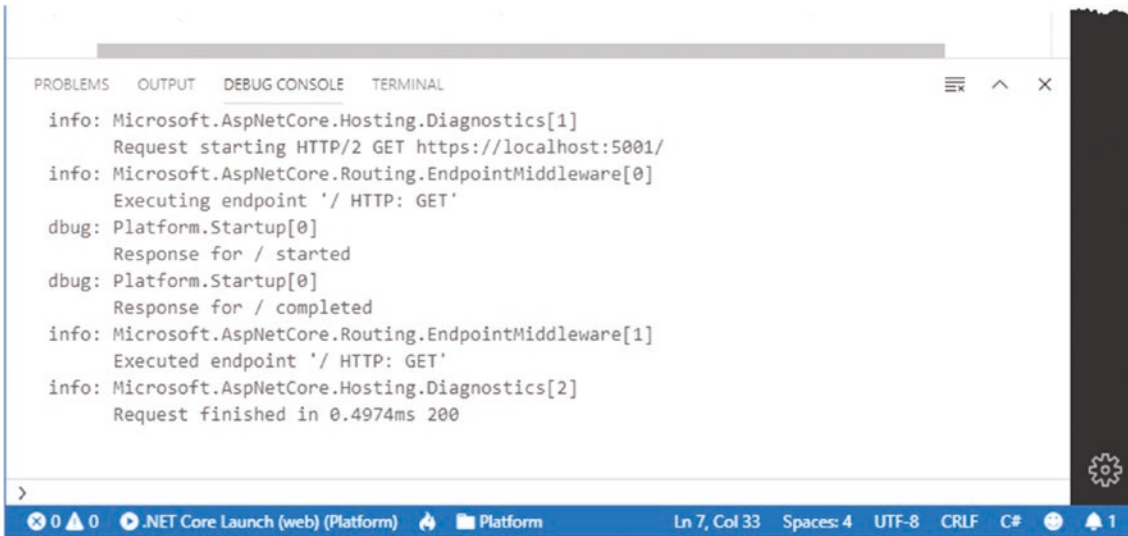


Figure 15-11. Viewing logging messages in the Visual Studio Code Debug Console window

Configuring Minimum Logging Levels

In Chapter 14, I showed you the default contents of the `appsettings.json` and `appsettings.Development.json` files and explained how they are merged to create the application's configuration settings. The settings in the JSON file are used to configure the logging service, which ASP.NET Core provides to record messages about the state of the application.

The `Logging:LogLevel` section of the `appsettings.json` file is used to set the minimum level for logging messages. Log messages that are below the minimum level are discarded. The `appsettings.json` file contains the following levels:

```

...
"Default": "Information",
"Microsoft": "Warning",
"Microsoft.Hosting.Lifetime": "Information",
...
  
```

The category for the log messages—which is set using the generic type argument in Listing 15-20—is used to select a minimum filter level. For the log messages generated by the startup class, for example, the category will be `Platform.Startup`, which means that they can be matched directly by adding a `Platform.Startup` entry to the `appsettings.json` file or indirectly by specifying just the `Platform` namespace. Any category for which there is no minimum log level is matched by the `Default` entry, which is set to `Information`.

It is common to increase the detail of the log messages displayed during development, which is why the levels in the `appsettings.Development.json` file specify more detailed logging levels, like this:

```

...
"Default": "Debug",
"System": "Information",
"Microsoft": "Information"
...
  
```

When the application is configured for the `Development` environment, the default logging level is `Debug`. The levels for the `System` and `Microsoft` categories are set to `Information`, which affects the logging messages generated by ASP.NET Core and the other packages and frameworks provided by Microsoft.

You can tailor the logging levels to focus the log on those parts of the application that are of interest by setting a level to `Trace`, `Debug`, `Information`, `Error`, or `Critical`. Logging messages can be disabled for a category using the `None` value.

In Listing 15-22, I added a new logging level to the `appsettings.Development.json` file that will disable the messages that report the time taken to respond to requests. I also increased the level to `Trace` for the `Microsoft` setting, which will increase the detail displayed for the classes in the ASP.NET Core platform.

■ **Tip** If you are using Visual Studio, you may have to expand the `appsettings.json` item in the Solution Explorer to see the `appsettings.Development.json` file.

Listing 15-22. Selecting a Log Level in the `appsettings.Development.json` File in the Platform Folder

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Trace",
      "Microsoft.AspNetCore.Hosting.Diagnostics": "None"
    }
  }
}
```

Restart ASP.NET Core and request the `http://localhost:5000` URL, and you will see that the amount of detail displayed for each request has changed. The amount of detail will depend on how the application is run. If you are using Visual Studio Code or using the `dotnet run` command, you may see additional details about the generation of HTTP/2 responses. You won't see these messages if you use Visual Studio because the application will be run using IIS Express, which handles the incoming HTTP requests on behalf of the ASP.NET Core platform. There are some messages that will always be seen, however:

```
...
dbug: Microsoft.AspNetCore.Hosting.HostFilteringMiddleware[0]
      Wildcard detected, all requests with hosts will be allowed.
trce: Microsoft.AspNetCore.Hosting.HostFilteringMiddleware[2]
      All hosts are allowed.
...
```

These messages are generated by the Host Filter middleware, which is used to restrict requests based on the `Hosts` header, as described in Chapter 16.

Using Static Content and Client-Side Packages

Most web applications rely on a mix of dynamically generated and static content. The dynamic content is generated by the application based on the user's identity and actions, such as the contents of a shopping cart or the detail of a specific product and is generated fresh for each request. I describe the different ways that dynamic content can be created using ASP.NET Core in Part 3.

Static content doesn't change and is used to provide images, CSS stylesheets, JavaScript files, and anything else on which the application relies but which doesn't have to be generated for every request. The conventional location for static content in an ASP.NET Core project is the `wwwroot` folder.

To prepare static content to use in the examples for this section, create the `Platform/wwwroot` folder and add to it a file called `static.html`, with the content shown in Listing 15-23. You can create the file with the HTML Page template if you are using Visual Studio.

Listing 15-23. The Contents of the static.html File in the wwwroot Folder

```
<!DOCTYPE html>
<html lang="en">
<head>
  <title>Static Content</title>
</head>
<body>
  <h3>This is static content</h3>
</body>
</html>
```

The file contains a basic HTML document with just the basic elements required to display a message in the browser.

Adding the Static Content Middleware

ASP.NET Core provides a middleware component that handles requests for static content, which is added to the request pipeline in Listing 15-24.

Listing 15-24. Adding Middleware in the Startup.cs File in the Platform Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Options;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging;

namespace Platform {
  public class Startup {

    public Startup(IConfiguration configService) {
      Configuration = configService;
    }

    private IConfiguration Configuration { get; set; }

    public void ConfigureServices(IServiceCollection services) {
      services.Configure<MessageOptions>(Configuration.GetSection("Location"));
    }

    public void Configure(IApplicationBuilder app,
      IWebHostEnvironment env, ILogger<Startup> logger) {

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

      app.UseMiddleware<LocationMiddleware>();
    }
  }
}
```


The `FileProvider` and `RequestPath` properties are the most commonly used. The `FileProvider` property is used to select a different location for static content, and the `RequestPath` property is used to specify a URL prefix that denotes requests for static context. Listing 15-25 uses both properties to configure the static file middleware in the `Startup` class.

■ **Tip** There is also a version of the `UseStaticFiles` method that accepts a single string argument, which is used to set the `RequestPath` configuration property. This is a convenient way of adding support for URLs without needing to create an options object.

Listing 15-25. Configuring the Static File Middleware in the `Startup.cs` File in the Platform Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Options;
using Microsoft.Extensions.Configuration;
using Microsoft.Extensions.Logging;
using Microsoft.Extensions.FileProviders;

namespace Platform {
    public class Startup {

        public Startup(IConfiguration configService) {
            Configuration = configService;
        }

        private IConfiguration Configuration { get; set; }

        public void ConfigureServices(IServiceCollection services) {
            services.Configure<MessageOptions>(Configuration.GetSection("Location"));
        }

        public void Configure(IApplicationBuilder app,
            IWebHostEnvironment env, ILogger<Startup> logger) {

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

            app.UseStaticFiles(new StaticFileOptions {
                FileProvider = new
                PhysicalFileProvider($"{env.ContentRootPath}/staticfiles"),
                RequestPath = "/files"
            });

            app.UseRouting();

            app.UseMiddleware<LocationMiddleware>();
        }
    }
}
```


WHAT HAPPENED TO BOWER?

Earlier versions of ASP.NET Core relied on a tool named Bower to manage client-side packages. Bower was widely used in JavaScript development until its creator retired the project and recommended migration to other tools. Microsoft's introduction of Library Manager, which is a .NET Core application, should help provide a consistent experience for dealing with client-side packages and avoid the turbulence that can arise around tools in the dynamic (and sometimes chaotic) world of JavaScript development.

Preparing the Project for Client-Side Packages

Use the command prompt to run the command shown in Listing 15-27, which installs LibMan as a global .NET Core tool (although you may find that LibMan has already been installed).

Listing 15-27. Installing LibMan

```
dotnet tool install --global Microsoft.Web.LibraryManager.Cli --version 2.0.96
```

The next step is to create the LibMan configuration file, which specifies the repository that will be used to get client-side packages and the directory into which packages will be downloaded. Open a PowerShell command prompt and run the command shown in Listing 15-28 in the Platform folder.

Listing 15-28. Initializing LibMan

```
libman init -p cdnjs
```

The `-p` argument specifies the provider that will get packages. I have used `cdnjs`, which selects `cdnjs.com`. The other option is `unpkg`, which selects `unpkg.com`. If you don't have existing experience with package repositories, then you should start with the `cdnjs` option.

The command in Listing 15-28 creates a file named `libman.json` in the Platform folder; the file contains the following settings:

```
...
{
  "version": "1.0",
  "defaultProvider": "cdnjs",
  "libraries": []
}
...
```

If you are using Visual Studio, you can create and edit the `libman.json` file directly by selecting Project ► Manage Client-Side Libraries.

Installing Client-Side Packages

If you are using Visual Studio, you can install client-side packages by right-clicking the Platform project item in the Solution Explorer and selecting Add ► Client-Side Library from the popup menu. Visual Studio will present a simple user interface that allows packages to be located in the repository and installed into the project. As you type into the Library text field, the repository is queried for packages with matching names. Enter `twitter-bootstrap` into the text field, and the popular Bootstrap CSS framework will be selected, as shown in Figure 15-14.

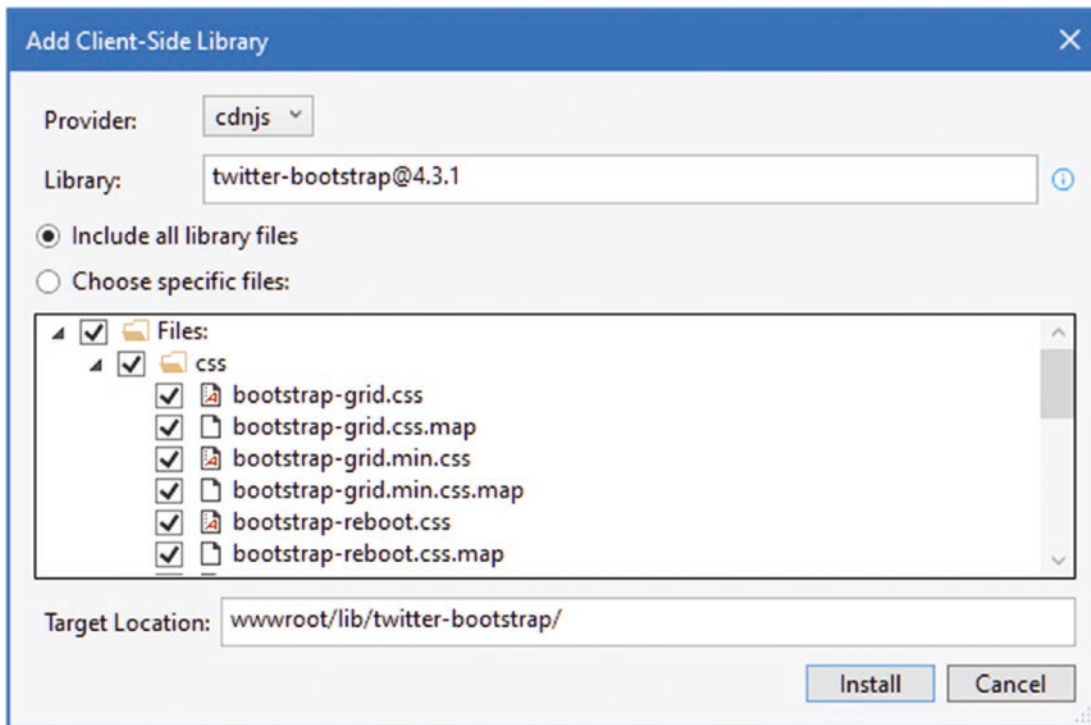


Figure 15-14. Selecting a client-side package in Visual Studio

The latest version of the package is selected, which is 4.3.1 at the time of writing. Click the Install button to download and install the Bootstrap package.

PACKAGE NAMES AND REPOSITORIES

The same package may be known by different names in different repositories. If you are accustomed to using Bower or the Node Package Manager tool (NPM) to install packages, then you may be used to referring to the Bootstrap CSS framework as just `bootstrap`. The CDNJS repository refers to the same package as `twitter-bootstrap`, even though the package and its contents are the same. The best way to figure out what package you require is to go to the repository website and perform a search directly. For the LibMan configuration I use in this chapter, that means going to `cdnjs.com`.

Packages can also be installed from the command line. If you are using Visual Studio Code (or simply prefer the command line, as I do), then run the command shown in Listing 15-29 in the Platform folder to install the Bootstrap package.

Listing 15-29. Installing the Bootstrap Package

```
libman install twitter-bootstrap@4.3.1 -d wwwroot/lib/twitter-bootstrap
```

The required version is separated from the package name by the @ character, and the `-d` argument is used to specify where the package will be installed. The `wwwroot/lib` folder is the conventional location for installing client-side packages in ASP.NET Core projects.

Regardless of which approach you take to install the client-side package, the result will be the `wwwroot/lib/twitter-bootstrap` folder that contains the CSS stylesheets and JavaScript files that are provided by the Bootstrap CSS framework.

Using a Client-Side Package

Once a client-side package has been installed, its files can be referenced by `script` or `link` HTML elements or by using the features provided by the higher-level ASP.NET Core features described in later chapters.

For simplicity in this chapter, Listing 15-30 adds a `link` element to the static HTML file created earlier in this section.

Listing 15-30. Using a Client-Side Package in the `static.html` File in the `Platform/wwwroot` Folder

```
<!DOCTYPE html>
<html lang="en">
<head>
  <link rel="stylesheet" href="/lib/twitter-bootstrap/css/bootstrap.min.css" />
  <title>Static Content</title>
</head>
<body>
  <h3 class="p-2 bg-primary text-white">This is static content</h3>
</body>
</html>
```

Restart ASP.NET Core and request `http://localhost:5000/static.html`. When the browser receives and processes the contents of the `static.html` file, it will encounter the `link` element and send an HTTP request to the ASP.NET Core runtime for the `/lib/twitter-bootstrap/css/bootstrap.min.css` URL. The original static file middleware component added in Listing 15-24 will receive this request, determine that it corresponds to a file in the `wwwroot` folder, and return its contents, providing the browser with the Bootstrap CSS stylesheet. The Bootstrap styles are applied through the classes to which the `h3` element has been assigned, producing the result shown in Figure 15-15.

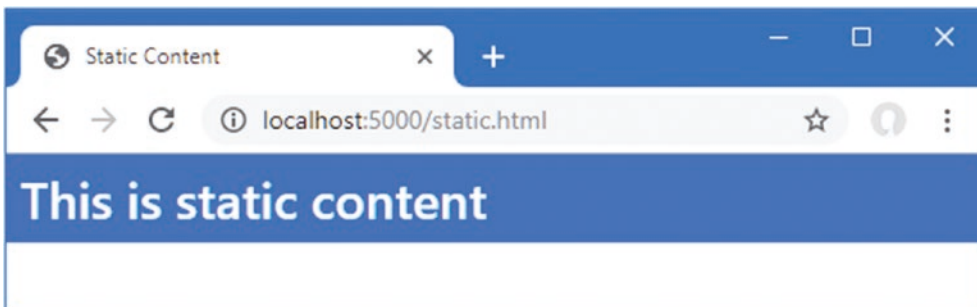


Figure 15-15. Using a client-side package

Summary

In this chapter, I described some of the most important and useful features that ASP.NET Core provides for web applications. I showed you how ASP.NET Core applications are configured, how logging messages are generated and managed, and how to serve static content, including the contents of client-side packages. In the next chapter, I continue to describe the ASP.NET Core platform features.

CHAPTER 16



Using the Platform Features, Part 2

In this chapter, I continue to describe the basic features provided by the ASP.NET Core platform. I explain how cookies are used and how the user's consent for tracking cookies is managed. I describe how sessions provide a robust alternative to basic cookies, how to use and enforce HTTPS requests, how to deal with errors, and how to filter requests based on the Host header. Table 16-1 summarizes the chapter.

Table 16-1. Chapter Summary

Problem	Solution	Listing
Using cookies	Use the context objects to read and write cookies	1-3
Managing cookie consent	Use the consent middleware	4-6
Storing data across requests	Use sessions	7, 8
Securing HTTP requests	Use the HTTPS middleware	9-13
Handling errors	Use the error and status code middleware	14-19
Restricting a request with the host header	Set the AllowedHosts configuration setting	20

Preparing for This Chapter

In this chapter, I continue to use the Platform project from Chapter 15. To prepare for this chapter, replace the contents of the Startup.cs file with the contents of Listing 16-1, which remove the middleware and services from the previous chapter.

Listing 16-1. Replacing the Contents of the Startup.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace Platform {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
        }

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

```

        app.UseEndpoints(endpoints => {
            endpoints.MapFallback(async context =>
                await context.Response.WriteAsync("Hello World!"));
        });
    }
}
}

```

Start the application by selecting Start Without Debugging or Run Without Debugging from the Debug menu or by opening a new PowerShell command prompt, navigating to the folder that contains the Platform.csproj file, and running the command shown in Listing 16-2.

■ **Tip** You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/apress/pro-asp.net-core-3>. See Chapter 1 for how to get help if you have problems running the examples.

Listing 16-2. Starting the ASP.NET Core Runtime

```
dotnet run
```

If the application was started using Visual Studio or Visual Studio Code, a new browser window will open and display the content shown in Figure 16-1. If the application was started from the command line, open a new browser tab and navigate to <http://localhost:5000>; you will see the content shown in Figure 16-1.

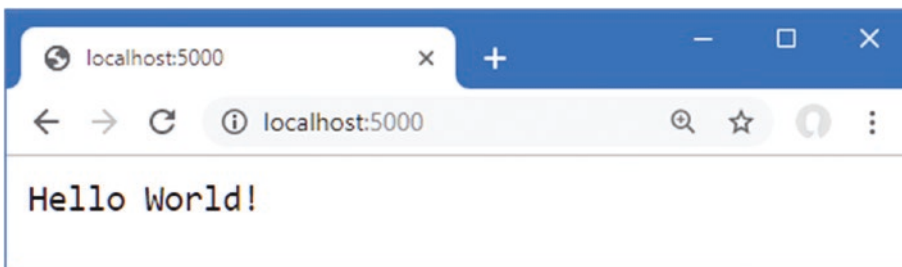


Figure 16-1. Running the example application

Using Cookies

Cookies are small amounts of text added to responses that the browser includes in subsequent requests. Cookies are important for web applications because they allow features to be developed that span a series of HTTP requests, each of which can be identified by the cookies that the browser sends to the server.

ASP.NET Core provides support for working with cookies through the `HttpRequest` and `HttpResponse` objects that are provided to middleware components. To demonstrate, Listing 16-3 changes the routing configuration in the example application to add endpoints that implement a counter.

Listing 16-3. Using Cookies in the Startup.cs File in the Platform Folder

```

using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using System;
using System.Threading.Tasks;

```

```

namespace Platform {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
        }

        public void Configure(IApplicationBuilder app) {
            app.UseDeveloperExceptionPage();
            app.UseRouting();

            app.UseEndpoints(endpoints => {

                endpoints.MapGet("/cookie", async context => {
                    int counter1 =
                        int.Parse(context.Request.Cookies["counter1"] ?? "0") + 1;
                    context.Response.Cookies.Append("counter1", counter1.ToString(),
                        new CookieOptions {
                            MaxAge = TimeSpan.FromMinutes(30)
                        });
                    int counter2 =
                        int.Parse(context.Request.Cookies["counter2"] ?? "0") + 1;
                    context.Response.Cookies.Append("counter2", counter1.ToString(),
                        new CookieOptions {
                            MaxAge = TimeSpan.FromMinutes(30)
                        });
                    await context.Response
                        .WriteAsync($"Counter1: {counter1}, Counter2: {counter2}");
                });

                endpoints.MapGet("clear", context => {
                    context.Response.Cookies.Delete("counter1");
                    context.Response.Cookies.Delete("counter2");
                    context.Response.Redirect("/");
                    return Task.CompletedTask;
                });

                endpoints.MapFallback(async context =>
                    await context.Response.WriteAsync("Hello World!"));
            });
        }
    }
}

```

The new endpoints rely on cookies called `counter1` and `counter2`. When the `/cookie` URL is requested, the middleware looks for the cookies and parses the values to an `int`. If there is no cookie, a fallback zero is used.

```

...
int counter1 = int.Parse(context.Request.Cookies["counter1"] ?? "0") + 1;
...

```

Cookies are accessed through the `HttpRequest.Cookies` property, where the name of the cookie is used as the key. The value retrieved from the cookie is incremented and used to set a cookie in the response, like this:

```

...
context.Response.Cookies.Append("counter1", counter1.ToString(),
    new CookieOptions {
        MaxAge = TimeSpan.FromMinutes(30)
    });
...

```

Cookies are set through the `HttpResponse.Cookies` property, and the `Append` method creates or replaces a cookie in the response. The arguments to the `Append` method are the name of the cookie, its value, and a `CookieOptions` object, which is used to configure the cookie. The `CookieOptions` class defines the properties described in Table 16-2, each of which corresponds to a cookie field.

Table 16-2. *The CookieOptions Properties*

Name	Description
Domain	This property specifies the hosts to which the browser will send the cookie. By default, the cookie will be sent only to the host that created the cookie.
Expires	This property sets the expiry for the cookie.
HttpOnly	When true, this property tells the browser not to include the cookie in requests made by JavaScript code.
IsEssential	This property is used to indicate that a cookie is essential, as described in the “Managing Cookie Consent” section.
MaxAge	This property specifies the number of seconds until the cookie expires. Older browsers do not support cookies with this setting.
Path	This property is used to set a URL path that must be present in the request before the cookie will be sent by the browser.
SameSite	This property is used to specify whether the cookie should be included in cross-site requests. The values are <code>Lax</code> , <code>Strict</code> , and <code>None</code> (which is the default value).
Secure	When true, this property tells the browser to send the cookie using HTTPS only.

The only cookie option set in Listing 16-3 is `MaxAge`, which tells the browser that the cookies expire after 30 minutes. The middleware in Listing 16-3 deletes the cookies when the `/clear` URL is requested, which is done using the `HttpResponse.Cookie.Delete` method, after which the browser is redirected to the `/` URL.

```
...
} else if (context.Request.Path == "/clear") {
    context.Response.Cookies.Delete("counter1");
    context.Response.Cookies.Delete("counter2");
    context.Response.Redirect("/");
}
...
```

Restart ASP.NET Core and navigate to `http://localhost:5000/cookie`. The response will contain cookies that are included in subsequent requests, and the counters will be incremented each time the browser is reloaded, as shown in Figure 16-2. A request for `http://localhost:5000/clear` will delete the cookies, and the counters will be reset.

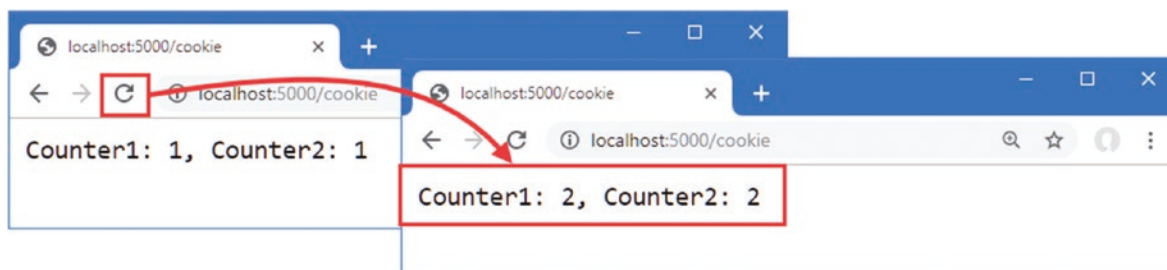


Figure 16-2. *Using a cookie*

Enabling Cookie Consent Checking

The EU General Data Protection Regulation (GDPR) requires the user’s consent before nonessential cookies can be used. ASP.NET Core provides support for obtaining consent and preventing nonessential cookies being sent to the browser when consent has not been granted. The options pattern is used to create a policy for cookies, which is applied by a middleware component, as shown in Listing 16-4.

■ **Caution** Cookie consent is only one part of GDPR. See https://en.wikipedia.org/wiki/General_Data_Protection_Regulation for a good overview of the regulations.

Listing 16-4. Enabling Cookie Consent in the Startup.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using System;
using System.Threading.Tasks;

namespace Platform {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.Configure<CookiePolicyOptions>(opts => {
                opts.CheckConsentNeeded = context => true;
            });
        }

        public void Configure(IApplicationBuilder app) {
            app.UseDeveloperExceptionPage();
            app.UseCookiePolicy();
            app.UseRouting();

            app.UseEndpoints(endpoints => {

                endpoints.MapGet("/cookie", async context => {
                    int counter1 =
                        int.Parse(context.Request.Cookies["counter1"] ?? "0") + 1;
                    context.Response.Cookies.Append("counter1", counter1.ToString(),
                        new CookieOptions {
                            MaxAge = TimeSpan.FromMinutes(30),
                            IsEssential = true
                        });
                    int counter2 =
                        int.Parse(context.Request.Cookies["counter2"] ?? "0") + 1;
                    context.Response.Cookies.Append("counter2", counter1.ToString(),
                        new CookieOptions {
                            MaxAge = TimeSpan.FromMinutes(30)
                        });
                    await context.Response
                        .WriteAsync($"Counter1: {counter1}, Counter2: {counter2}");
                });

                endpoints.MapGet("clear", context => {
                    context.Response.Cookies.Delete("counter1");
                    context.Response.Cookies.Delete("counter2");
                    context.Response.Redirect("/");
                    return Task.CompletedTask;
                });

                endpoints.MapFallback(async context =>
                    await context.Response.WriteAsync("Hello World!"));
            });
        }
    }
}
```

```

    }
  }
}

```

The options pattern is used to configure a `CookiePolicyOptions` object, which sets the overall policy for cookies in the application using the properties described in Table 16-3.

Table 16-3. The `CookiePolicyOptions` Properties

Name	Description
CheckConsentNeeded	This property is assigned a function that receives an <code>HttpContext</code> object and returns true if it represents a request for which cookie consent is required. The function is called for every request, and the default function always returns false.
ConsentCookie	This property returns an object that is used to configure the cookie sent to the browser to record the user's cookie consent.
HttpOnly	This property sets the default value for the <code>HttpOnly</code> property, as described in Table 16-2.
MinimumSameSitePolicy	This property sets the lowest level of security for the <code>SameSite</code> property, as described in Table 16-2.
Secure	This property sets the default value for the <code>Secure</code> property, as described in Table 16-2.

To enable consent checking, I assigned a new function to the `CheckConsentNeeded` property that always returns true. The function is called for every request that ASP.NET Core receives, which means that sophisticated rules can be defined to select the requests for which consent is required. For this application, I have taken the most cautious approach and required consent for all requests.

The middleware that enforces the cookie policy is added to the request pipeline using the `UseCookiePolicy` method. The result is that only cookies whose `IsEssential` property is true will be added to responses. Listing 16-4 sets the `IsEssential` property on `cookie1` only, and you can see the effect by restarting ASP.NET Core, requesting `http://localhost:5000/cookie`, and reloading the browser. Only the counter whose cookie is marked as essential updates, as shown in Figure 16-3.

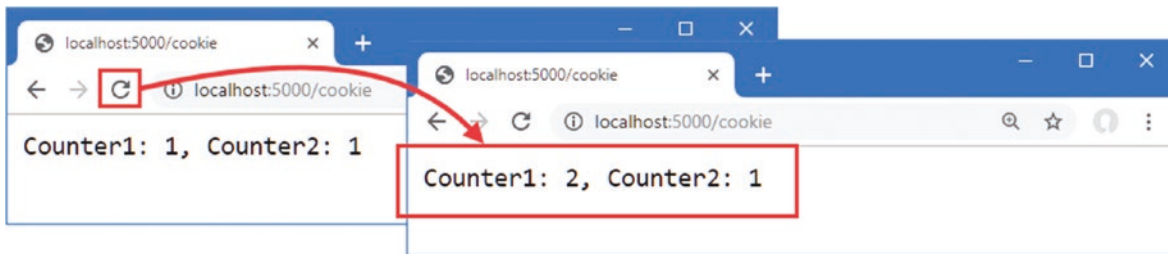


Figure 16-3. Using cookie consent

Managing Cookie Consent

Unless the user has given consent, only cookies that are essential to the core features of the web application are allowed. Consent is managed through a *request feature*, which provides middleware components with access to the implementation details of how requests and responses are handled by ASP.NET Core. Features are accessed through the `HttpRequest.Features` property, and each feature is represented by an interface whose properties and methods deal with one aspect of low-level request handling.

Features deal with aspects of request handling that rarely need to be altered, such as the structure of responses. The exception is the management of cookie consent, which is handled through the `ITrackingConsentFeature` interface, which defines the methods and properties described in Table 16-4.

Table 16-4. The *ITrackingConsentFeature* Members

Name	Description
CanTrack	This property returns true if nonessential cookies can be added to the current request, either because the user has given consent or because consent is not required.
CreateConsentCookie()	This method returns a cookie that can be used by JavaScript clients to indicate consent.
GrantConsent()	Calling this method adds a cookie to the response that grants consent for nonessential cookies.
HasConsent	This property returns true if the user has given consent for nonessential cookies.
IsConsentNeeded	This property returns true if consent for nonessential cookies is required for the current request.
WithdrawConsent()	This method deletes the consent cookie.

To deal with consent, add a class file named `ConsentMiddleware.cs` to the `Platform` folder and the code shown in Listing 16-5. Managing cookie consent can be done using lambda expressions, but I have used a class in this example to keep the `Configure` method uncluttered.

Listing 16-5. The Contents of the `ConsentMiddleware.cs` File in the `Platform` Folder

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Http.Features;
using System.Threading.Tasks;

namespace Platform {
    public class ConsentMiddleware {
        private RequestDelegate next;

        public ConsentMiddleware(RequestDelegate nextDelgate) {
            next = nextDelgate;
        }

        public async Task Invoke(HttpContext context) {
            if (context.Request.Path == "/consent") {
                ITrackingConsentFeature consentFeature
                    = context.Features.Get<ITrackingConsentFeature>();
                if (!consentFeature.HasConsent) {
                    consentFeature.GrantConsent();
                } else {
                    consentFeature.WithdrawConsent();
                }
                await context.Response.WriteAsync(consentFeature.HasConsent
                    ? "Consent Granted \n" : "Consent Withdrawn\n");
            }
            await next(context);
        }
    }
}
```

Request features are obtained using the `Get` method, where the generic type argument specifies the feature interface that is required, like this:

```
...
ITrackingConsentFeature consentFeature
    = context.Features.Get<ITrackingConsentFeature>();
...
```


Using the properties and methods described in Table 16-4, the new middleware component responds to the /consent URL to determine and change the cookie consent. Listing 16-6 adds the new middleware to the request pipeline.

Listing 16-6. Adding Middleware in the Startup.cs File in the Platform Folder

```
...
public void Configure(IApplicationBuilder app) {
    app.UseDeveloperExceptionPage();
    app.UseCookiePolicy();
    app.UseMiddleware<ConsentMiddleware>();
    app.UseRouting();

    // ...statements omitted for brevity...
}
...
```

To see the effect, restart ASP.NET Core and request `http://localhost:5000/consent` and then `http://localhost:5000/cookie`. When consent is granted, nonessential cookies are allowed, and both the counters in the example will work, as shown in Figure 16-4. Repeat the process to withdraw consent, and you will find that only the counter whose cookie has been denoted as essential works.

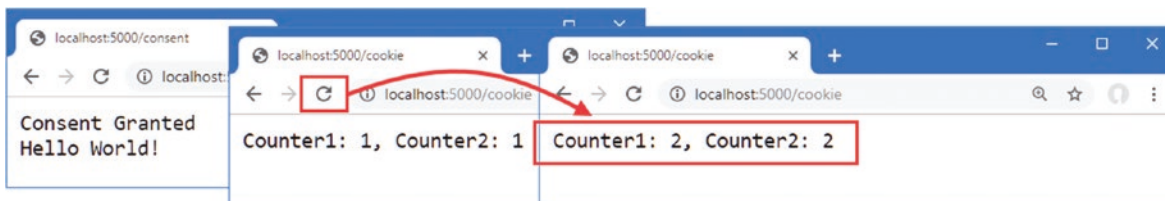


Figure 16-4. Managing cookie consent

Using Sessions

The example in the previous section used cookies to store the application's state data, providing the middleware component with the data required. The problem with this approach is that the contents of the cookie are stored at the client, where it can be manipulated and used to alter the behavior of the application.

A better approach is to use the ASP.NET Core session feature. The session middleware adds a cookie to responses, which allows related requests to be identified and which is also associated with data stored at the server.

When a request containing the session cookie is received, the session middleware component retrieves the server-side data associated with the session and makes it available to other middleware components through the `HttpContext` object. Using sessions means that the application's data remains at the server and only the identifier for the session is sent to the browser.

Configuring the Session Service and Middleware

Setting up sessions requires configuring services and adding a middleware component to the request pipeline. Listing 16-7 adds the statements to the Startup class to set up sessions for the example application and removes the endpoints from the previous section.

Listing 16-7. Configuring Sessions in the Startup.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using System;
using System.Threading.Tasks;
```

```

namespace Platform {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.Configure<CookiePolicyOptions>(opts => {
                opts.CheckConsentNeeded = context => true;
            });

            services.AddDistributedMemoryCache();

            services.AddSession(options => {
                options.IdleTimeout = TimeSpan.FromMinutes(30);
                options.Cookie.IsEssential = true;
            });
        }

        public void Configure(IApplicationBuilder app) {
            app.UseDeveloperExceptionPage();
            app.UseCookiePolicy();
            app.UseMiddleware<ConsentMiddleware>();
            app.UseSession();
            app.UseRouting();

            app.UseEndpoints(endpoints => {
                endpoints.MapFallback(async context =>
                    await context.Response.WriteAsync("Hello World!"));
            });
        }
    }
}

```

When you use sessions, you must decide how to store the associated data. ASP.NET Core provides three options for session data storage, each of which has its own method to register a service in the `ConfigureServices` method of the `Startup` class, as described in Table 16-5.

Table 16-5. *The Session Storage Methods*

Name	Description
<code>AddDistributedMemoryCache</code>	This method sets up an in-memory cache. Despite the name, the cache is not distributed and is responsible only for storing data for the instance of the ASP.NET Core runtime where it is created.
<code>AddDistributedSqlServerCache</code>	This method sets up a cache that stores data in SQL Server and is available when the <code>Microsoft.Extensions.Caching.SqlServer</code> package is installed. This cache is used in Chapter 17.
<code>AddStackExchangeRedisCache</code>	This method sets up a Redis cache and is available when the <code>Microsoft.Extensions.Caching.Redis</code> package is installed.

Caching is described in detail in Chapter 17, but for this chapter, I used the in-memory cache, which is set up in the `ConfigureServices` method:

```

...
services.AddDistributedMemoryCache();
...

```

Despite its name, the cache service created by the `AddDistributedMemoryCache` method isn't distributed and stores the session data for a single instance of the ASP.NET Core runtime. If you scale an application by deploying multiple instances of the runtime, then you should use one of the other caches, such as the SQL Server cache, which is demonstrated in Chapter 17.

The next step is to use the options pattern to configure the session middleware, like this:

```
...
services.AddSession(options => {
    options.IdleTimeout = TimeSpan.FromMinutes(30);
    options.Cookie.IsEssential = true;
});
...
```

Table 16-6 shows that the options class for sessions is `SessionOptions` and describes the key properties it defines.

Table 16-6. Properties Defined by the `SessionOptions` Class

Name	Description
<code>Cookie</code>	This property is used to configure the session cookie.
<code>IdleTimeout</code>	This property is used to configure the time span after which a session expires.

The `Cookie` property returns an object that can be used to configure the session cookie. Table 16-7 describes the most useful cookie configuration properties for session data.

Table 16-7. Cookie Configuration Properties

Name	Description
<code>HttpOnly</code>	This property specifies whether the browser will prevent the cookie from being included in HTTP requests sent by JavaScript code. This property should be set to <code>true</code> for projects that use a JavaScript application whose requests should be included in the session. The default value is <code>true</code> .
<code>IsEssential</code>	This property specifies whether the cookie is required for the application to function and should be used even when the user has specified that they don't want the application to use cookies. The default value is <code>false</code> . See the "Managing Cookie Consent" section for more details.
<code>SecurityPolicy</code>	This property sets the security policy for the cookie, using a value from the <code>CookieSecurePolicy</code> enum. The values are <code>Always</code> (which restricts the cookie to HTTPS requests), <code>SameAsRequest</code> (which restricts the cookie to HTTPS if the original request was made using HTTPS) and <code>None</code> (which allows the cookie to be used on HTTP and HTTPS requests). The default value is <code>None</code> .

The options set in Listing 16-7 allow the session cookie to be included in requests started by JavaScript and flag the cookie as essential so that it will be used even when the user has expressed a preference not to use cookies (see the "Managing Cookie Consent" section for more details about essential cookies). The `IdleTimeout` option has been set so that sessions expire if no request containing the sessions cookie is received for 30 minutes.

■ **Caution** The session cookie isn't denoted as essential by default, which can cause problems when cookie consent is used. Listing 16-7 sets the `IsEssential` property to `true` to ensure that sessions always work. If you find sessions don't work as expected, then this is the likely cause, and you must either set `IsEssential` to `true` or adapt your application to deal with users who don't grant consent and won't accept session cookies.

The final step is to add the session middleware component to the request pipeline, which is done with the `UseSession` method. When the middleware processes a request that contains a session cookie, it retrieves the session data from the cache and makes it available through the `HttpContext` object, before passing the request along the request pipeline and providing it to other middleware components. When a request arrives without a session cookie, a new session is started, and a cookie is added to the response so that subsequent requests can be identified as being part of the session.

Using Session Data

The session middleware provides access to details of the session associated with a request through the `Session` property of the `HttpContext` object. The `Session` property returns an object that implements the `ISession` interface, which provides the methods shown in Table 16-8 for accessing session data.

Table 16-8. Useful `ISession` Methods and Extension Methods

Name	Description
<code>Clear()</code>	This method removes all the data in the session.
<code>CommitAsync()</code>	This asynchronous method commits changed session data to the cache.
<code>GetString(key)</code>	This method retrieves a string value using the specified key.
<code>GetInt32(key)</code>	This method retrieves an integer value using the specified key.
<code>Id</code>	This property returns the unique identifier for the session.
<code>IsAvailable</code>	This returns <code>true</code> when the session data has been loaded.
<code>Keys</code>	This enumerates the keys for the session data items.
<code>Remove(key)</code>	This method removes the value associated with the specified key.
<code>SetString(key, val)</code>	This method stores a string using the specified key.
<code>SetInt32(key, val)</code>	This method stores an integer using the specified key.

Session data is stored in key/value pairs, where the keys are strings and the values are strings or integers. This simple data structure allows session data to be stored easily by each of the caches listed in Table 16-6. Applications that need to store more complex data can use serialization, which is the approach I took for the `SportsStore`. Listing 16-8 uses session data to re-create the counter example.

Listing 16-8. Using Session Data in the `Startup.cs` File in the Platform Folder

```
...
public void Configure(IApplicationBuilder app) {
    app.UseDeveloperExceptionPage();
    app.UseCookiePolicy();
    app.UseMiddleware<ConsentMiddleware>();
    app.UseSession();
    app.UseRouting();

    app.UseEndpoints(endpoints => {

        endpoints.MapGet("/cookie", async context => {
            int counter1 = (context.Session.GetInt32("counter1") ?? 0) + 1;
            int counter2 = (context.Session.GetInt32("counter2") ?? 0) + 1;
            context.Session.SetInt32("counter1", counter1);
            context.Session.SetInt32("counter2", counter2);
            await context.Session.CommitAsync();
            await context.Response
                .WriteAsync($"Counter1: {counter1}, Counter2: {counter2}");
        });

        endpoints.MapFallback(async context =>
            await context.Response.WriteAsync("Hello World!"));
    });
}
...
```

The `GetInt32` method is used to read the values associated with the keys `counter1` and `counter2`. If this is the first request in a session, no value will be available, and the null coalescing operator is used to provide an initial value. The value is incremented and then stored using the `SetInt32` method and used to generate a simple result for the client.

The use of the `CommitAsync` method is optional, but it is good practice to use it because it will throw an exception if the session data can't be stored in the cache. By default, no error is reported if there are caching problems, which can lead to unpredictable and confusing behavior.

All changes to the session data must be made before the response is sent to the client, which is why I read, update, and store the session data before calling the `Response.WriteAsync` method in Listing 16-8.

Notice that the new statements in Listing 16-8 do not have to deal with the session cookie, detect expired sessions, or load the session data from the cache. All this work is done automatically by the session middleware, which presents the results through the `HttpContext.Session` property. One consequence of this approach is that the `HttpContext.Session` property is not populated with data until after the session middleware has processed a request, which means that you should attempt to access session data only in middleware or endpoints that are added to the request pipeline after the `UseSession` method is called.

Restart ASP.NET Core and navigate to the `http://localhost:5000/cookie` URL, and you will see the value of the counter. Reload the browser, and the counter values will be incremented, as shown in Figure 16-5. The sessions and session data will be lost when ASP.NET Core is stopped because I chose the in-memory cache. The other storage options operate outside of the ASP.NET Core runtime and survive application restarts.

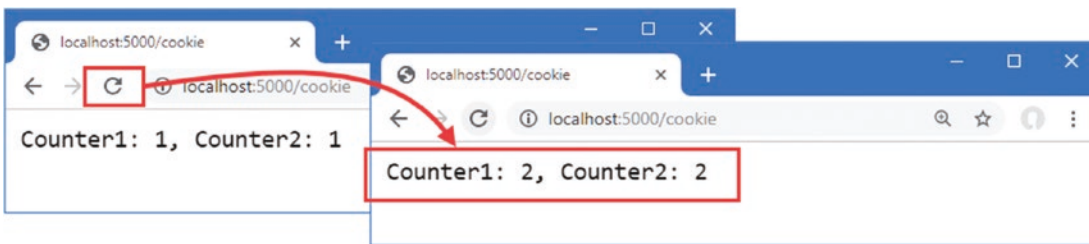


Figure 16-5. Using session data

Working with HTTPS Connections

Users increasingly expect web applications to use HTTPS connections, even for requests that don't contain or return sensitive data. ASP.NET Core supports both HTTP and HTTPS connections and provides middleware that can force HTTP clients to use HTTPS.

HTTPS VS SSL VS TLS

HTTPS is the combination of HTTP and the Transport Layer Security (TLS) or Secure Sockets Layer (SSL). TLS has replaced the obsolete SSL protocol, but the term SSL has become synonymous with secure networking and is often used when TLS is actually being used. If you are interested in security and cryptography, then the details of HTTPS are worth exploring, and <https://en.wikipedia.org/wiki/HTTPS> is a good place to start.

Enabling HTTP Connections

If you are using Visual Studio, select `Project` ► `Platform Properties`, navigate to the `Debug` section, and check the `Enable SSL` option, as shown in Figure 16-6. Once you have checked the option, select `File` ► `Save All` to save the configuration change.

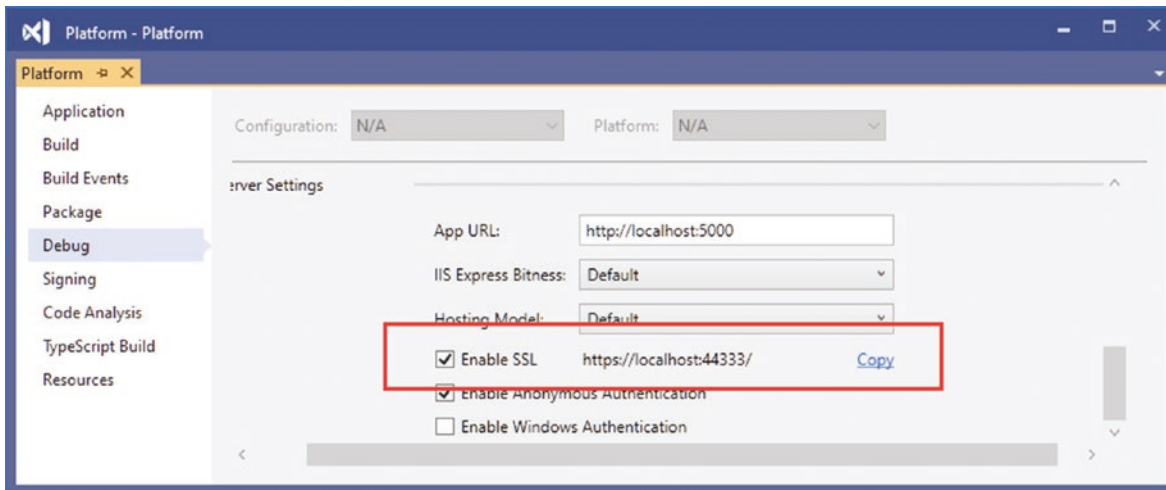


Figure 16-6. Enabling HTTPS in Visual Studio

The port used to receive HTTPS is assigned randomly. To change the port, click the Show All button at the top of the Solution Explorer to reveal the Properties folder and the `launchSettings.json` file if they are not already visible. Edit the file to change the HTTPS port that is used by the application, as shown in Listing 16-9.

Listing 16-9. Changing the HTTPS Port in the `launchSettings.json` File in the Platform/Properties Folder

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:5000",
      "sslPort": 44350
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "Platform": {
      "commandName": "Project",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      },
      "applicationUrl": "https://localhost:44350;http://localhost:5000"
    }
  }
}
```

The new value for the `sslPort` setting changes the port used when the application is started within Visual Studio. The new `applicationUrl` setting sets the port used when the application is started from the command line or with Visual Studio.

■ **Note** IIS Express supports HTTPS only on port numbers between 4400 and 44399.

The .NET Core runtime includes a test certificate that is used for HTTPS requests. Run the commands shown in Listing 16-10 in the Platform folder to regenerate and trust the test certificate.

Listing 16-10. Regenerating the Development Certificates

```
dotnet dev-certs https --clean
dotnet dev-certs https --trust
```

Select Yes to the prompts to delete the existing certificate it has already been trusted and select Yes to trust the new certificate, as shown in Figure 16-7.

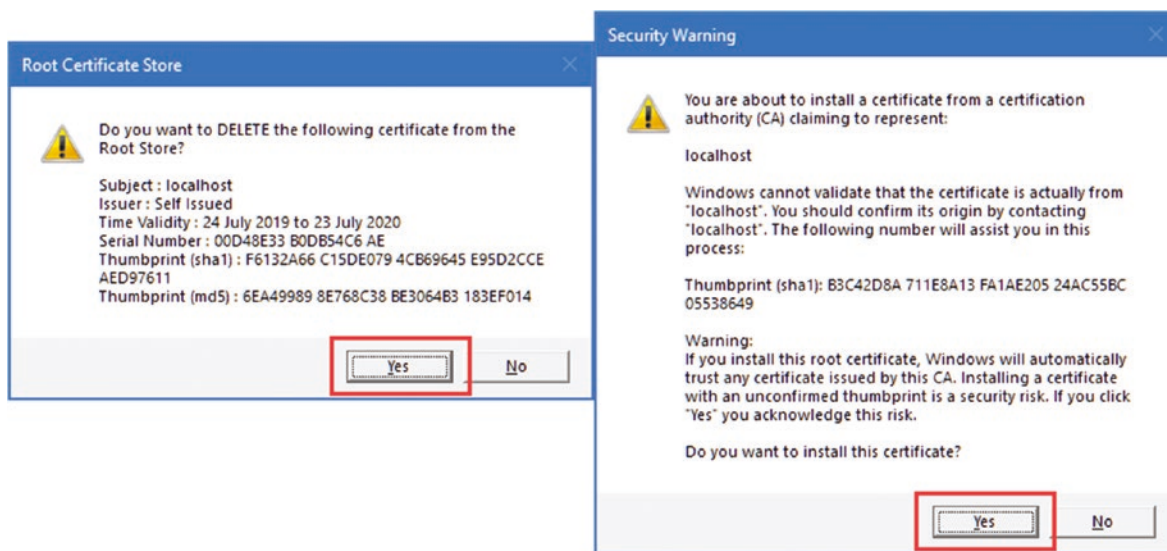


Figure 16-7. Regenerating the HTTPS certificate

Detecting HTTPS Requests

Requests made using HTTPS can be detected through the `HttpRequest.IsHttps` property. In Listing 16-11, I added a new middleware component to the request pipeline that adds a message to the response when a request is made using HTTPS.

Listing 16-11. Detecting HTTPS in the Startup.cs File in the Platform Folder

```
...
public void Configure(IApplicationBuilder app) {
    app.UseDeveloperExceptionPage();
    app.UseCookiePolicy();
    app.UseMiddleware<ConsentMiddleware>();
    app.UseSession();
    app.UseRouting();

    app.Use(async (context, next) => {
        await context.Response
            .WriteAsync($"HTTPS Request: {context.Request.IsHttps} \n");
    });
}
```

```

    await next();
});

app.UseEndpoints(endpoints => {

    endpoints.MapGet("/cookie", async context => {
        int counter1 = (context.Session.GetInt32("counter1") ?? 0) + 1;
        int counter2 = (context.Session.GetInt32("counter2") ?? 0) + 1;
        context.Session.SetInt32("counter1", counter1);
        context.Session.SetInt32("counter2", counter2);
        await context.Session.CommitAsync();
        await context.Response
            .WriteAsync($"Counter1: {counter1}, Counter2: {counter2}");
    });

    endpoints.MapFallback(async context =>
        await context.Response.WriteAsync("Hello World!"));
});
}
...

```

To test HTTPS, restart ASP.NET Core and navigate to `http://localhost:5000`. This is a regular HTTP request and will produce the result shown on the left of Figure 16-8. Next, navigate to `https://localhost:44350`, paying close attention to the URL scheme, which is `https` and not `http`, as it has been in previous examples. The new middleware will detect the HTTPS connection and produce the output on the right of Figure 16-8.

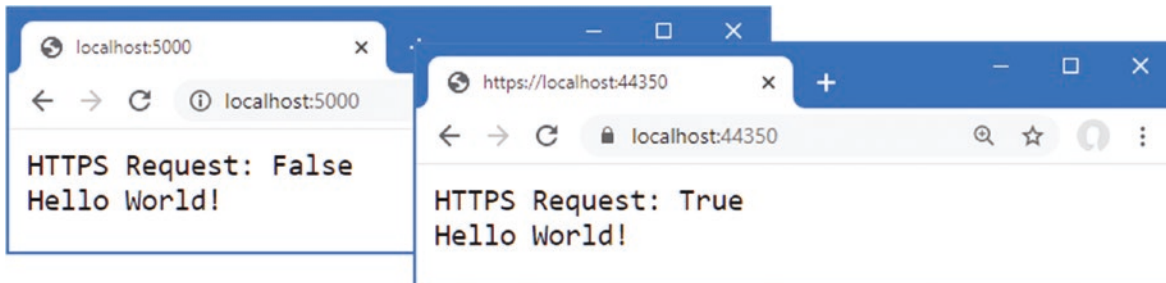


Figure 16-8. Making an HTTPS request

Enforcing HTTPS Requests

ASP.NET Core provides a middleware component that enforces the use of HTTPS by sending a redirection response for requests that arrive over HTTP. Listing 16-12 adds this middleware to the request pipeline.

Listing 16-12. Enforcing HTTPS in the Startup.cs File in the Platform Folder

```

...
public void Configure(IApplicationBuilder app) {
    app.UseDeveloperExceptionPage();
    app.UseHttpsRedirection();
    app.UseCookiePolicy();
    app.UseMiddleware<ConsentMiddleware>();
    app.UseSession();
    app.UseRouting();
}

```



```

app.Use(async (context, next) => {
    await context.Response
        .WriteAsync($"HTTPS Request: {context.Request.IsHttps} \n");
    await next();
});

app.UseEndpoints(endpoints => {

    endpoints.MapGet("/cookie", async context => {
        int counter1 = (context.Session.GetInt32("counter1") ?? 0) + 1;
        int counter2 = (context.Session.GetInt32("counter2") ?? 0) + 1;
        context.Session.SetInt32("counter1", counter1);
        context.Session.SetInt32("counter2", counter2);
        await context.Session.CommitAsync();
        await context.Response
            .WriteAsync($"Counter1: {counter1}, Counter2: {counter2}");
    });

    endpoints.MapFallback(async context =>
        await context.Response.WriteAsync("Hello World!"));
});
}
...

```

The `UseHttpsRedirection` method adds the middleware component, which appears at the start of the request pipeline so that the redirection to HTTPS occurs before any other component can short-circuit the pipeline and produce a response using regular HTTP.

CONFIGURING HTTPS REDIRECTION

The options pattern can be used to configure the HTTPS redirection middleware, by calling the `AddHttpsRedirection` method in the `ConfigureServices` method, like this:

```

...
services.AddHttpsRedirection(opts => {
    opts.RedirectStatusCode = StatusCodes.Status307TemporaryRedirect;
    opts.HttpsPort = 443;
});
...

```

The only two configuration options are shown in this fragment, which sets the status code used in the redirection responses and the port to which the client is redirected, overriding the value that is loaded from the configuration files. Specifying the HTTPS port can be useful when deploying the application, but care should be taken when changing the redirection status code.

Restart ASP.NET Core and request `http://localhost:5000`, which is the HTTP URL for the application. The HTTPS redirection middleware will intercept the request and redirect the browser to the HTTPS URL, as shown in Figure 16-9.

■ **Tip** Modern browsers often hide the URL scheme, which is why you should pay attention to the port number that is displayed. To display the URL scheme in the figure, I had to click the URL bar so the browser would display the full URL.

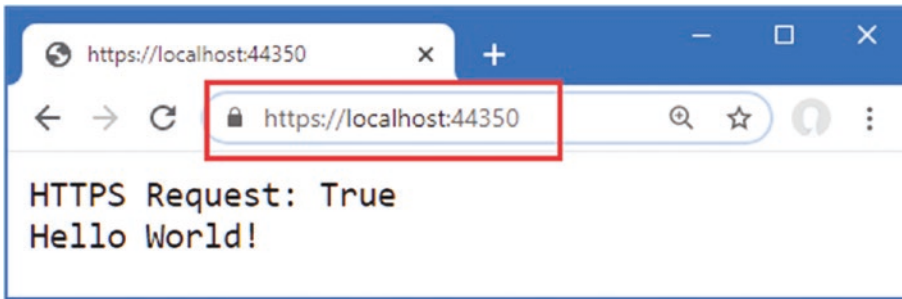


Figure 16-9. Forcing HTTPS requests

Enabling HTTP Strict Transport Security

One limitation of HTTPS redirection is that the user can make an initial request using HTTP before being redirected to a secure connection, presenting a security risk.

The HTTP Strict Transport Security (HSTS) protocol is intended to help mitigate this risk and works by including a header in responses that tells browsers to use HTTPS only when sending requests to the web application's host. After an HSTS header has been received, browsers that support HSTS will send requests to the application using HTTPS even if the user specifies an HTTP URL. Listing 16-13 shows the addition of the HSTS middleware to the request pipeline.

Listing 16-13. Enabling HSTS in the Startup.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using System;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Hosting;

namespace Platform {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.Configure<CookiePolicyOptions>(opts => {
                opts.CheckConsentNeeded = context => true;
            });

            services.AddDistributedMemoryCache();

            services.AddSession(options => {
                options.IdleTimeout = TimeSpan.FromMinutes(30);
                options.Cookie.IsEssential = true;
            });

            services.AddHsts(opts => {
                opts.MaxAge = TimeSpan.FromDays(1);
                opts.IncludeSubDomains = true;
            });
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
            app.UseDeveloperExceptionPage();
            if (env.IsProduction()) {
                app.UseHsts();
            }
        }
    }
}
```

```

app.UseHttpsRedirection();
app.UseCookiePolicy();
app.UseMiddleware<ConsentMiddleware>();
app.UseSession();
app.UseRouting();

app.Use(async (context, next) => {
    await context.Response
        .WriteAsync($"HTTPS Request: {context.Request.IsHttps} \n");
    await next();
});

app.UseEndpoints(endpoints => {

    endpoints.MapGet("/cookie", async context => {
        int counter1 = (context.Session.GetInt32("counter1") ?? 0) + 1;
        int counter2 = (context.Session.GetInt32("counter2") ?? 0) + 1;
        context.Session.SetInt32("counter1", counter1);
        context.Session.SetInt32("counter2", counter2);
        await context.Session.CommitAsync();
        await context.Response
            .WriteAsync($"Counter1: {counter1}, Counter2: {counter2}");
    });

    endpoints.MapFallback(async context =>
        await context.Response.WriteAsync("Hello World!"));
});
}
}
}

```

The middleware is added to the request pipeline using the `UseHsts` method. The HSTS middleware can be configured with the `AddHsts` method, using the properties described in Table 16-9.

Table 16-9. The HSTS Configuration Properties

Name	Description
<code>ExcludeHosts</code>	This property returns a <code>List<string></code> that contains the hosts for which the middleware won't send an HSTS header. The defaults exclude <code>localhost</code> and the loopback addresses for IP version 4 and version 6.
<code>IncludeSubDomains</code>	When true, the browser will apply the HSTS setting to subdomains. The default value is false.
<code>MaxAge</code>	This property specifies the time period for which the browser should make only HTTPS requests. The default value is 30 days.
<code>Preload</code>	This property is set to true for domains that are part of the HSTS preload scheme. The domains are hard-coded into the browser, which avoids the initial insecure request and ensures that only HTTPS is used. See httpspreload.org for more details.

HSTS is disabled during development and enabled only in production, which is why the `UseHsts` method is called only for that environment.

```

...
if (env.IsProduction()) {
    app.UseHsts();
}
...

```

HSTS must be used with care because it is easy to create a situation where clients cannot access the application, especially when nonstandard ports are used for HTTP and HTTPS.

If the example application is deployed to a server named `myhost`, for example, and the user requests `http://myhost:5000`, the browser will be sent the HSTS header and redirected to `https://myhost:5001`, and the application will work as expected. But the next time the user requests `http://myhost:5000`, they will receive an error stating that a secure connection cannot be established.

This problem arises because browsers take a simplistic approach to HSTS and assume that HTTP requests are handled on port 80 and HTTPS requests on port 443.

When the user requests `http://myhost:5000`, the browser checks its HSTS data and sees that it previously received an HSTS header for `myhost`. Instead of the HTTP URL that the user entered, the browser sends a request to `https://myhost:5000`. ASP.NET Core doesn't handle HTTPS on the port it uses for HTTP, and the request fails. The browser doesn't remember or understand the redirection it previously received for port 5001.

This isn't an issue where port 80 is used for HTTP and 443 is used for HTTPS. The URL `http://myhost` is equivalent to `http://myhost:80`, and `https://myhost` is equivalent to `https://myhost:443`, which means that changing the scheme targets the right port.

Once a browser has received an HSTS header, it will continue to honor it for the duration of the header's `MaxAge` property. When you first deploy an application, it is a good idea to set the HSTS `MaxAge` property to a relatively short duration until you are confident that your HTTPS infrastructure is working correctly, which is why I have set `MaxAge` to one day in Listing 16-13. Once you are sure that clients will not need to make HTTP requests, you can increase the `MaxAge` property. A `MaxAge` value of one year is commonly used.

■ **Tip** If you are testing HSTS with Google Chrome, you can inspect and edit the list of domains to which HSTS is applied by navigating to `chrome://net-internals/#hsts`.

Handling Exceptions and Errors

If an exception occurs when processing a request, the ASP.NET Core will return a response using the status code 500, which tells the browser that something has gone wrong.

A plain 500 result isn't useful during development because it reveals nothing about the cause of the problem. The `UseDeveloperExceptionPage` method adds a middleware component that intercepts exceptions and presents a more useful response.

To demonstrate the way that exceptions are handled, Listing 16-14 replaces the middleware and endpoints used in earlier examples with a new component that deliberately throws an exception.

Listing 16-14. Adding a Middleware Component in the `Startup.cs` File in the Platform Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using System;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Hosting;

namespace Platform {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.Configure<CookiePolicyOptions>(opts => {
                opts.CheckConsentNeeded = context => true;
            });

            services.AddDistributedMemoryCache();

            services.AddSession(options => {
                options.IdleTimeout = TimeSpan.FromMinutes(30);
                options.Cookie.IsEssential = true;
            });
        }
    }
}
```

```

        services.AddHsts(opts => {
            opts.MaxAge = TimeSpan.FromDays(1);
            opts.IncludeSubDomains = true;
        });
    }

    public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
        app.UseDeveloperExceptionPage();
        if (env.IsProduction()) {
            app.UseHsts();
        }
        app.UseHttpsRedirection();
        app.UseCookiePolicy();
        app.UseMiddleware<ConsentMiddleware>();
        app.UseSession();

        app.Run(context => {
            throw new Exception("Something has gone wrong");
        });
    }
}
}
}

```

Restart ASP.NET Core and navigate to `https://localhost:44350` to see the response that the middleware component generates, which is shown in Figure 16-10. The page presents a stack trace and details about the request, including details of the headers and cookies it contained.

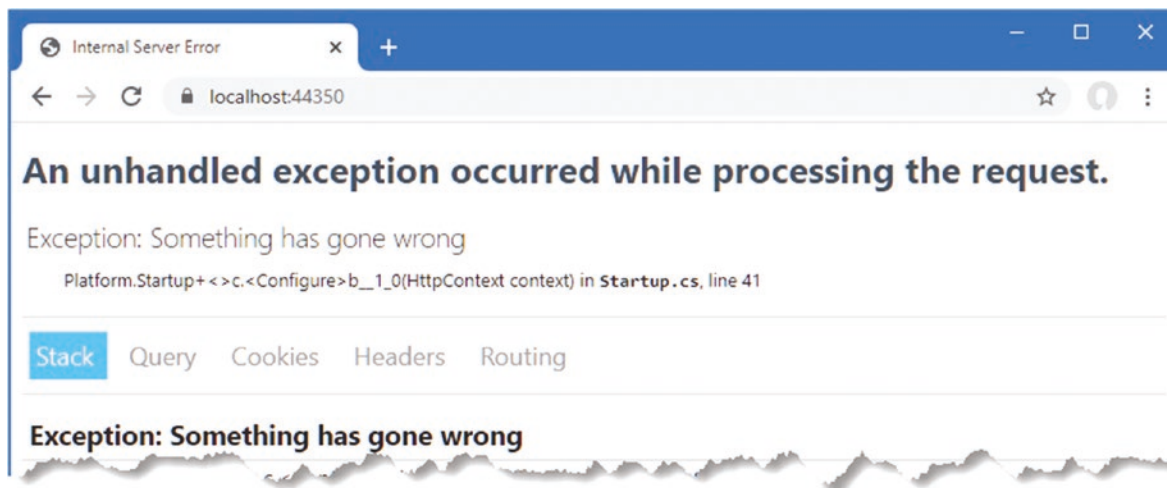


Figure 16-10. The developer exception page

MANAGING DEVELOPMENT AND PRODUCTION MIDDLEWARE

The details shown in Figure 16-10 should be revealed only to developers, which is why the middleware is added to the pipeline only during development. A common pattern is to use the `IWebHostEnvironment` service to check the environment, calling the `UseDeveloperExceptionPage` method in `Development` and the `UseHsts` method in the other environments, like this:

```
...
public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
    if (env.IsDevelopment()) {
        app.UseDeveloperExceptionPage();
    } else {
        app.UseHsts();
    }
    app.UseHttpsRedirection();
    app.UseCookiePolicy();
    app.UseMiddleware<ConsentMiddleware>();
    app.UseSession();

    app.Run(context => {
        throw new Exception("Something has gone wrong");
    });
}
...
```

This combination ensures that the exception handling middleware is added only in development and the HSTS middleware is added only outside of development.

Returning an HTML Error Response

When the developer exception middleware is disabled, as it should be when the application is in production, ASP.NET Core deals with unhandled exceptions by sending a response that contains just an error code. Listing 16-15 disables the development exception middleware so that the default behavior can be seen.

Listing 16-15. Disabling Middleware in the Startup.cs File in the Platform Folder

```
...
public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
    //app.UseDeveloperExceptionPage();
    if (env.IsProduction()) {
        app.UseHsts();
    }
    app.UseHttpsRedirection();
    app.UseCookiePolicy();
    app.UseMiddleware<ConsentMiddleware>();
    app.UseSession();

    app.Run(context => {
        throw new Exception("Something has gone wrong");
    });
}
...
```

Restart ASP.NET Core and navigate to `https://localhost:44350`. The response you see will depend on your browser because ASP.NET Core has only provided it with a response containing status code 500, without any content to display. Figure 16-11 shows how this is handled by Google Chrome.

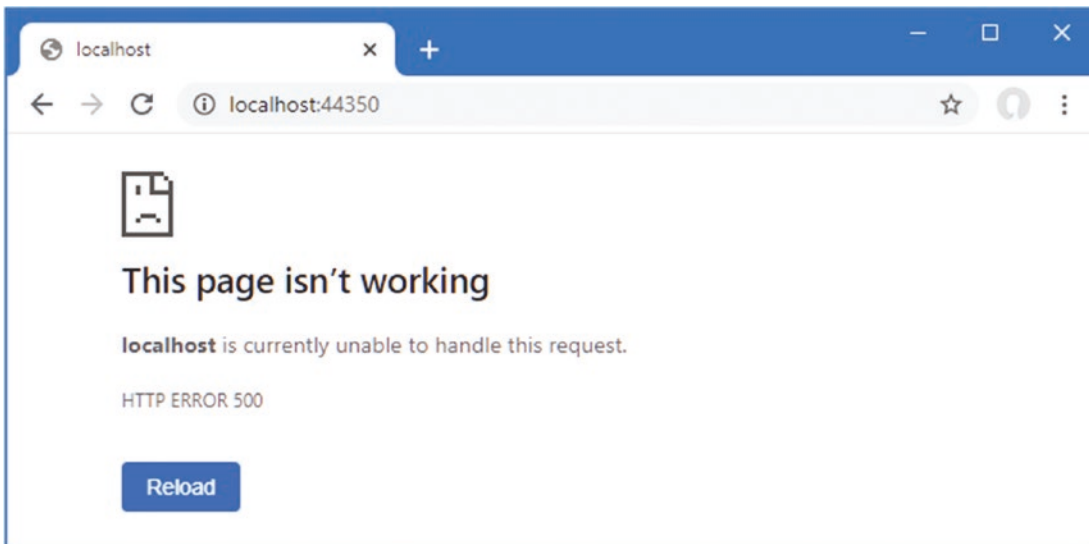


Figure 16-11. Returning an error response

As an alternative to returning just status codes, ASP.NET Core provides middleware that intercepts unhandled exceptions and sends a redirection to the browser instead, which can be used to show a friendlier response than the raw status code. The exception redirection middleware is added with the `UseExceptionHandler` method, as shown in Listing 16-16.

Listing 16-16. Returning an HTML Error Response in the Startup.cs File in the Platform Folder

```
...
public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
    //app.UseDeveloperExceptionPage();
    app.UseExceptionHandler("/error.html");
    if (env.IsProduction()) {
        app.UseHsts();
    }
    app.UseHttpsRedirection();
    app.UseCookiePolicy();
    app.UseStaticFiles();
    app.UseMiddleware<ConsentMiddleware>();
    app.UseSession();

    app.Run(context => {
        throw new Exception("Something has gone wrong");
    });
}
...
```

When an exception is thrown, the exception handler middleware will intercept the response and redirect the browser to the URL provided as the argument to the `UseExceptionHandler` method. For this example, the redirection is to a URL that will be handled by a static file, so the `UseStaticFiles` middleware has also been added to the pipeline.

To add the file that the browser will receive, create an HTML file named `error.html` in the `wwwroot` folder and add the content shown in Listing 16-17.

Listing 16-17. The Contents of the `error.html` File in the Platform/wwwroot Folder

```
<!DOCTYPE html>
<html lang="en">
```

```

<head>
  <link rel="stylesheet" href="/lib/twitter-bootstrap/css/bootstrap.min.css" />
  <title>Error</title>
</head>
<body class="text-center">
  <h3 class="p-2">Something went wrong...</h3>
  <h6>You can go back to the <a href="/">homepage</a> and try again</h6>
</body>
</html>

```

Restart ASP.NET Core and navigate to <https://localhost:44350> to see the effect of the new middleware. Instead of the raw status code, the browser will be sent a redirection to the `/error.html` URL, as shown in Figure 16-12.

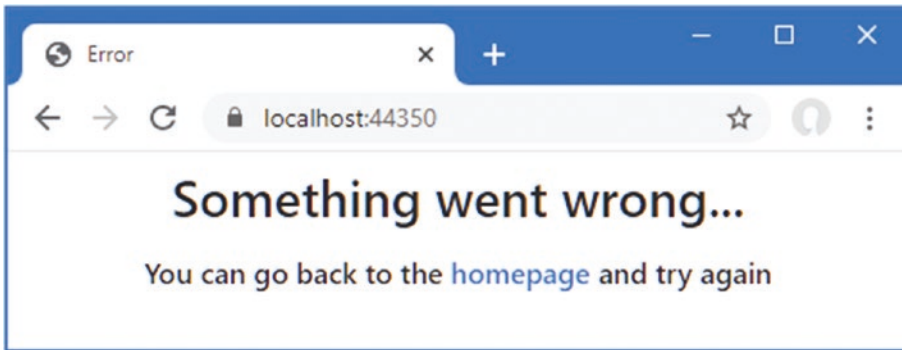


Figure 16-12. Displaying an HTML error

There are versions of the `UseExceptionHandler` method that allow more complex responses to be composed, but my advice is to keep error handling as simple as possible because you can't anticipate all of the problems an application may encounter, and you run the risk of encountering another exception when trying to handle the one that triggered the handler, resulting in a confusing response or no response at all.

Enriching Status Code Responses

Not all error responses will be the result of uncaught exceptions. Some requests cannot be processed for reasons other than software defects, such as requests for URLs that are not supported or that require authentication. For this type of problem, redirecting the client to a different URL can be problematic because some clients rely on the error code to detect problems. You will see examples of this in later chapters when I show you how to create and consume RESTful web applications.

ASP.NET Core provides middleware that adds user-friendly content to error responses without requiring redirection. This preserves the error status code while providing a human-readable message that helps users make sense of the problem.

The simplest approach is to define a string that will be used as the body for the response. This is more awkward than simply pointing at a file, but it is a more reliable technique and, as a rule, simple and reliable techniques are preferable when handling errors. To create the string response for the example project, add a class file named `ResponseStrings.cs` to the `Platform` folder with the code shown in Listing 16-18.

Listing 16-18. The Contents of the `ResponseStrings.cs` File in the `Platform` Folder

```

namespace Platform {

public static class Responses {

    public static string DefaultResponse = @"
        <!DOCTYPE html>
        <html lang=""en"">

```



```

        <head>
            <link rel=""stylesheet""
                href=""/lib/twitter-bootstrap/css/bootstrap.min.css"" />
            <title>Error</title>
        </head>
        <body class=""text-center"">
            <h3 class=""p-2"">Error {0}</h3>
            <h6>
                You can go back to the <a href=""/"">homepage</a> and try again
            </h6>
        </body>
    </html>";
}
}
}

```

The Responses class defines a DefaultResponse property to which I have assigned a multiline string containing a simple HTML document. There is a placeholder—{0}—into which the response status code will be inserted when the response is sent to the client.

Listing 16-19 adds the status code middleware to the request pipeline and adds a new middleware component that will return a 404 status code, indicating that the requested URL was not found.

Listing 16-19. Adding Status Code Middleware in the Startup.cs File in the Platform Folder

```

...
public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
    //app.UseDeveloperExceptionPage();
    app.UseExceptionHandler("/error.html");
    if (env.IsProduction()) {
        app.UseHsts();
    }
    app.UseStaticFiles();
    app.UseHttpsRedirection();
    app.UseStatusCodePages("text/html", Responses.DefaultResponse);
    app.UseCookiePolicy();
    app.UseMiddleware<ConsentMiddleware>();
    app.UseSession();

    app.Use(async (context, next) => {
        if (context.Request.Path == "/error") {
            context.Response.StatusCode = StatusCodes.Status404NotFound;
            await Task.CompletedTask;
        } else {
            await next();
        }
    });

    app.Run(context => {
        throw new Exception("Something has gone wrong");
    });
}
...

```

The UseStatusCodePages method adds the response-enriching middleware to the request pipeline. The first argument is the value that will be used for the response's Content-Type header, which is text/html in this example. The second argument is the string that will be used as the body of the response, which is the HTML string from Listing 16-18.

The custom middleware component sets the HttpResponseMessage.StatusCode property to specify the status code for the response, using a value defined by the StatusCode class. Middleware components are required to return a Task, so I have used the Task.CompletedTask property because there is no work for this middleware component to do.

To see how the 404 status code is handled, restart ASP.NET Core and request `https://localhost:44350/error`. The status code middleware will intercept the result and add the content shown in Figure 16-13 to the response. The string used as the second argument to `UseStatusCodePages` is interpolated using the status code to resolve the placeholder.

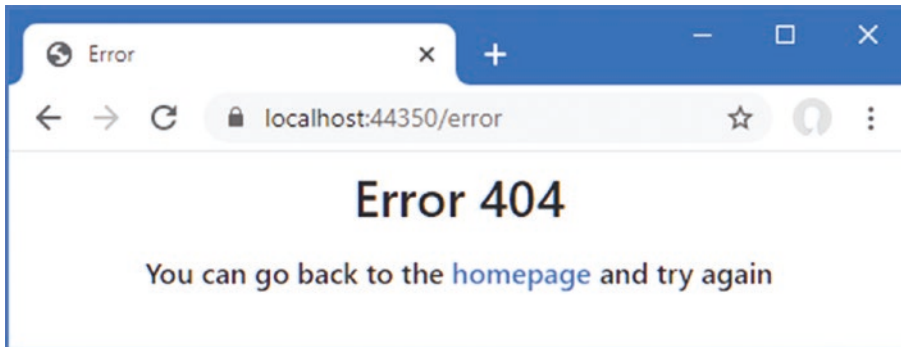


Figure 16-13. Using the status code middleware

The status code middleware responds only to status codes between 400 and 600 and doesn't alter responses that already contain content, which means you won't see the response in the figure if an error occurs after another middleware component has started to generate a response. The status code middleware won't respond to unhandled exceptions because exceptions disrupt the flow of a request through the pipeline, meaning that the status code middleware isn't given the opportunity to inspect the response before it is sent to the client. As a result, the `UseStatusCodePages` method is typically used in conjunction with the `UseExceptionHandler` or `UseDeveloperExceptionPage` method.

■ **Note** There are two related methods, `UseStatusCodePagesWithRedirects` and `UseStatusCodePagesWithReExecute`, which work by redirecting the client to a different URL or by rerunning the request through the pipeline with a different URL. In both cases, the original status code may be lost.

Filtering Requests Using the Host Header

The HTTP specification requires requests to include a `Host` header that specifies the hostname the request is intended for, which makes it possible to support virtual servers where one HTTP server receives requests on a single port and handles them differently based on the hostname that was requested.

The default set of middleware that is added to the request pipeline by the `Program` class includes middleware that filters requests based on the `Host` header so that only requests that target a list of approved hostnames are handled and all other requests are rejected.

The default configuration for the `Hosts` header middleware is included in the `appsettings.json` file, as follows:

```
...
{
  "Location": {
    "CityName": "Buffalo"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information"
    }
  }
},
```

```

"AllowedHosts": "*"
}
...

```

The AllowedHosts configuration property is added to the JSON file when the project is created, and the default value accepts requests regardless of the Host header value. You can change the configuration by editing the JSON file. The configuration can also be changed using the options pattern, as shown in Listing 16-20.

■ **Note** The middleware is already added to the pipeline by the Program class, but you can use the UseHostFiltering method if you need to add the middleware explicitly.

Listing 16-20. Configuring Host Header Filtering in the Startup.cs File in the Platform Folder

```

using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using System;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.Hosting;
using Microsoft.AspNetCore.HostFiltering;

namespace Platform {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
            services.Configure<CookiePolicyOptions>(opts => {
                opts.CheckConsentNeeded = context => true;
            });

            services.AddDistributedMemoryCache();

            services.AddSession(options => {
                options.IdleTimeout = TimeSpan.FromMinutes(30);
                options.Cookie.IsEssential = true;
            });

            services.AddHsts(opts => {
                opts.MaxAge = TimeSpan.FromDays(1);
                opts.IncludeSubDomains = true;
            });

            services.Configure<HostFilteringOptions>(opts => {
                opts.AllowedHosts.Clear();
                opts.AllowedHosts.Add("*.example.com");
            });
        }

        public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {

            // ...statements omitted for brevity...
        }
    }
}

```

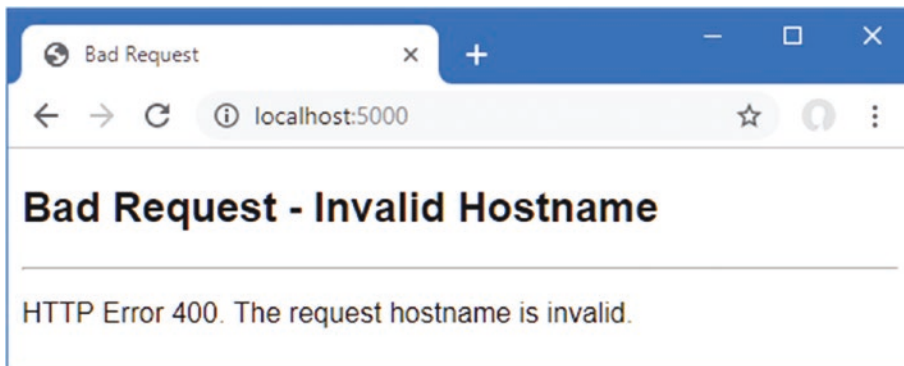
The HostFilteringOptions class is used to configure the host filtering middleware using the properties described in Table 16-10.

Table 16-10. *The HostFilteringOptions Properties*

Name	Description
AllowedHosts	This property returns a <code>List<string></code> that contains the domains for which requests are allowed. Wildcards are allowed so that <code>*.example.com</code> accepts all names in the <code>example.com</code> domain and <code>*</code> accepts all header values.
AllowEmptyHosts	When false, this property tells the middleware to reject requests that do not contain a Host header. The default value is true.
IncludeFailureMessage	When true, this property includes a message in the response that indicates the reason for the error. The default value is true.

In Listing 16-20, I called the `Clear` method to remove the wildcard entry that has been loaded from the `appsettings.json` file and then called the `Add` method to accept all hosts in the `example.com` domain. Requests sent from the browser to `localhost` will no longer contain an acceptable Host header. You can see what happens by restarting ASP.NET Core and using the browser to request `http://localhost:5000`. The Host header middleware checks the Host header in the request determines that the request hostname doesn't match the `AllowedHosts` list and terminates the request with the 400 status code, which indicates a bad request. Figure 16-14 shows the error message.

■ **Tip** Notice that the browser isn't redirected to use HTTPS because the Host header middleware is added to the request pipeline before the HTTPS redirection middleware.

**Figure 16-14.** *A request rejected based on the Host header*

Summary

In this chapter, I continued to describe the basic features provided by the ASP.NET Core platform. I showed you how to manage cookies and track cookie consent, how to use sessions, how to use and enforce HTTPS, how to deal with unhandled exceptions and status code responses, and, finally, how to filter requests based on the Host header. In the next chapter, I explain how ASP.NET Core supports working with data.



Working with Data

All the examples in the earlier chapters in this part of the book have generated fresh responses for each request, which is easy to do when dealing with simple strings or small fragments of HTML. Most real projects deal with data that is expensive to produce and needs to be used as efficiently as possible. In this chapter, I describe the features that ASP.NET Core provides for caching data and caching entire responses. I also show you how to create and configure the services required to access data in a database using Entity Framework Core. Table 17-1 puts the ASP.NET Core features for working with data in context.

■ **Note** The examples in this chapter rely on the SQL Server LocalDB feature that was installed in Chapter 2. You will encounter errors if you have not installed LocalDB and the required updates.

Table 17-1. Putting the ASP.NET Core Data Features in Context

Question	Answer
What are they?	The features described in this chapter allow responses to be produced using data that has been previously created, either because it was created for an earlier request or because it has been stored in a database.
Why are they useful?	Most web applications deal with data that is expensive to re-create for every request. The features in this chapter allow responses to be produced more efficiently and with fewer resources.
How are they used?	Data values are cached using a service. Responses are cached by a middleware component based on the Cache-Control header. Databases are accessed through a service that translates LINQ queries into SQL statements.
Are there any pitfalls or limitations?	For caching, it is important to test the effect of your cache policy before deploying the application to ensure you have found the right balance between efficiency and responsiveness. For Entity Framework Core, it is important to pay attention to the queries sent to the database to ensure that they are not retrieving large amounts of data that is processed and then discarded by the application.
Are there any alternatives?	All the features described in this chapter are optional. You can elect not to cache data or responses or to use an external cache. You can choose not to use a database or to access a database using a framework other than Entity Framework Core.

Table 17-2 summarizes the chapter.

Table 17-2. Chapter Summary

Problem	Solution	Listing
Caching data values	Set up a cache service and use it in endpoints and middleware components to store data values	7, 8
Creating a persistent cache	Use the database-backed cache	9-14
Caching entire responses	Enable the caching middleware and set the Cache-Control header in responses	15, 16
Storing application data	Use Entity Framework Core	17-23, 26-28
Creating a database schema	Create and apply migrations	24, 25
Accessing data in endpoints	Consume the database context service	29
Including all request details in logging messages	Enable the sensitive data logging feature	30

Preparing for This Chapter

In this chapter, I continue to use the Platform project from Chapter 16. To prepare for this chapter, replace the contents of the Startup.cs file with the code shown in Listing 17-1.

■ **Tip** You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/apress/pro-asp.net-core-3>. See Chapter 1 for how to get help if you have problems running the examples.

Listing 17-1. Replacing the Contents of the Startup.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace Platform {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
        }

        public void Configure(IApplicationBuilder app) {
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseRouting();
            app.UseEndpoints(endpoints => {
                endpoints.MapGet("/", async context => {
                    await context.Response.WriteAsync("Hello World!");
                });
            });
        }
    }
}
```

To reduce the detail level of the logging messages displayed by the application, make the changes shown in Listing 17-2 to the appsettings.Development.json file.

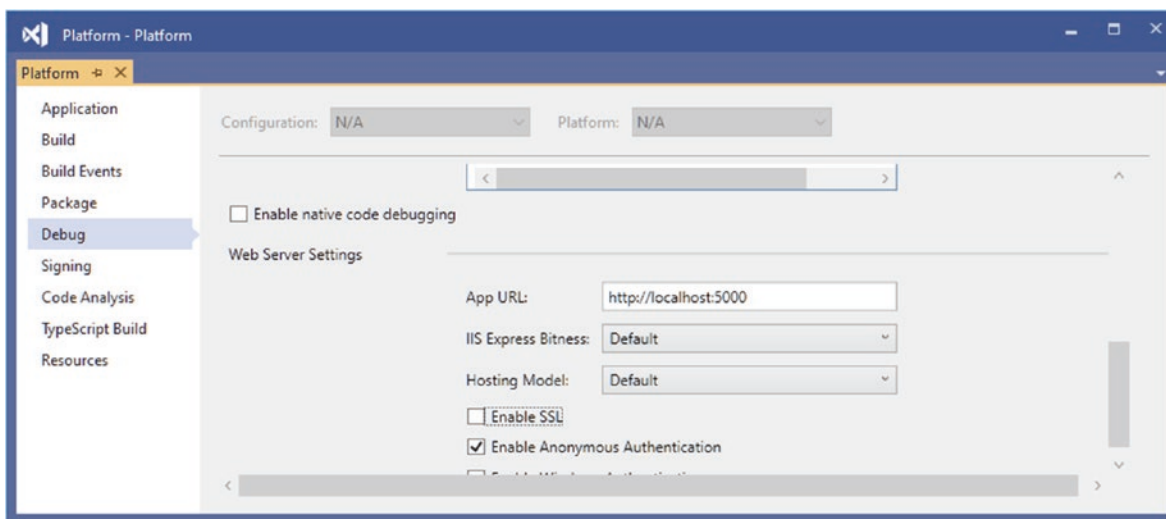
Listing 17-2. Changing the Log Detail in the appsettings.Development.json File in the Platform Folder

```

{
  "Logging": {
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information",
      "Microsoft.AspNetCore.Hosting.Diagnostics": "None"
    }
  }
}

```

If you are using Visual Studio, select Project ► Platform Properties, navigate to the Debug tab, uncheck the Enable SSL option (as shown in Figure 17-1), and select File ► Save All.

**Figure 17-1.** Disabling HTTPS

If you are using Visual Studio Code, open the Properties/launchSettings.json file to remove the HTTPS URL, as shown in Listing 17-3.

Listing 17-3. Disabling SSL in the launchSettings.json File in the Platform/Properties Folder

```

{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:5000",
      "sslPort": 0
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    }
  }
}

```

```

    "Platform": {
      "commandName": "Project",
      "launchBrowser": true,
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      },
      "applicationUrl": "http://localhost:5000"
    }
  }
}

```

Start the application by selecting Start Without Debugging or Run Without Debugging from the Debug menu or by opening a new PowerShell command prompt, navigating to the Platform project folder (which contains the Platform.csproj file), and running the command shown in Listing 17-4.

Listing 17-4. Starting the ASP.NET Core Runtime

```
dotnet run
```

If the application was started using Visual Studio or Visual Studio Code, a new browser window will open and display the content shown in Figure 17-2. If the application was started from the command line, open a new browser tab and navigate to `https://localhost:5000`; you will see the content shown in Figure 17-2.

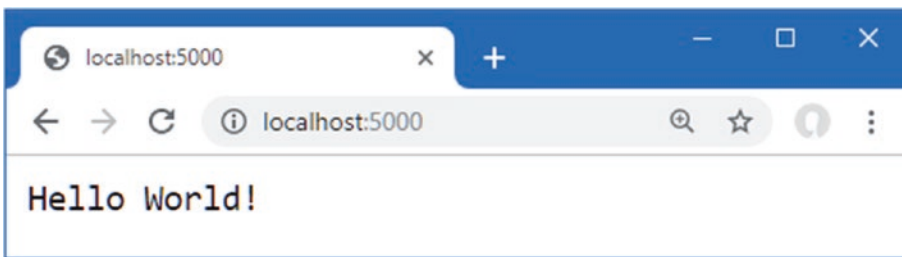


Figure 17-2. Running the example application

Caching Data

In most web applications, there will be some items of data that are relatively expensive to generate but are required repeatedly. The exact nature of the data is specific to each project, but repeatedly performing the same set of calculations can increase the resources required to host the application and drive up hosting costs. To represent an expensive response, add a class file called `SumEndpoint.cs` to the Platform folder with the code shown in Listing 17-5.

Listing 17-5. The Contents of the SumEndpoint.cs File in the Platform Folder

```

using Microsoft.AspNetCore.Http;
using System;
using System.Threading.Tasks;

namespace Platform {

    public class SumEndpoint {

        public async Task Endpoint(HttpContext context) {
            int count = int.Parse((string)context.Request.RouteValues["count"]);
            long total = 0;

```



```

        for (int i = 1; i <= count; i++) {
            total += i;
        }
        string totalString = $"({ DateTime.Now.ToLongTimeString() }) {total}";
        await context.Response.WriteAsync(
            $"({DateTime.Now.ToLongTimeString()}) Total for {count}"
            + $" values:\n{totalString}\n");
    }
}

```

Listing 17-6 creates a route that uses the endpoint, which is applied using the `MapEndpoint` extension methods created in Chapter 13.

Listing 17-6. Adding an Endpoint in the Startup.cs File in the Platform Folder

```

using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace Platform {
    public class Startup {

        public void ConfigureServices(IServiceCollection services) {
        }

        public void Configure(IApplicationBuilder app) {
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseRouting();
            app.UseEndpoints(endpoints => {

                endpoints.MapEndpoint<SumEndpoint>("/sum/{count:int=1000000000}");

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

```

Restart ASP.NET Core and use a browser to request `https://localhost:5000/sum`. The endpoint will sum 1,000,000,000 integer values and produce the result shown in Figure 17-3.

Reload the browser window, and the endpoint will repeat the calculation. Both the timestamps in the response change, as shown in the figure, indicating that every part of the response was produced fresh for each request.

■ **Tip** You may need to increase or decrease the default value for the route parameter based on the capabilities of your machine. Try to find a value that takes two or three seconds to produce the result—just long enough that you can tell when the calculation is being performed but not so long that you can step out for coffee while it happens.

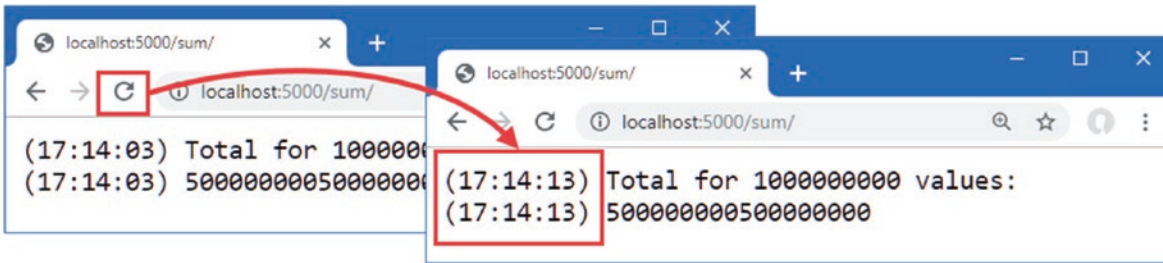


Figure 17-3. An expensive response

Caching Data Values

ASP.NET Core provides a service that can be used to cache data values through the `IDistributedCache` interface. Listing 17-7 revises the endpoint to declare a dependency on the service and use it to cache calculated values.

Listing 17-7. Using the Cache Service in the `SumEndpoint.cs` File in the Platform Folder

```
using Microsoft.AspNetCore.Http;
using System;
using System.Threading.Tasks;
using Microsoft.Extensions.Caching.Distributed;

namespace Platform {

    public class SumEndpoint {

        public async Task Endpoint(HttpContext context, IDistributedCache cache) {
            int count = int.Parse((string)context.Request.RouteValues["count"]);
            string cacheKey = $"sum_{count}";
            string totalString = await cache.GetStringAsync(cacheKey);
            if (totalString == null) {
                long total = 0;
                for (int i = 1; i <= count; i++) {
                    total += i;
                }
                totalString = $"({DateTime.Now.ToLongTimeString()}) {total}";
                await cache.SetStringAsync(cacheKey, totalString,
                    new DistributedCacheEntryOptions {
                        AbsoluteExpirationRelativeToNow = TimeSpan.FromMinutes(2)
                    });
            }
            await context.Response.WriteAsync(
                $"({DateTime.Now.ToLongTimeString()}) Total for {count}"
                + $" values:\n{totalString}\n");
        }
    }
}
```

The cache service can store only byte arrays, which can be restrictive but allows for a range of `IDistributedCache` implementations to be used. There are extension methods available that allow strings to be used, which is a more convenient way of caching most data. Table 17-3 describes the most useful methods for using the cache.

Table 17-3. Useful *IDistributedCache* Methods

Name	Description
<code>GetString(key)</code>	This method returns the cached string associated with the specified key, or <code>null</code> if there is no such item.
<code>GetStringAsync(key)</code>	This method returns a <code>Task<string></code> that produces the cached string associated with the key, or <code>null</code> if there is no such item.
<code>SetString(key, value, options)</code>	This method stores a string in the cache using the specified key. The cache entry can be configured with an optional <code>DistributedCacheEntryOptions</code> object.
<code>SetStringAsync(key, value, options)</code>	This method asynchronously stores a string in the cache using the specified key. The cache entry can be configured with an optional <code>DistributedCacheEntryOptions</code> object.
<code>Refresh(key)</code>	This method resets the expiry interval for the value associated with the key, preventing it from being flushed from the cache.
<code>RefreshAsync(key)</code>	This method asynchronously resets the expiry interval for the value associated with the key, preventing it from being flushed from the cache.
<code>Remove(key)</code>	This method removes the cached item associated with the key.
<code>RemoveAsync(key)</code>	This method asynchronously removes the cached item associated with the key.

By default, entries remain in the cache indefinitely, but the `SetString` and `SetStringAsync` methods accept an optional `DistributedCacheEntryOptions` argument that is used to set an expiry policy, which tells the cache when to eject the item. Table 17-4 shows the properties defined by the `DistributedCacheEntryOptions` class.

Table 17-4. The *DistributedCacheEntryOptions* Properties

Name	Description
<code>AbsoluteExpiration</code>	This property is used to specify an absolute expiry date.
<code>AbsoluteExpirationRelativeToNow</code>	This property is used to specify a relative expiry date.
<code>SlidingExpiration</code>	This property is used to specify a period of inactivity, after which the item will be ejected from the cache if it hasn't been read.

In Listing 17-7, the endpoint uses the `GetStringAsync` to see whether there is a cached result available from a previous request. If there is no cached value, the endpoint performs the calculation and caches the result using `SetStringAsync` method, with the `AbsoluteExpirationRelativeToNow` property to tell the cache to eject the item after two minutes.

```
...
await cache.SetStringAsync(cacheKey, totalStr,
    new DistributedCacheEntryOptions {
        AbsoluteExpirationRelativeToNow = TimeSpan.FromMinutes(2)
    });
...
```

The next step is to set up the cache service in the `Startup` class, as shown in Listing 17-8.

Listing 17-8. Adding a Service in the `Startup.cs` File in the Platform Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;

namespace Platform {
    public class Startup {
```

```

public void ConfigureServices(IServiceCollection services) {
    services.AddDistributedMemoryCache(opts => {
        opts.SizeLimit = 200;
    });
}

public void Configure(IApplicationBuilder app) {
    app.UseDeveloperExceptionPage();
    app.UseStaticFiles();
    app.UseRouting();
    app.UseEndpoints(endpoints => {

        endpoints.MapEndpoint<SumEndpoint>("/sum/{count:int=1000000000}");

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

```

AddDistributedMemoryCache is the same method I used in Chapter 16 to provide the data store for session data. This is one of the three methods used to select an implementation for the IDistributedCache service, as described in Table 17-5.

Table 17-5. The Cache Service Implementation Methods

Name	Description
AddDistributedMemoryCache	This method sets up an in-memory cache.
AddDistributedSqlServerCache	This method sets up a cache that stores data in SQL Server and is available when the Microsoft.Extensions.Caching.SqlServer package is installed. See the “Caching Responses” section for details.
AddStackExchangeRedisCache	This method sets up a Redis cache and is available when the Microsoft.Extensions.Caching.Redis package is installed.

Listing 17-8 uses the AddDistributedMemoryCache method to create an in-memory cache as the implementation for the IDistributedCache service. This cache is configured using the MemoryCacheOptions class, whose most useful properties are described in Table 17-6.

Table 17-6. Useful MemoryCacheOptions Properties

Name	Description
ExpirationScanFrequency	This property is used to set a TimeSpan that determines how often the cache scans for expired items.
SizeLimit	This property specifies the maximum number of items in the cache. When the size is reached, the cache will eject items.
CompactionPercentage	This property specifies the percentage by which the size of the cache is reduced when SizeLimit is reached.

The statement in Listing 17-8 uses the SizeLimit property to restrict the cache to 200 items. Care must be taken when using an in-memory cache to find the right balance between allocating enough memory for the cache to be effective without exhausting server resources.

To see the effect of the cache, restart ASP.NET Core and request the `http://localhost:5000/sum` URL. Reload the browser, and you will see that only one of the timestamps will change, as shown in Figure 17-4. This is because the cache has provided the calculation response, which allows the endpoint to produce the result without having to repeat the calculation.

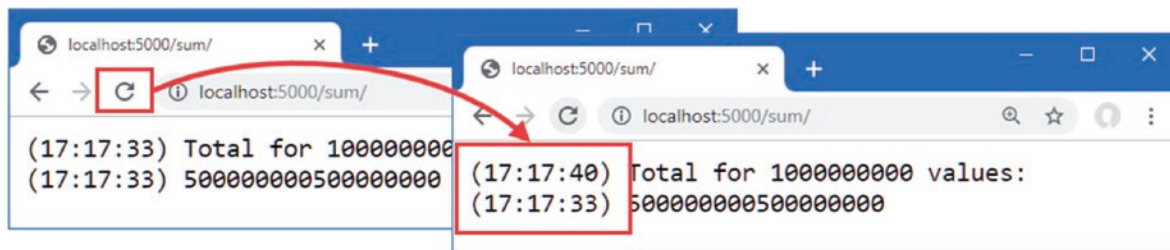


Figure 17-4. Caching data values

If you wait for two minutes and then reload the browser, then both timestamps will change because the cached result will have been ejected, and the endpoint will have to perform the calculation to produce the result.

Using a Shared and Persistent Data Cache

The cache created by the `AddDistributedMemoryCache` method isn't distributed, despite the name. The items are stored in memory as part of the ASP.NET Core process, which means that applications that run on multiple servers or containers don't share cached data. It also means that the contents of the cache are lost when ASP.NET Core is stopped.

The `AddDistributedSqlServerCache` method stores the cache data in a SQL Server database, which can be shared between multiple ASP.NET Core servers and which stores the data persistently.

The first step is to create a database that will be used to store the cached data. You can store the cached data alongside the application's other data, but for this chapter, I am going to use a separate database, which will be named `CacheDb`. You can create the database using Azure Data Studio or SQL Server Management Studio, both of which are available for free from Microsoft. Databases can also be created from the command line using `sqlcmd`. Open a new PowerShell command prompt and run the command shown in Listing 17-9 to connect to the LocalDB server.

■ **Tip** The `sqlcmd` tool should have been installed as part of the Visual Studio workload or as part of the SQL Server Express installation. If it has not been installed, then you can download an installer from <https://docs.microsoft.com/en-us/sql/tools/sqlcmd-utility?view=sql-server-2017>.

Listing 17-9. Connecting to the Database

```
sqlcmd -S "(localdb)\MSSQLLocalDB"
```

Pay close attention to the argument that specifies the database. There is one backslash, which is followed by `MSSQLLocalDB`. It can be hard to spot the repeated letters: `MS-SQL-LocalDB` (but without the hyphens).

When the connection has been established, you will see a `1>` prompt. Enter the commands shown in Listing 17-10 and press the Enter key after each command.

■ **Caution** If you are using Visual Studio, you must apply the updates for SQL Server described in Chapter 2. The version of SQL Server that is installed by default when you install Visual Studio cannot create LocalDB databases.

Listing 17-10. Creating the Database

```
CREATE DATABASE CacheDb
GO
```

If no errors are reported, then enter `exit` and press Enter to terminate the connection. The next step is to run the command shown in Listing 17-11 to create a table in the new database, which uses a global .NET Core tool to prepare the database.

■ **Tip** If you need to reset the cache database, use the command in Listing 17-9 to open a connection and use the command `DROP DATABASE CacheDB`. You can then re-create the database using the commands in Listing 17-10.

Listing 17-11. Creating the Cache Database Table

```
dotnet sql-cache create "Server=(localdb)\MSSQLLocalDB;Database=CacheDb" dbo DataCache
```

The arguments for this command are the connection string that specifies the database, the schema, and the name of the table that will be used to store the cached data. Enter the command on a single line and press Enter. It will take a few seconds for the tool to connect to the database. If the process is successful, you will see the following message:

```
Table and index were created successfully.
```

Creating the Persistent Cache Service

Now that the database is ready, I can create the service that will use it to store cached data. To add the NuGet package required for SQL Server caching support, open a new PowerShell command prompt, navigate to the Platform project folder, and run the command shown in Listing 17-12. (If you are using Visual Studio, you can add the package by selecting Project ► Manage Nuget Packages.)

Listing 17-12. Adding a Package to the Project

```
dotnet add package Microsoft.Extensions.Caching.SqlServer --version 3.1.1
```

The next step is to define a connection string, which describes the database connection in the JSON configuration file, as shown in Listing 17-13.

■ **Note** The cache created by the `AddDistributedSqlServerCache` method is distributed, meaning that multiple applications can use the same database and share cache data. If you are deploying the same application to multiple servers or containers, all instances will be able to share cached data. If you are sharing a cache between different applications, then you should pay close attention to the keys you use to ensure that applications receive the data types they expect.

Listing 17-13. Defining a Connection String in the `appsettings.json` File in the Platform Folder

```
{
  "Location": {
    "CityName": "Buffalo"
  },

```

```

"Logging": {
  "LogLevel": {
    "Default": "Information",
    "Microsoft": "Warning",
    "Microsoft.Hosting.Lifetime": "Information"
  }
},
"AllowedHosts": "*",
"ConnectionStrings": {
  "CacheConnection": "Server=(localdb)\\MSSQLLocalDB;Database=CacheDb"
}
}

```

Notice that the connection string uses two backslash characters (\\) to escape the character in the JSON file. Listing 17-14 change the implementation for the cache service in the Startup class to use SQL Server with the connection string from Listing 17-13.

Listing 17-14. Using a Persistent Data Cache in the Startup.cs File in the Platform Folder

```

using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;

namespace Platform {
  public class Startup {

    public Startup(IConfiguration config) {
      Configuration = config;
    }

    private IConfiguration Configuration {get; set;}

    public void ConfigureServices(IServiceCollection services) {

      services.AddDistributedSqlServerCache(opts => {
        opts.ConnectionString
            = Configuration["ConnectionStrings:CacheConnection"];
        opts.SchemaName = "dbo";
        opts.TableName = "DataCache";
      });
    }

    public void Configure(IApplicationBuilder app) {
      app.UseDeveloperExceptionPage();
      app.UseStaticFiles();
      app.UseRouting();
      app.UseEndpoints(endpoints => {

        endpoints.MapEndpoint<SumEndpoint>("/sum/{count:int=1000000000}");

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

```

The `IConfiguration` service is used to access the connection string from the application's configuration data. The cache service is created using the `AddDistributedSqlServerCache` method and is configured using an instance of the `SqlServerCacheOptions` class, whose most useful properties are described in Table 17-7.

Table 17-7. Useful `SqlServerCacheOptions` Properties

Name	Description
<code>ConnectionString</code>	This property specifies the connection string, which is conventionally stored in the JSON configuration file and accessed through the <code>IConfiguration</code> service.
<code>SchemaName</code>	This property specifies the schema name for the cache table.
<code>TableName</code>	This property specifies the name of the cache table.
<code>ExpiredItemsDeletionInterval</code>	This property specifies how often the table is scanned for expired items. The default is 30 minutes.
<code>DefaultSlidingExpiration</code>	This property specifies how long an item remains unread in the cache before it expires. The default is 20 minutes.

The listing uses the `ConnectionString`, `SchemaName`, and `TableName` properties to configure the cache middleware to use the database table. Restart ASP.NET Core and use a browser to request the `http://localhost:5000/sum` URL. There is no change in the response produced by the application, which is shown in Figure 17-4, but you will find that the cached responses are persistent and will be used even when you restart ASP.NET Core.

CACHING SESSION-SPECIFIC DATA VALUES

When you use the `IDistributedCache` service, the data values are shared between all requests. If you want to cache different data values for each user, then you can use the session middleware described in Chapter 16. The session middleware relies on the `IDistributedCache` service to store its data, which means that session data will be stored persistently and be available to a distributed application when the `AddDistributedSqlServerCache` method is used.

Caching Responses

An alternative to caching individual data items is to cache entire responses, which can be a useful approach if a response is expensive to compose and is likely to be repeated. Caching responses requires the addition of a service and a middleware component, as shown in Listing 17-15.

Listing 17-15. Configuring Response Caching in the `Startup.cs` File in the Platform Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Platform.Services;

namespace Platform {
    public class Startup {

        public Startup(IConfiguration config) {
            Configuration = config;
        }

        private IConfiguration Configuration {get; set;}
    }
}
```



```

public void ConfigureServices(IServiceCollection services) {
    services.AddDistributedSqlServerCache(opts => {
        opts.ConnectionString
            = Configuration["ConnectionStrings:CacheConnection"];
        opts.SchemaName = "dbo";
        opts.TableName = "DataCache";
    });
    services.AddResponseCaching();
    services.AddSingleton<IResponseFormatter, HtmlResponseFormatter>();
}

public void Configure(IApplicationBuilder app) {
    app.UseDeveloperExceptionPage();
    app.UseResponseCaching();
    app.UseStaticFiles();
    app.UseRouting();
    app.UseEndpoints(endpoints => {

        endpoints.MapEndpoint<SumEndpoint>("/sum/{count:int=1000000000}");

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

```

The `AddResponseCaching` method is used in the `ConfigureServices` method to set up the service used by the cache. The middleware component is added with the `UseResponseCaching` method, which should be called before any endpoint or middleware that needs its responses cached.

I have also defined the `IResponseFormatter` service, which I used to explain how dependency injection works in Chapter 14. Response caching is used only in certain circumstances, and, as I explain shortly, demonstrating the feature requires an HTML response.

■ **Note** The response caching feature does not use the `IDistributedCache` service. Responses are cached in memory and are not distributed.

In Listing 17-16, I have updated the `SumEndpoint` class so that it requests response caching instead of caching just a data value.

Listing 17-16. Using Response Caching in the `SumEndpoint.cs` File in the Platform Folder

```

using Microsoft.AspNetCore.Http;
using System;
using System.Threading.Tasks;
using Microsoft.Extensions.Caching.Distributed;
using Microsoft.AspNetCore.Routing;
using Platform.Services;

namespace Platform {

    public class SumEndpoint {

        public async Task Endpoint(HttpContext context, IDistributedCache cache,
            IResponseFormatter formatter, LinkGenerator generator) {

```

```

int count = int.Parse((string)context.Request.RouteValues["count"]);
long total = 0;
for (int i = 1; i <= count; i++) {
    total += i;
}
string totalString = $"{DateTime.Now.ToLongTimeString()} {total}";

context.Response.Headers["Cache-Control"] = "public, max-age=120";

string url = generator.GetPathByRouteValues(context, null,
    new { count = count });

await formatter.Format(context,
    $"<div>{DateTime.Now.ToLongTimeString()} Total for {count}"
    + $" values:</div><div>{totalString}</div>"
    + $"<a href={url}>Reload</a>");
}
}
}

```

Some of the changes to the endpoint enable response caching, but others are just to demonstrate that it is working. For enabling response caching, the important statement is the one that adds a header to the response, like this:

```

...
context.Response.Headers["Cache-Control"] = "public, max-age=120";
...

```

The Cache-Control header is used to control response caching. The middleware will only cache responses that have a Cache-Control header that contains the public directive. The max-age directive is used to specify the period that the response can be cached for, expressed in seconds. The Cache-Control header used in Listing 17-16 enables caching and specifies that responses can be cached for two minutes.

Enabling response caching is simple, but checking that it is working requires care. When you reload the browser window or press Return in the URL bar, browsers will include a Cache-Control header in the request that sets the max-age directive to zero, which bypasses the response cache and causes a new response to be generated by the endpoint. The only reliable way to request a URL without the Cache-Control header is to navigate using an HTML anchor element, which is why the endpoint in Listing 17-16 uses the IResponseFormatter service to generate an HTML response and uses the LinkGenerator service to create a URL that can be used in the anchor element's href attribute.

To check the response cache, restart ASP.NET Core and use the browser to request `http://localhost:5000/sum`. Once the response has been generated, click the Reload link to request the same URL. You will see that neither of the timestamps in the response change, indicating that the entire response has been cached, as shown in Figure 17-5.

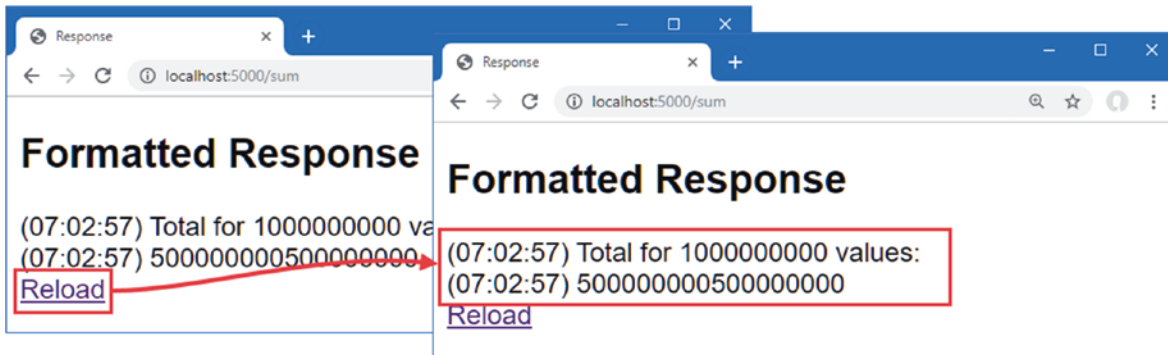


Figure 17-5. Caching responses

The Cache-Control header can be combined with the Vary header to provide fine-grained control over which requests are cached. See <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Cache-Control> and <https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers/Vary> for details of the features provided by both headers.

COMPRESSING RESPONSES

ASP.NET Core includes middleware that will compress responses for browsers that have indicated they can handle compressed data. The middleware is added to the pipeline with the `UseResponseCompression` method. Compression is a trade-off between the server resources required for compression and the bandwidth required to deliver content to the client, and it should not be switched on without testing to determine the performance impact.

Using Entity Framework Core

Not all data values are produced directly by the application, and most projects will need to access data in a database. Entity Framework Core is well-integrated into the ASP.NET Core platform, with good support for creating a database from C# classes and for creating C# classes to represent an existing database. In the sections that follow, I demonstrate the process for creating a simple data model, using it to create a database, and querying that database in an endpoint.

WORKING WITH ENTITY FRAMEWORK CORE

The most common complaint about Entity Framework Core is poor performance. When I review projects that have Entity Framework Core performance issues, the problem is almost always because the development team has treated Entity Framework Core as a black box and not paid attention to the SQL queries that are sent to the database. Not all LINQ features can be translated into SQL, and the most common problem is a query that retrieves large amounts of data from the database, which is then discarded after it has been reduced to produce a single value.

Using Entity Framework Core requires a good understanding of SQL and ensuring that the LINQ queries made by the application are translated into efficient SQL queries. There are rare applications that have high-performance data requirements that cannot be met by Entity Framework Core, but that isn't the case for most typical web applications.

That is not to say that Entity Framework Core is perfect. It has its quirks and requires an investment in time to become proficient. If you don't like the way that Entity Framework Core works, then you may prefer to use an alternative, such as Dapper (<https://github.com/StackExchange/Dapper>). But if your issue is that queries are not being performed fast enough, then you should spend some time exploring how those queries are being processed, which you can do using the techniques described in the remainder of the chapter.

Installing Entity Framework Core

Entity Framework Core requires a global tool package that is used to manage databases from the command line and to manage packages for the project that provide data access. To install the tools package, open a new PowerShell command prompt and run the commands shown in Listing 17-17.

Listing 17-17. Installing the Entity Framework Core Global Tool Package

```
dotnet tool uninstall --global dotnet-ef
dotnet tool install --global dotnet-ef --version 3.1.1
```

The first command removes any existing version of the `dotnet-ef` package, and the second command installs the version required for the examples in this book. This package provides the `dotnet ef` commands that you will see in later examples. To ensure the package is working as expected, run the command shown in Listing 17-18.

Listing 17-18. Testing the Entity Framework Core Global Tool

```
dotnet ef --help
```

This command shows the help message for the global tool and produces the following output:

```
Entity Framework Core .NET Command-line Tools 3.1.1
Usage: dotnet ef [options] [command]
Options:
  --version          Show version information
  -h|--help         Show help information
  -v|--verbose      Show verbose output.
  --no-color        Don't colorize output.
  --prefix-output   Prefix output with level.
Commands:
  database          Commands to manage the database.
  dbcontext         Commands to manage DbContext types.
  migrations        Commands to manage migrations.
Use "dotnet ef [command] --help" for more information about a command.
```

Entity Framework Core also requires packages to be added to the project. If you are using Visual Studio Code or prefer working from the command line, navigate to the Platform project folder (the folder that contains the Platform.csproj file) and run the commands shown in Listing 17-19.

Listing 17-19. Adding Entity Framework Core Packages to the Project

```
dotnet add package Microsoft.EntityFrameworkCore.Design --version 3.1.1
dotnet add package Microsoft.EntityFrameworkCore.SqlServer --version 3.1.1
```

Creating the Data Model

For this chapter, I am going to define the data model using C# classes and use Entity Framework Core to create the database and schema. Create the Platform/Models folder and add to it a class file called Calculation.cs with the contents shown in Listing 17-20.

Listing 17-20. The Contents of the Calculation.cs File in the Platform/Models Folder

```
namespace Platform.Models {
    public class Calculaton {
        public long Id { get; set; }
        public int Count { get; set; }
        public long Result { get; set; }
    }
}
```

You can see more complex data models in other chapters, but for this example, I am going to keep with the theme of this chapter and model the calculation performed in earlier examples. The Id property will be used to create a unique key for each object stored in the database, and the Count and Result properties will describe a calculation and its result.

Entity Framework Core uses a context class that provides access to the database. Add a file called CalculationContext.cs to the Platform/Models folder with the content shown in Listing 17-21.

Listing 17-21. The Contents of the CalculationContext.cs File in the Platform/Models Folder

```
using Microsoft.EntityFrameworkCore;

namespace Platform.Models {

    public class CalculationContext: DbContext {

        public CalculationContext(DbContextOptions<CalculationContext> opts)
            : base(opts) {}

        public DbSet<Calcalaton> Calculations { get; set; }
    }
}
```

The CalculationContext class defines a constructor that is used to receive an options object that is passed on to the base constructor. The Calculations property provides access to the Calculation objects that Entity Framework Core will retrieve from the database.

Configuring the Database Service

Access to the database is provided through a service that is configured in the Startup class, as shown in Listing 17-22.

Listing 17-22. Configuring the Data Service in the Startup.cs File in the Platform Folder

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Platform.Services;
using Microsoft.EntityFrameworkCore;
using Platform.Models;

namespace Platform {
    public class Startup {

        public Startup(IConfiguration config) {
            Configuration = config;
        }

        private IConfiguration Configuration {get; set;}

        public void ConfigureServices(IServiceCollection services) {
            services.AddDistributedSqlServerCache(opts => {
                opts.ConnectionString
                    = Configuration["ConnectionStrings:CacheConnection"];
                opts.SchemaName = "dbo";
                opts.TableName = "DataCache";
            });
            services.AddResponseCaching();
            services.AddSingleton<IResponseFormatter, HtmlResponseFormatter>();

            services.AddDbContext<CalculationContext>(opts => {
                opts.UseSqlServer(Configuration["ConnectionStrings:CalcConnection"]);
            });
        }
    }
}
```

```

public void Configure(IApplicationBuilder app) {
    app.UseDeveloperExceptionPage();
    app.UseResponseCaching();
    app.UseStaticFiles();
    app.UseRouting();
    app.UseEndpoints(endpoints => {

        endpoints.MapEndpoint<SumEndpoint>("/sum/{count:int=1000000000}");

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

```

The `AddDbContext` method creates a service for an Entity Framework Core context class. The method receives an options object that is used to select the database provider, which is done with the `UseSqlServer` method. The `IConfiguration` service is used to get the connection string for the database, which is defined in Listing 17-23.

Listing 17-23. Defining a Connection String in the `appsettings.json` File in the Platform Folder

```

{
  "Location": {
    "CityName": "Buffalo"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information",
      "Microsoft.EntityFrameworkCore: Information"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "CacheConnection": "Server=(localdb)\\MSSQLLocalDB;Database=CacheDb",
    "CalcConnection: Server=(localdb)\\MSSQLLocalDB;Database=CalcDb"
  }
}

```

The listing also sets the logging level for the `Microsoft.EntityFrameworkCore` category, which will show the SQL statements that are used by Entity Framework Core to query the database.

■ **Tip** Set the `MultipleActiveResultSets` option to `True` for connection strings that will be used to make queries with multiple result sets. You can see an example of this option set in the connection strings for the `SportsStore` project in Chapter 7.

Creating and Applying the Database Migration

Entity Framework Core manages the relationship between data model classes and the database using a feature called *migrations*. When changes are made to the model classes, a new migration is created that modifies the database to match those changes. To create the initial migration, which will create a new database and prepare it to store `Calculation` objects, open a new PowerShell command prompt, navigate to the folder that contains the `Platform.csproj` file, and run the command shown in Listing 17-24.

Listing 17-24. Creating a Migration

```
dotnet ef migrations add Initial
```

The `dotnet ef` commands relate to Entity Framework Core. The command in Listing 17-24 creates a new migration named `Initial`, which is the name conventionally given to the first migration for a project. You will see that a `Migrations` folder has been added to the project and that it contains class files whose statements prepare the database so that it can store the objects in the data model. To apply the migration, run the command shown in Listing 17-25 in the `Platform` project folder.

Listing 17-25. Applying a Migration

```
dotnet ef database update
```

This command executes the commands in the migration created in Listing 17-24 and uses them to prepare the database, which you can see in the SQL statements written to the command prompt.

Seeding the Database

Most applications require some seed data, especially during development. Entity Framework Core does provide a database seeding feature, but it is of limited use for most projects because it doesn't allow data to be seeded where the database allocates unique keys to the objects it stores. This is an important feature in most data models because it means the application doesn't have to worry about allocating unique key values.

A more flexible approach is to use the regular Entity Framework Core features to add seed data to the database. Create a file called `SeedData.cs` in the `Platform/Models` folder with the code shown in Listing 17-26.

Listing 17-26. The Contents of the `SeedData.cs` File in the `Platform/Models` Folder

```
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.Logging;
using System.Collections.Generic;
using System.Linq;

namespace Platform.Models {
    public class SeedData {
        private CalculationContext context;
        private ILogger<SeedData> logger;

        private static Dictionary<int, long> data
            = new Dictionary<int, long>() {
                {1, 1}, {2, 3}, {3, 6}, {4, 10}, {5, 15},
                {6, 21}, {7, 28}, {8, 36}, {9, 45}, {10, 55}
            };

        public SeedData(CalculationContext dataContext, ILogger<SeedData> log) {
            context = dataContext;
            logger = log;
        }

        public void SeedDatabase() {
            context.Database.Migrate();
            if (context.Calculations.Count() == 0) {
                logger.LogInformation("Preparing to seed database");
                context.Calculations!.AddRange(data.Select(kvp => new Calculaton() {
                    Count = kvp.Key, Result = kvp.Value
                }));
            }
        }
    }
}
```

```

        context.SaveChanges();
        logger.LogInformation("Database seeded");
    } else {
        logger.LogInformation("Database not seeded");
    }
}
}
}
}
}

```

The `SeedData` class declares constructor dependencies on the `CalculationContext` and `ILogger<T>` types, which are used in the `SeedDatabase` method to prepare the database. The context's `Database.Migrate` method is used to apply any pending migrations to the database, and the `Calculations` property is used to store new data using the `AddRange` method, which accepts a sequence of `Calculation` objects.

The new objects are stored in the database using the `SaveChanges` method. To use the `SeedData` class, make the changes shown in Listing 17-27 to the `Startup` class.

Listing 17-27. Enabling Database Seeding in the `Startup.cs` File in the Platform Folder

```

using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Configuration;
using Platform.Services;
using Microsoft.EntityFrameworkCore;
using Platform.Models;
using Microsoft.Extensions.Hosting;
using Microsoft.AspNetCore.Hosting;

namespace Platform {
    public class Startup {

        public Startup(IConfiguration config) {
            Configuration = config;
        }

        private IConfiguration Configuration {get; set;}

        public void ConfigureServices(IServiceCollection services) {
            services.AddDistributedSqlServerCache(opts => {
                opts.ConnectionString
                    = Configuration["ConnectionStrings:CacheConnection"];
                opts.SchemaName = "dbo";
                opts.TableName = "DataCache";
            });
            services.AddResponseCaching();
            services.AddSingleton<IResponseFormatter, HtmlResponseFormatter>();

            services.AddDbContext<CalculationContext>(opts => {
                opts.UseSqlServer(Configuration["ConnectionStrings:CalcConnection"]);
            });
            services.AddTransient<SeedData>();
        }

        public void Configure(IApplicationBuilder app,
            IHostApplicationLifetime lifetime, IWebHostEnvironment env,
            SeedData seedData) {
            app.UseDeveloperExceptionPage();
            app.UseResponseCaching();
        }
    }
}

```


Using Data in an Endpoint

Endpoints and middleware components access Entity Framework Core data by declaring a dependency on the context class and using its `DbSet<T>` properties to perform LINQ queries. The LINQ queries are translated into SQL and sent to the database. The tabular data received from the database is used to create data model objects that are used to produce responses. Listing 17-29 updates the `SumEndpoint` class to use Entity Framework Core.

Listing 17-29. Using a Database in the `SumEndpoint.cs` File in the Platform Folder

```
using Microsoft.AspNetCore.Http;
using System;
using System.Threading.Tasks;
using Microsoft.Extensions.Caching.Distributed;
using Microsoft.AspNetCore.Routing;
using Platform.Services;
using Platform.Models;
using System.Linq;

namespace Platform {

    public class SumEndpoint {

        public async Task Endpoint(HttpContext context,
            CalculationContext dataContext) {
            int count = int.Parse((string)context.Request.RouteValues["count"]);
            long total = dataContext.Calculations
                .FirstOrDefault(c => c.Count == count)?.Result ?? 0;
            if (total == 0) {
                for (int i = 1; i <= count; i++) {
                    total += i;
                }
                dataContext.Calculations!
                    .Add(new Calculaton() { Count = count, Result = total});
                await dataContext.SaveChangesAsync();
            }
            string totalString = $"({ DateTime.Now.ToLongTimeString() }) {total}";
            await context.Response.WriteAsync(
                 $"({DateTime.Now.ToLongTimeString()}) Total for {count}"
                 + $" values:\n{totalString}\n");
        }
    }
}
```

The endpoint uses the LINQ `FirstOrDefault` to search for a stored `Calculation` object for the calculation that has been requested like this:

```
...
dataContext.Calculations.FirstOrDefault(c => c.Count == count)?.Result ?? 0;
...
```

If an object has been stored, it is used to prepare the response. If not, then the calculation is performed, and a new `Calculation` object is stored by these statements:

```
...
dataContext.Calculations!.Add(new Calculaton() { Count = count, Result = total});
await dataContext.SaveChangesAsync();
...
```

The `Add` method is used to tell Entity Framework Core that the object should be stored, but the update isn't performed until the `SaveChangesAsync` method is called. To see the effect of the changes, restart ASP.NET Core MVC (without the `INITDB` argument if you are using the command line) and request the `http://localhost:5000/sum/10` URL. This is one of the calculations with which the database has been seeded, and you will be able to see the query sent to the database in the logging messages produced by the application.

```
...
Executing DbCommand [Parameters=[@__count_0='?' (DbType = Int32)],
  CommandType='Text', CommandTimeout='30']
SELECT TOP(1) [c].[Id], [c].[Count], [c].[Result]
FROM [Calculations] AS [c]
WHERE ([c].[Count] = @__count_0) AND @__count_0 IS NOT NULL
...
```

If you request `http://localhost:5000/sum/100`, the database will be queried, but no result will be found. The endpoint performs the calculation and stores the result in the database before producing the result shown in Figure 17-6.



Figure 17-6. Performing a calculation

Once a result has been stored in the database, subsequent requests for the same URL will be satisfied using the stored data. You can see the SQL statement used to store the data in the logging output produced by Entity Framework Core.

```
...
Executing DbCommand [Parameters=[@p0='?' (DbType = Int32), @p1='?' (DbType = Int64)],
  CommandType='Text', CommandTimeout='30']
SET NOCOUNT ON;
INSERT INTO [Calculations] ([Count], [Result])
VALUES (@p0, @p1);
SELECT [Id]
FROM [Calculations]
WHERE @@ROWCOUNT = 1 AND [Id] = scope_identity();
...
```

■ **Note** Notice that the data retrieved from the database is not cached and that each request leads to a new SQL query. Depending on the frequency and complexity of the queries you require, you may want to cache data values or responses using the techniques described earlier in the chapter.

Enabling Sensitive Data Logging

Entity Framework Core doesn't include parameter values in the logging messages it produces, which is why the logging output contains question marks, like this:

```
...
Executing DbCommand [Parameters=[@__count_0='?' (DbType = Int32)], CommandType='Text', CommandTimeout='30']
...
```

The data is omitted as a precaution to prevent sensitive data from being stored in logs. If you are having problems with queries and need to see the values sent to the database, then you can use the `EnableSensitiveDataLogging` method when configuring the database context, as shown in Listing 17-30.

Listing 17-30. Enabling Sensitive Data Logging in the Startup.cs File in the Platform Folder

```
...
services.AddDbContext<CalculationContext>(opts => {
    opts.UseSqlServer(Configuration["ConnectionStrings:CalcConnection"]);
    opts.EnableSensitiveDataLogging(true);
});
...
```

Restart ASP.NET Core MVC and request the `http://localhost:5000/sum/100` URL again. When the request is handled, Entity Framework Core will include parameter values in the logging message it creates to show the SQL query, like this:

```
...
Executing DbCommand [Parameters=[@__count_0='100'], CommandType='Text',
    CommandTimeout='30']
SELECT TOP(1) [c].[Id], [c].[Count], [c].[Result]
FROM [Calculations] AS [c]
WHERE ([c].[Count] = @__count_0) AND @__count_0 IS NOT NULL
...
```

This is a feature that should be used with caution because logs are often accessible by people who would not usually have access to the sensitive data that applications handle, such as credit card numbers and account details.

Summary

In this chapter, I demonstrated the ASP.NET Core platform features that are useful for working with data. I showed you how to cache individual data values, both locally and in a shared database. I also showed you how to cache responses and how to read and write data in a database using Entity Framework Core. In Part 3 of this book, I explain how to build on the ASP.NET Core platform to create web applications.

PART III



ASP.NET Core Applications

CHAPTER 18

Creating the Example Project

In this chapter, you will create the example project used throughout this part of the book. The project contains a simple data model, a client-side package for formatting HTML content, and a simple request pipeline.

Creating the Project

Open a new PowerShell command prompt from the Windows Start menu and run the commands shown in Listing 18-1.

■ **Tip** You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/apress/pro-asp.net-core-3>. See Chapter 1 for how to get help if you have problems running the examples.

Listing 18-1. Creating the Project

```
dotnet new globaljson --sdk-version 3.1.101 --output WebApp
dotnet new web --no-https --output WebApp --framework netcoreapp3.1
dotnet new sln -o WebApp

dotnet sln WebApp add WebApp
```

If you are using Visual Studio, open the `WebApp.sln` file in the `WebApp` folder. Select **Project** ► **Platform Properties**, navigate to the **Debug** page, and change the **App URL** field to `http://localhost:5000`, as shown in Figure 18-1. This changes the port that will be used to receive HTTP requests. Select **File** ► **Save All** to save the configuration changes.

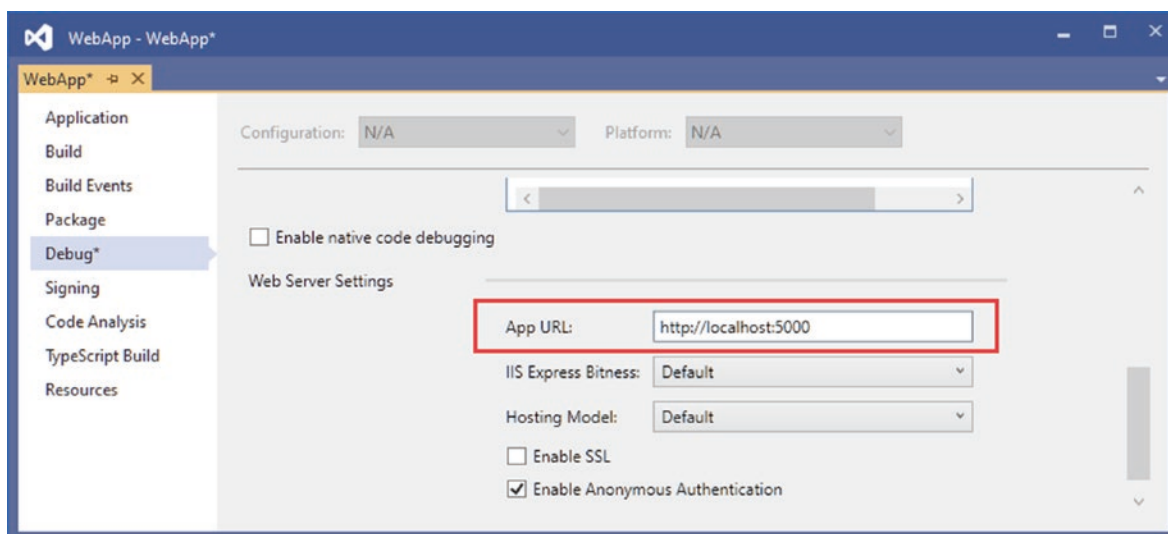


Figure 18-1. Changing the HTTP port

If you are using Visual Studio Code, open the `WebApp` folder. Click the Yes button when prompted to add the assets required for building and debugging the project, as shown in Figure 18-2.

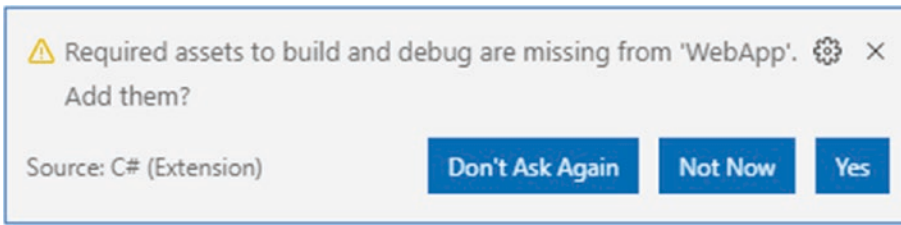


Figure 18-2. Adding project assets

Adding a Data Model

A data model helps demonstrate the different ways that web applications can be built using ASP.NET Core, showing how complex responses can be composed and how data can be submitted by the user. In the sections that follow, I create a simple data model and use it to create the database schema that will be used to store the application's data.

Adding NuGet Packages to the Project

The data model will use Entity Framework Core to store and query data in a SQL Server LocalDB database. To add the NuGet packages for Entity Framework Core, use a PowerShell command prompt to run the commands shown in Listing 18-2 in the `WebApp` project folder.

Listing 18-2. Adding Packages to the Project

```
dotnet add package Microsoft.EntityFrameworkCore.Design --version 3.1.1
dotnet add package Microsoft.EntityFrameworkCore.SqlServer --version 3.1.1
```

If you are using Visual Studio, you can add the packages by selecting `Project ► Manage NuGet Packages`. Take care to choose the correct version of the packages to add to the project.

If you have not followed the examples in earlier chapters, you will need to install the global tool package that is used to create and manage Entity Framework Core migrations. Run the commands shown in Listing 18-3 to remove any existing version of the package and install the version required for this book. (You can skip these commands if you installed this version of the tools package in earlier chapters.)

Listing 18-3. Installing a Global Tool Package

```
dotnet tool uninstall --global dotnet-ef
dotnet tool install --global dotnet-ef --version 3.1.1
```

Creating the Data Model

The data model for this part of the book will consist of three related classes: `Product`, `Supplier`, and `Category`. Create a new folder named `Models` and add to it a class file named `Category.cs`, with the contents shown in Listing 18-4.

Listing 18-4. The Contents of the Category.cs File in the Models Folder

```
using System.Collections.Generic;

namespace WebApp.Models {
    public class Category {

        public long CategoryId { get; set; }
        public string Name { get; set; }

        public IEnumerable<Product> Products { get; set; }
    }
}
```

Add a class called `Supplier.cs` to the Models folder and use it to define the class shown in Listing 18-5.

Listing 18-5. The Contents of the Supplier.cs File in the Models Folder

```
using System.Collections.Generic;

namespace WebApp.Models {
    public class Supplier {

        public long SupplierId { get; set; }
        public string Name { get; set; }
        public string City { get; set; }

        public IEnumerable<Product> Products { get; set; }
    }
}
```

Next, add a class named `Product.cs` to the Models folder and use it to define the class shown in Listing 18-6.

Listing 18-6. The Contents of the Product.cs File in the Models Folder

```
using System.ComponentModel.DataAnnotations.Schema;

namespace WebApp.Models {
    public class Product {

        public long ProductId { get; set; }

        public string Name { get; set; }
        [Column(TypeName = "decimal(8, 2)")]
        public decimal Price { get; set; }

        public long CategoryId { get; set; }
        public Category Category { get; set; }

        public long SupplierId { get; set; }
        public Supplier Supplier { get; set; }
    }
}
```

Each of the three data model classes defines a key property whose value will be allocated by the database when new objects are stored. There are also navigation properties that will be used to query for related data so that it will be possible to query for all the products in a specific category, for example.

The Price property has been decorated with the Column attribute, which specifies the precision of the values that will be stored in the database. There isn't a one-to-one mapping between C# and SQL numeric types, and the Column attribute tells Entity Framework Core which SQL type should be used in the database to store Price values. In this case, the decimal(8, 2) type will allow a total of eight digits, including two following the decimal point.

To create the Entity Framework Core context class that will provide access to the database, add a file called DataContext.cs to the Models folder and add the code shown in Listing 18-7.

Listing 18-7. The Contents of the DataContext.cs File in the Models Folder

```
using Microsoft.EntityFrameworkCore;

namespace WebApp.Models {
    public class DataContext: DbContext {

        public DataContext(DbContextOptions<DataContext> opts)
            : base(opts) { }

        public DbSet<Product> Products { get; set; }
        public DbSet<Category> Categories { get; set; }
        public DbSet<Supplier> Suppliers { get; set; }
    }
}
```

The context class defines properties that will be used to query the database for Product, Category, and Supplier data.

Preparing the Seed Data

Add a class called SeedData.cs to the Models folder and add the code shown in Listing 18-8 to define the seed data that will be used to populate the database.

Listing 18-8. The Contents of the SeedData.cs File in the Models Folder

```
using Microsoft.EntityFrameworkCore;
using System.Linq;

namespace WebApp.Models {
    public static class SeedData {

        public static void SeedDatabase(DataContext context) {
            context.Database.Migrate();
            if (context.Products.Count() == 0 && context.Suppliers.Count() == 0
                && context.Categories.Count() == 0) {

                Supplier s1 = new Supplier
                    { Name = "Splash Dudes", City = "San Jose" };
                Supplier s2 = new Supplier
                    { Name = "Soccer Town", City = "Chicago" };
                Supplier s3 = new Supplier
                    { Name = "Chess Co", City = "New York" };

                Category c1 = new Category { Name = "Watersports" };
                Category c2 = new Category { Name = "Soccer" };
                Category c3 = new Category { Name = "Chess" };

                context.Products.AddRange(
                    new Product { Name = "Kayak", Price = 275,
                        Category = c1, Supplier = s1 },
```

```

        new Product { Name = "Lifejacket", Price = 48.95m,
                      Category = c1, Supplier = s1},
        new Product { Name = "Soccer Ball", Price = 19.50m,
                      Category = c2, Supplier = s2},
        new Product { Name = "Corner Flags", Price = 34.95m,
                      Category = c2, Supplier = s2},
        new Product { Name = "Stadium", Price = 79500,
                      Category = c2, Supplier = s2},
        new Product { Name = "Thinking Cap", Price = 16,
                      Category = c3, Supplier = s3},
        new Product { Name = "Unsteady Chair", Price = 29.95m,
                      Category = c3, Supplier = s3},
        new Product { Name = "Human Chess Board", Price = 75,
                      Category = c3, Supplier = s3},
        new Product { Name = "Bling-Bling King", Price = 1200,
                      Category = c3, Supplier = s3}
    );
    context.SaveChanges();
}
}
}
}
}

```

The static `SeedDatabase` method ensures that all pending migrations have been applied to the database. If the database is empty, it is seeded with categories, suppliers, and products. Entity Framework Core will take care of mapping the objects into the tables in the database, and the key properties will be assigned automatically when the data is stored.

Configuring Entity Framework Core Services and Middleware

Make the changes to the `Startup` class shown in Listing 18-9, which configure Entity Framework Core and set up the `DataContext` services that will be used throughout this part of the book to access the database.

Listing 18-9. Preparing Services and Middleware in the `Startup.cs` File in the `WebApp` Folder

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using WebApp.Models;

namespace WebApp {
    public class Startup {

        public Startup(IConfiguration config) {
            Configuration = config;
        }

        public IConfiguration Configuration { get; set; }
    }
}

```

```

public void ConfigureServices(IServiceCollection services) {
    services.AddDbContext<DataContext>(opts => {
        opts.UseSqlServer(Configuration[
            "ConnectionStrings:ProductConnection"]);
        opts.EnableSensitiveDataLogging(true);
    });
}

public void Configure(IApplicationBuilder app, DataContext context) {

    app.UseDeveloperExceptionPage();
    app.UseRouting();

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

    SeedData.SeedDatabase(context);
}
}
}

```

To define the connection string that will be used for the application's data, add the configuration settings shown in Listing 18-10 in the `appsettings.json` file. The connection string should be entered on a single line.

Listing 18-10. Defining a Connection String in the `appsettings.json` File in the WebApp Folder

```

{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information",
      "Microsoft.EntityFrameworkCore": "Information"
    }
  },
  "AllowedHosts": "*",
  "ConnectionStrings": {
    "ProductConnection": "Server=(localdb)\\MSSQLLocalDB;Database=Products;MultipleActiveResultSets=True"
  }
}

```

In addition to the connection string, Listing 18-10 increases the logging detail for Entity Framework Core so that the SQL queries sent to the database are logged.

Creating and Applying the Migration

To create the migration that will set up the database schema, use a PowerShell command prompt to run the command shown in Listing 18-11 in the WebApp project folder.

Listing 18-11. Creating an Entity Framework Core Migration

```
dotnet ef migrations add Initial
```

Once the migration has been created, apply it to the database using the command shown in Listing 18-12.

Listing 18-12. Applying the Migration to the Database

```
dotnet ef database update
```

The logging messages displayed by the application will show the SQL commands that are sent to the database.

■ **Note** If you need to reset the database, then run the `dotnet ef database drop --force` command and then the command in Listing 18-12.

Adding the CSS Framework

Later chapters will demonstrate the different ways that HTML responses can be generated. Run the commands shown in Listing 18-13 to remove any existing version of the LibMan package and install the version used in this book. (You can skip these commands if you installed this version of LibMan in earlier chapters.)

Listing 18-13. Installing the LibMan Tool Package

```
dotnet tool uninstall --global Microsoft.Web.LibraryManager.Cli
dotnet tool install --global Microsoft.Web.LibraryManager.Cli --version 2.0.96
```

To add the Bootstrap CSS framework so that the HTML responses can be styled, run the commands shown in Listing 18-14 in the WebApp project folder.

Listing 18-14. Installing the Bootstrap CSS Framework

```
libman init -p cdnjs
libman install twitter-bootstrap@4.3.1 -d wwwroot/lib/twitter-bootstrap
```

Configuring the Request Pipeline

To define a simple middleware component that will be used to make sure the example project has been set up correctly, add a class file called `TestMiddleware.cs` to the WebApp folder and add the code shown in Listing 18-15.

Listing 18-15. The Contents of the TestMiddleware.cs File in the WebApp Folder

```
using Microsoft.AspNetCore.Http;
using System.Linq;
using System.Threading.Tasks;
using WebApp.Models;

namespace WebApp {
    public class TestMiddleware {
        private RequestDelegate nextDelegate;

        public TestMiddleware(RequestDelegate next) {
            nextDelegate = next;
        }
    }
}
```



```
        });  
    });  
    SeedData.SeedDatabase(context);  
}  
}
```

Running the Example Application

Start the application, either by selecting Start Without Debugging or Run Without Debugging from the Debug menu or by running the command shown in Listing 18-17 in the WebApp project folder.

Listing 18-17. Running the Example Application

```
dotnet run
```

Use a new browser tab and request `http://localhost:5000/test`, and you will see the response shown in Figure 18-3.

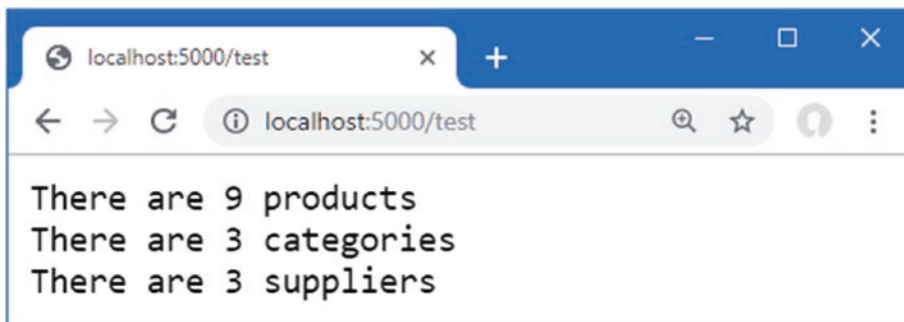


Figure 18-3. Running the example application

Summary

In this chapter, I created the example application that is used throughout this part of the book. The project was created with the Empty template, contains a data model that relies on Entity Framework Core, and is configured with a request pipeline that contains a simple test middleware component. In the next chapter, I show you how to create web services using ASP.NET Core.



Creating RESTful Web Services

Web services accept HTTP requests and generate responses that contain data. In this chapter, I explain how the features provided by the MVC Framework, which is an integral part of ASP.NET Core, can be used to build on the capabilities described in Part 2 to create web services.

The nature of web services means that some of the examples in this chapter are tested using command-line tools provided by PowerShell, and it is important to enter the commands exactly as shown. Chapter 20 introduces more sophisticated tools for working with web services, but the command-line approach is better suited to following examples in a book chapter, even if they can feel a little awkward as you type them in. Table 19-1 puts RESTful web services in context.

Table 19-1. *Putting RESTful Web Services in Context*

Question	Answer
What are they?	Web services provide access to an application's data, typically expressed in the JSON format.
Why are they useful?	Web services are most often used to provide rich client-side applications with data.
How are they used?	The combination of the URL and an HTTP method describes an operation that is handled by an action method defined by an ASP.NET Core controller.
Are there any pitfalls or limitations?	There is no widespread agreement about how web services should be implemented, and care must be taken to produce just the data the client expects.
Are there any alternatives?	There are a number of different approaches to providing clients with data, although RESTful web services are the most common.

Table 19-2 summarizes the chapter.

Table 19-2. *Chapter Summary*

Problem	Solution	Listing
Defining a web service	Create a controller with action methods that correspond to the operations that you require	1-14
Generating data sequences over time	Use the <code>IAsyncEnumerable<T></code> response, which will prevent the request thread from blocking while results are generated.	15
Preventing request values being used for sensitive data properties	Use a binding target to restrict the model binding process to only safe properties	16-18
Expressing nondata outcomes	Use action results to describe the response that ASP.NET Core should send	19-24
Validating data	Use the ASP.NET Core model binding and model validation features	25-27
Automatically validating requests	Use the <code>ApiController</code> attribute	28
Omitting null values from data responses	Map the data objects to filter out properties or configure the JSON serializer to ignore null properties	29-31

Preparing for This Chapter

In this chapter, I continue to use the WebApp project created in Chapter 18. To prepare for this chapter, drop the database by opening a new PowerShell command prompt, navigating to the folder that contains the WebApp.csproj file, and running the command shown in Listing 19-1.

■ **Tip** You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/apress/pro-asp.net-core-3>. See Chapter 1 for how to get help if you have problems running the examples.

Listing 19-1. Dropping the Database

```
dotnet ef database drop --force
```

Start the application by selecting Start Without Debugging or Run Without Debugging from the Debug menu or by running the command shown in Listing 19-2 in the project folder.

Listing 19-2. Starting the Example Application

```
dotnet run
```

Request the URL `http://localhost:5000/test` once ASP.NET Core has started, and you will see the response shown in Figure 19-1.

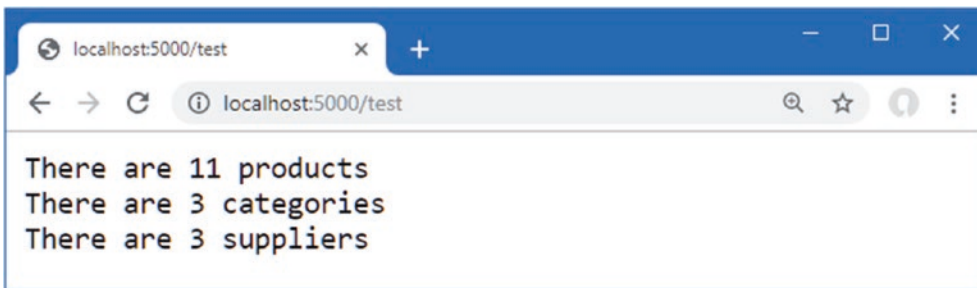


Figure 19-1. Running the example application

Understanding RESTful Web Services

Web services respond to HTTP requests with data that can be consumed by clients, such as JavaScript applications. There are no hard-and-fast rules for how web services should work, but the most common approach is to adopt the Representational State Transfer (REST) pattern. There is no authoritative specification for REST, and there is no consensus about what constitutes a RESTful web service, but there are some common themes that are widely used for web services. The lack of a detailed specification leads to endless disagreement about what REST means and how RESTful web services should be created, all of which can be safely ignored if the web services you create work for your projects.

Understanding Request URLs and Methods

The core premise of REST—and the only aspect for which there is broad agreement—is that a web service defines an API through a combination of URLs and HTTP methods such as GET and POST, which are also known as the HTTP *verbs*. The method specifies the type of operation, while the URL specifies the data object or objects that the operation applies to.

As an example, here is a URL that might identify a Product object in the example application:

```
/api/products/1
```

This URL may identify the Product object that has a value of 1 for its ProductId property. The URL identifies the Product, but it is the HTTP method that specifies what should be done with it. Table 19-3 lists the HTTP methods that are commonly used in web services and the operations they conventionally represent.

Table 19-3. HTTP Methods and Operations

HTTP Method	Description
GET	This method is used to retrieve one or more data objects.
POST	This method is used to create a new object.
PUT	This method is used to update an existing object.
PATCH	This method is used to update part of an existing object.
DELETE	This method is used to delete an object.

Understanding JSON

Most RESTful web services format the response data using the JavaScript Object Notation (JSON) format. JSON has become popular because it is simple and easily consumed by JavaScript clients. JSON is described in detail at www.json.org, but you don't need to understand every aspect of JSON to create web services because ASP.NET Core provides all the features required to create JSON responses.

UNDERSTANDING THE ALTERNATIVES TO RESTFUL WEB SERVICES

REST isn't the only way to design web services, and there are some popular alternatives. *GraphQL* is most closely associated with the React JavaScript framework, but it can be used more widely. Unlike REST web services, which provide specific queries through individual combinations of a URL and an HTTP method, GraphQL provides access to all an application's data and lets clients query for just the data they require in the format they require. GraphQL can be complex to set up—and can require more sophisticated clients—but the result is a more flexible web service that puts the developers of the client in control of the data they consume. GraphQL isn't supported directly by ASP.NET Core, but there are .NET implementations available. See <https://graphql.org> for more detail.

A new alternative is gRPC, a full remote procedure call framework that focuses on speed and efficiency. At the time of writing, gRPC cannot be used in web browsers, such as by the Angular or React framework, because browsers don't provide the fine-grained access that gRPC requires to formulate its HTTP requests.

Creating a Web Service Using a Custom Endpoint

As you learn about the facilities that ASP.NET Core provides for web services, it can be easy to forget they are built on the features described in Part 2. To create a simple web service, add a file named `WebServiceEndpoint.cs` to the `WebApp` folder and use it to define the class shown in Listing 19-3.

Listing 19-3. The Contents of the `WebServiceEndpoint.cs` File in the `WebApp` Folder

```
using Microsoft.AspNetCore.Http;
using Microsoft.AspNetCore.Routing;
using Microsoft.Extensions.DependencyInjection;
using System.Collections.Generic;
```

```

using System.Text.Json;
using WebApp.Models;

namespace Microsoft.AspNetCore.Builder {

    public static class WebServiceEndpoint {
        private static string BASEURL = "api/products";

        public static void MapWebService(this IEndpointRouteBuilder app) {

            app.MapGet($"{BASEURL}/{id}", async context => {
                long key = long.Parse(context.Request.RouteValues["id"] as string);
                DataContext data = context.RequestServices.GetService<DataContext>();
                Product p = data.Products.Find(key);
                if (p == null) {
                    context.Response.StatusCode = StatusCodes.Status404NotFound;
                } else {
                    context.Response.ContentType = "application/json";
                    await context.Response
                        .WriteAsync(JsonSerializer.Serialize<Product>(p));
                }
            });

            app.MapGet(BASEURL, async context => {
                DataContext data = context.RequestServices.GetService<DataContext>();
                context.Response.ContentType = "application/json";
                await context.Response.WriteAsync(JsonSerializer
                    .Serialize<IEnumerable<Product>>(data.Products));
            });

            app.MapPost(BASEURL, async context => {
                DataContext data = context.RequestServices.GetService<DataContext>();
                Product p = await
                    JsonSerializer.DeserializeAsync<Product>(context.Request.Body);
                await data.AddAsync(p);
                await data.SaveChangesAsync();
                context.Response.StatusCode = StatusCodes.Status200OK;
            });
        }
    }
}

```

The `MapWebService` extension method creates three routes that form a basic web service using only the features that have been described in earlier chapters. The routes match URLs that start with `/api`, which is the conventional URL prefix for web services. The endpoint for the first route receives a value from a segment variable that is used to locate a single `Product` object in the database. The endpoint for the second route retrieves all the `Product` objects in the database. The third endpoint handles POST requests and reads the request body to get a JSON representation of a new object to add to the database.

There are better ASP.NET Core features for creating web services, which you will see shortly, but the code in Listing 19-3 shows how the HTTP method and the URL can be combined to describe an operation. Listing 19-4 uses the `MapWebService` extension method to add the endpoints to the example application's routing configuration.

Listing 19-4. Adding Routes in the Startup.cs File in the WebApp Folder

```

...
public void Configure(IApplicationBuilder app, DataContext context) {
    app.UseDeveloperExceptionPage();
    app.UseRouting();
    app.UseMiddleware<TestMiddleware>();
}

```

```

app.UseEndpoints(endpoints => {
    endpoints.MapGet("/", async context => {
        await context.Response.WriteAsync("Hello World!");
    });
    endpoints.MapWebService();
});
SeedData.SeedDatabase(context);
}
...

```

To test the web service, restart ASP.NET Core and request `http://localhost:5000/api/products/1`. The request will be matched by the first route defined in Listing 19-4 and will produce the response shown on the left of Figure 19-2. Next, request `http://localhost:5000/api/products`, which will be matched by the second route and produce the response shown on the right of Figure 19-2.

■ **Note** The responses shown in the figure contain null values for the `Supplier` and `Category` properties because the LINQ queries do not include related data. See Chapter 20 for details.

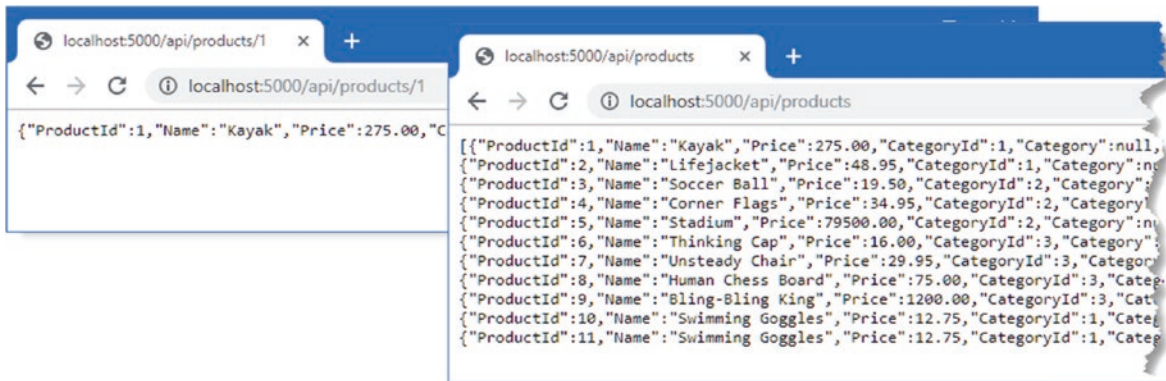


Figure 19-2. Web service response

Testing the third route requires a different approach because it isn't possible to send HTTP POST requests using the browser. Open a new PowerShell command prompt and run the command shown in Listing 19-5. It is important to enter the command exactly as shown because the `Invoke-RestMethod` command is fussy about the syntax of its arguments.

■ **Tip** You may receive an error when you use the `Invoke-RestMethod` or `Invoke-WebRequest` command to test the examples in this chapter if you have not performed the initial setup for Microsoft Edge or Internet Explorer. The problem can be fixed by running IE and selecting the initial configurations you require.

Listing 19-5. Sending a POST Request

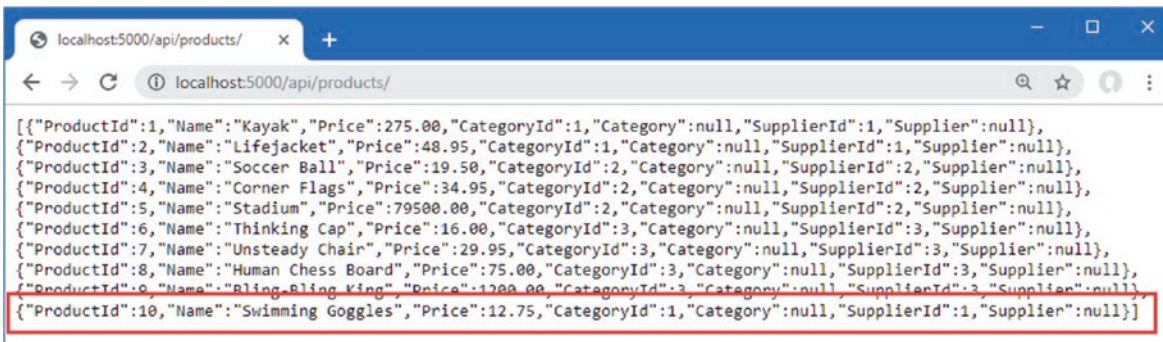
```

Invoke-RestMethod http://localhost:5000/api/products -Method POST -Body (@{ Name="Swimming Goggles";
Price=12.75; CategoryId=1; SupplierId=1} | ConvertTo-Json) -ContentType "application/json"

```

The command sends an HTTP POST command that is matched by the third route defined in Listing 19-5. The body of the request is a JSON-formatted object that is parsed to create a `Product`, which is then stored in the database. The JSON object included in the request contains values for the `Name`, `Price`, `CategoryId`, and `SupplierId` properties. The unique key for the object, which

is associated with the `ProductId` property, is assigned by the database when the object is stored. Use the browser to request the `http://localhost:5000/api/products` URL again, and you will see that the JSON response contains the new object, as shown in Figure 19-3.



```
[{"ProductId":1,"Name":"Kayak","Price":275.00,"CategoryId":1,"Category":null,"SupplierId":1,"Supplier":null},
{"ProductId":2,"Name":"Lifejacket","Price":48.95,"CategoryId":1,"Category":null,"SupplierId":1,"Supplier":null},
{"ProductId":3,"Name":"Soccer Ball","Price":19.50,"CategoryId":2,"Category":null,"SupplierId":2,"Supplier":null},
{"ProductId":4,"Name":"Corner Flags","Price":34.95,"CategoryId":2,"Category":null,"SupplierId":2,"Supplier":null},
{"ProductId":5,"Name":"Stadium","Price":79500.00,"CategoryId":2,"Category":null,"SupplierId":2,"Supplier":null},
{"ProductId":6,"Name":"Thinking Cap","Price":16.00,"CategoryId":3,"Category":null,"SupplierId":3,"Supplier":null},
{"ProductId":7,"Name":"Unsteady Chair","Price":29.95,"CategoryId":3,"Category":null,"SupplierId":3,"Supplier":null},
{"ProductId":8,"Name":"Human Chess Board","Price":75.00,"CategoryId":3,"Category":null,"SupplierId":3,"Supplier":null},
{"ProductId":9,"Name":"Bling-Bling King","Price":1200.00,"CategoryId":3,"Category":null,"SupplierId":3,"Supplier":null},
{"ProductId":10,"Name":"Swimming Goggles","Price":12.75,"CategoryId":1,"Category":null,"SupplierId":1,"Supplier":null}]
```

Figure 19-3. Storing new data using the web service

Creating a Web Service Using a Controller

The drawback of using endpoints to create a web service is that each endpoint has to duplicate a similar set of steps to produce a response: get the Entity Framework Core service so that it can query the database, set the Content-Type header for the response, serialize the objects into JSON, and so on. As a result, web services created with endpoints are difficult to understand and awkward to maintain.

A more elegant approach is to use a *controller*, which allows a web service to be defined in a single class. Controllers are part of the MVC Framework, which builds on the ASP.NET Core platform and takes care of handling data in the same way that endpoints take care of processing URLs.

THE RISE AND FALL OF THE MVC PATTERN IN ASP.NET CORE

The MVC Framework is an implementation of the Model-View-Controller pattern, which describes one way to structure an application. The examples in this chapter use two of the three pillars of the pattern: a data model (the *M* in MVC) and controllers (the *C* in MVC). Chapter 21 provides the missing piece and explains how views can be used to create HTML responses using Razor.

The MVC pattern was an important step in the evolution of ASP.NET and allowed the platform to break away from the Web Forms model that predated it. Web Forms applications were easy to start but quickly became difficult to manage and hid details of HTTP requests and responses from the developer. By contrast, the adherence to the MVC pattern provided a strong and scalable structure for applications written with the MVC Framework and hid nothing from the developer. The MVC Framework revitalized ASP.NET and provided the foundation for what became ASP.NET Core, which dropped support for Web Forms and focused solely on using the MVC pattern.

As ASP.NET Core evolved, other styles of web application have been embraced, and the MVC Framework is only one of the ways that applications can be created. That doesn't undermine the utility of the MVC pattern, but it doesn't have the central role that it used to in ASP.NET Core development, and the features that used to be unique to the MVC Framework can now be accessed through other approaches, such as Razor Pages and Blazor.

A consequence of this evolution is that understanding the MVC pattern is no longer a prerequisite for effective ASP.NET Core development. If you are interested in understanding the MVC pattern, then <https://en.wikipedia.org/wiki/Model-view-controller> is a good place to start. But for this book, understanding how the features provided by the MVC Framework build on the ASP.NET Core platform is all the context that is required.

Enabling the MVC Framework

The first step to creating a web service using a controller is to configure the MVC framework, which requires a service and an endpoint, as shown in Listing 19-6.

Listing 19-6. Enabling the MVC Framework in the Startup.cs File in the WebApp Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using WebApp.Models;

namespace WebApp {
    public class Startup {

        public Startup(IConfiguration config) {
            Configuration = config;
        }

        public IConfiguration Configuration { get; set; }

        public void ConfigureServices(IServiceCollection services) {
            services.AddDbContext<DataContext>(opts => {
                opts.UseSqlServer(Configuration[
                    "ConnectionStrings:ProductConnection"]);
                opts.EnableSensitiveDataLogging(true);
            });

            services.AddControllers();
        }

        public void Configure(IApplicationBuilder app, DataContext context) {
            app.UseDeveloperExceptionPage();
            app.UseRouting();
            app.UseMiddleware<TestMiddleware>();
            app.UseEndpoints(endpoints => {
                endpoints.MapGet("/", async context => {
                    await context.Response.WriteAsync("Hello World!");
                });
                //endpoints.MapWebService();
                endpoints.MapControllers();
            });
            SeedData.SeedDatabase(context);
        }
    }
}
```

The `AddControllers` method defines the services that are required by the MVC framework, and the `MapControllers` method defines routes that will allow controllers to handle requests. You will see other methods used to configure the MVC framework used in later chapters, which provide access to different features, but the methods used in Listing 19-6 are the ones that configure the MVC framework for web services.

Creating a Controller

Controllers are classes whose methods, known as *actions*, can process HTTP requests. Controllers are discovered automatically when the application is started. The basic discovery process is simple: any public class whose name ends with `Controller` is a controller, and any public method a controller defines is an action. To demonstrate how simple a controller can be, create the `WebApp/Controllers` folder and add to it a file named `ProductsController.cs` with the code shown in Listing 19-7.

■ **Tip** Controllers are conventionally defined in the `Controllers` folder, but they can be defined anywhere in the project, and they will still be discovered.

Listing 19-7. The Contents of the `ProductsController.cs` File in the `Controllers` Folder

```
using Microsoft.AspNetCore.Mvc;
using System.Collections.Generic;
using WebApp.Models;

namespace WebApp.Controllers {

    [Route("api/[controller]")]
    public class ProductsController: ControllerBase {

        [HttpGet]
        public IEnumerable<Product> GetProducts() {
            return new Product[] {
                new Product() { Name = "Product #1" },
                new Product() { Name = "Product #2" },
            };
        }

        [HttpGet("{id}")]
        public Product GetProduct() {
            return new Product() {
                ProductId = 1, Name = "Test Product"
            };
        }
    }
}
```

The `ProductsController` class meets the criteria that the MVC framework looks for in a controller. It defines public methods named `GetProducts` and `GetProduct`, which will be treated as actions.

Understanding the Base Class

Controllers are derived from the `ControllerBase` class, which provides access to features provided by the MVC Framework and the underlying ASP.NET Core platform. Table 19-4 describes the most useful properties provided by the `ControllerBase` class.

■ **Note** Although controllers are typically derived from the `ControllerBase` or `Controller` classes (described in Chapter 21), this is just convention, and the MVC Framework will accept any class whose name ends with `Controller`, that is derived from a class whose name ends with `Controller`, or that has been decorated with the `Controller` attribute. Apply the `NonController` attribute to classes that meet these criteria but that should not receive HTTP requests.

Table 19-4. Useful ControllerBase Properties

Name	Description
HttpContext	This property returns the HttpContext object for the current request.
ModelState	This property returns details of the data validation process, as demonstrated in the “Validating Data” section later in the chapter and described in detail in Chapter 29.
Request	This property returns the HttpRequest object for the current request.
Response	This property returns the HttpResponse object for the current response.
RouteData	This property returns the data extracted from the request URL by the routing middleware, as described in Chapter 13.
User	This property returns an object that describes the user associated with the current request, as described in Chapter 38.

A new instance of the controller class is created each time one of its actions is used to handle a request, which means the properties in Table 19-4 describe only the current request.

Understanding the Controller Attributes

The HTTP methods and URLs supported by the action methods are determined by the combination of attributes that are applied to the controller. The URL for the controller is specified by the Route attribute, which is applied to the class, like this:

```
...
[Route("api/[controller]")]
public class ProductsController: ControllerBase {
...

```

The [controller] part of the attribute argument is used to derive the URL from the name of the controller class. The Controller part of the class name is dropped, which means that the attribute in Listing 19-7 sets the URL for the controller to /api/products.

Each action is decorated with an attribute that specifies the HTTP method that it supports, like this:

```
...
[HttpGet]
public Product[] GetProducts() {
...

```

The name given to action methods doesn’t matter in controllers used for web services. There are other uses for controllers, described in Chapter 21, where the name does matter, but for web services, it is the HTTP method attributes and the route patterns that are important.

The HttpGet attribute tells the MVC framework that the GetProducts action method will handle HTTP GET requests. Table 19-5 describes the full set of attributes that can be applied to actions to specify HTTP methods.

Table 19-5. The HTTP Method Attributes

Name	Description
HttpGet	This attribute specifies that the action can be invoked only by HTTP requests that use the GET verb.
HttpPost	This attribute specifies that the action can be invoked only by HTTP requests that use the POST verb.
HttpDelete	This attribute specifies that the action can be invoked only by HTTP requests that use the DELETE verb.
HttpPut	This attribute specifies that the action can be invoked only by HTTP requests that use the PUT verb.
HttpPatch	This attribute specifies that the action can be invoked only by HTTP requests that use the PATCH verb.
HttpHead	This attribute specifies that the action can be invoked only by HTTP requests that use the HEAD verb.
AcceptVerbs	This attribute is used to specify multiple HTTP verbs.

The attributes applied to actions to specify HTTP methods can also be used to build on the controller's base URL.

```
...
[HttpGet("{id}")]
public Product GetProduct() {
...

```

This attribute tells the MVC framework that the `GetProduct` action method handles GET requests for the URL pattern `api/products/{id}`. During the discovery process, the attributes applied to the controller are used to build the set of URL patterns that the controller can handle, summarized in Table 19-6.

■ **Tip** When writing a controller, it is important to ensure that each combination of the HTTP method and URL pattern that the controller supports is mapped to only one action method. An exception will be thrown when a request can be handled by multiple actions because the MVC Framework is unable to decide which to use.

Table 19-6. *The URL Patterns*

HTTP Method	URL Pattern	Action Method Name
GET	<code>api/products</code>	<code>GetProducts</code>
GET	<code>api/products/{id}</code>	<code>GetProduct</code>

You can see how the combination of attributes is equivalent to the `MapGet` methods I used for the same URL patterns when I used endpoints to create a web service earlier in the chapter.

GET AND POST: PICK THE RIGHT ONE

The rule of thumb is that GET requests should be used for all read-only information retrieval, while POST requests should be used for any operation that changes the application state. In standards-compliance terms, GET requests are for *safe* interactions (having no side effects besides information retrieval), and POST requests are for *unsafe* interactions (making a decision or changing something). These conventions are set by the World Wide Web Consortium (W3C), at www.w3.org/Protocols/rfc2616/rfc2616-sec9.html.

GET requests are *addressable*: all the information is contained in the URL, so it's possible to bookmark and link to these addresses. Do not use GET requests for operations that change state. Many web developers learned this the hard way in 2005 when Google Web Accelerator was released to the public. This application prefetched all the content linked from each page, which is legal within the HTTP because GET requests should be safe. Unfortunately, many web developers had ignored the HTTP conventions and placed simple links to “delete item” or “add to shopping cart” in their applications. Chaos ensued.

Understanding Action Method Results

One of the main benefits provided by controllers is that the MVC Framework takes care of setting the response headers and serializing the data objects that are sent to the client. You can see this in the results defined by the action methods, like this:

```
...
[HttpGet("{id}")]
public Product GetProduct() {
...

```

When I used an endpoint, I had to work directly with the JSON serializer to create a string that can be written to the response and set the `Content-Type` header to tell the client that the response contained JSON data. The action method returns a `Product` object, which is processed automatically.

To see how the results from the action methods are handled, restart ASP.NET Core and request `http://localhost:5000/api/products`, which will produce the response shown on the left of Figure 19-4, which is produced by the `GetProducts` action method. Next, request `http://localhost:5000/api/products/1`, which will be handled by the `GetProduct` method and produce the result shown on the right side of Figure 19-4.

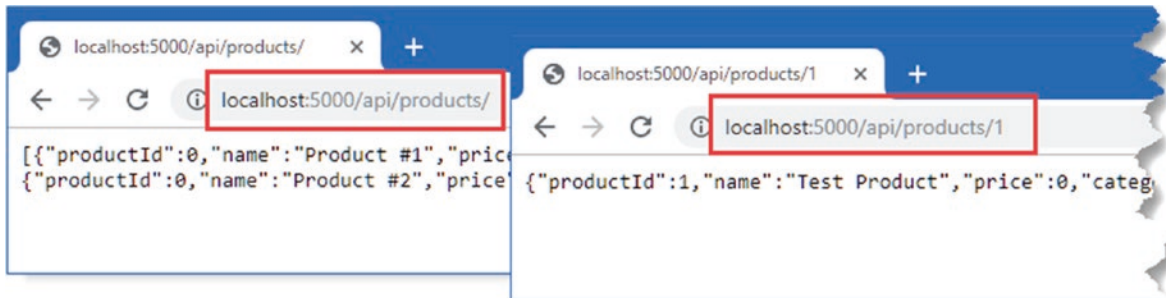


Figure 19-4. Using a controller

Using Dependency Injection in Controllers

A new instance of the controller class is created each time one of its actions is used to handle a request. The application's services are used to resolve any dependencies the controller declares through its constructor and any dependencies that the action method defines. This allows services that are required by all actions to be handled through the constructor while still allowing individual actions to declare their own dependencies, as shown in Listing 19-8.

Listing 19-8. Using Services in the `ProductsController.cs` File in the `Controllers` Folder

```
using Microsoft.AspNetCore.Mvc;
using WebApp.Models;
using System.Collections.Generic;
using Microsoft.Extensions.Logging;
using System.Linq;

namespace WebApp.Controllers {

    [Route("api/[controller]")]
    public class ProductsController: ControllerBase {
        private DataContext context;

        public ProductsController(DataContext ctx) {
            context = ctx;
        }

        [HttpGet]
        public IEnumerable<Product> GetProducts() {
            return context.Products;
        }

        [HttpGet("{id}")]
        public Product GetProduct([FromServices]
            ILogger<ProductsController> logger) {
            logger.LogDebug("GetProduct Action Invoked");
            return context.Products.FirstOrDefault();
        }
    }
}
```

The constructor declares a dependency on the `DataContext` service, which provides access to the application's data. The services are resolved using the request scope, which means that a controller can request all services, without needing to understand their lifecycle.

THE ENTITY FRAMEWORK CORE CONTEXT SERVICE LIFECYCLE

A new Entity Framework Core context object is created for each controller. Some developers will try to reuse context objects as a perceived performance improvement, but this causes problems because data from one query can affect subsequent queries, as described in Chapter 20. Behind the scenes, Entity Framework Core efficiently manages the connections to the database, and you should not try to store or reuse context objects outside of the controller for which they are created.

The `GetProducts` action method uses the `DataContext` to request all the `Product` objects in the database. The `GetProduct` method also uses the `DataContext` service, but it declares a dependency on `ILogger<T>`, which is the logging service described in Chapter 15. Dependencies that are declared by action methods must be decorated with the `FromServices` attribute, like this:

```
...
public Product GetProduct([FromServices] ILogger<ProductsController> logger) {
...

```

By default, the MVC Framework attempts to find values for action method parameters from the request URL, and the `FromServices` attribute overrides this behavior. To see the use of the services in the controller, restart ASP.NET Core and request `http://localhost:5000/api/products/1`, which will produce the response shown in Figure 19-5. You will also see the following logging message in the application's output:

```
...
dbug: WebApp.Controllers.ProductsController[0]
      GetProduct Action Invoked
...

```

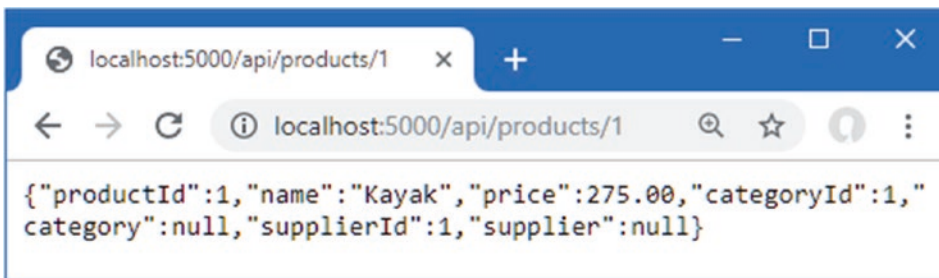


Figure 19-5. Using services in a controller

■ **Caution** One consequence of the controller lifecycle is that you can't rely on side effects caused by methods being called in a specific sequence. So, for example, I can't assign the `ILogger<T>` object received by the `GetProduct` method in Listing 19-8 to a property that can be read by the `GetProducts` action in later requests. Each controller object is used to handle one request, and only one action method will be invoked by the MVC Framework for each object.

Using Model Binding to Access Route Data

In the previous section, I noted that the MVC Framework uses the request URL to find values for action method parameters, a process known as *model binding*. Model binding is described in detail in Chapter 28, but Listing 19-9 shows a simple example.

Listing 19-9. Using Model Binding in the ProductsController.cs File in the Controllers Folder

```

using Microsoft.AspNetCore.Mvc;
using WebApp.Models;
using System.Collections.Generic;
using Microsoft.Extensions.Logging;
using System.Linq;

namespace WebApp.Controllers {

    [Route("api/[controller]")]
    public class ProductsController: ControllerBase {
        private DataContext context;

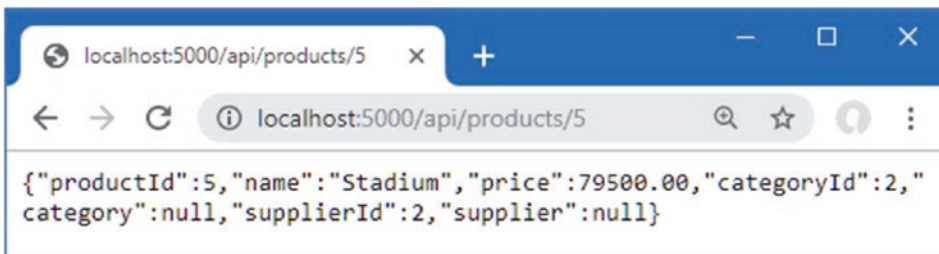
        public ProductsController(DataContext ctx) {
            context = ctx;
        }

        [HttpGet]
        public IEnumerable<Product> GetProducts() {
            return context.Products;
        }

        [HttpGet("{id}")]
        public Product GetProduct(long id,
            [FromServices] ILogger<ProductsController> logger) {
            logger.LogDebug("GetProduct Action Invoked");
            return context.Products.Find(id);
        }
    }
}

```

The listing adds a long parameter named `id` to the `GetProduct` method. When the action method is invoked, the MVC Framework injects the value with the same name from the routing data, automatically converting it to a long value, which is used by the action to query the database using the LINQ `Find` method. The result is that the action method responds to the URL, which you can see by restarting ASP.NET Core and requesting `http://localhost:5000/api/products/5`, which will produce the response shown in Figure 19-6.

**Figure 19-6.** Using model binding in an action

Model Binding from the Request Body

The model binding feature can also be used on the data in the request body, which allows clients to send data that is easily received by an action method. Listing 19-10 adds a new action method that responds to POST requests and allows clients to provide a JSON representation of the `Product` object in the request body.

Listing 19-10. Adding an Action in the ProductsController.cs File in the Controllers Folder

```

using Microsoft.AspNetCore.Mvc;
using WebApp.Models;
using System.Collections.Generic;
using Microsoft.Extensions.Logging;
using System.Linq;

namespace WebApp.Controllers {

    [Route("api/[controller]")]
    public class ProductsController: ControllerBase {
        private DataContext context;

        public ProductsController(DataContext ctx) {
            context = ctx;
        }

        [HttpGet]
        public IEnumerable<Product> GetProducts() {
            return context.Products;
        }

        [HttpGet("{id}")]
        public Product GetProduct(long id,
            [FromServices] ILogger<ProductsController> logger) {
            logger.LogDebug("GetProduct Action Invoked");
            return context.Products.Find(id);
        }

        [HttpPost]
        public void SaveProduct([FromBody]Product product) {
            context.Products.Add(product);
            context.SaveChanges();
        }
    }
}

```

The new action relies on two attributes. The `HttpPost` attribute is applied to the action method and tells the MVC Framework that the action can process POST requests. The `FromBody` attribute is applied to the action's parameter, and it specifies that the value for this parameter should be obtained by parsing the request body. When the action method is invoked, the MVC Framework will create a new `Product` object and populate its properties with the values in the request body. The model binding process can be complex and is usually combined with data validation, as described in Chapter 29, but for a simple demonstration, restart ASP.NET Core, open a new PowerShell command prompt, and run the command shown in Listing 19-11.

Listing 19-11. Sending a POST Request to the Example Application

```
Invoke-RestMethod http://localhost:5000/api/products -Method POST -Body (@{ Name="Soccer Boots"; Price=89.99;
CategoryId=2; SupplierId=2} | ConvertTo-Json) -ContentType "application/json"
```

Once the command has executed, use a web browser to request `http://localhost:5000/api/products`, and you will see the new object that has been stored in the database, as shown in Figure 19-7.

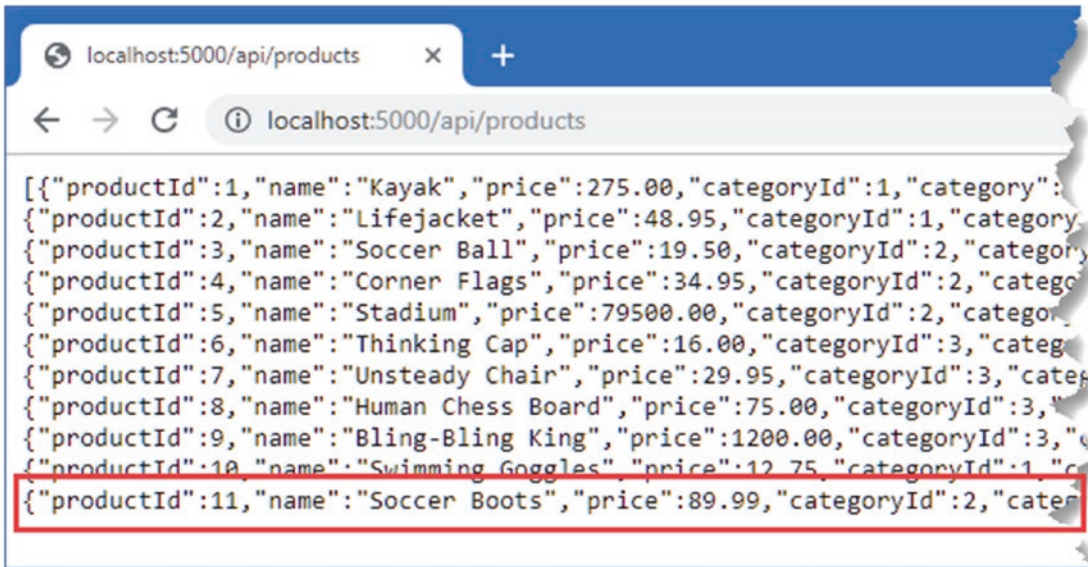


Figure 19-7. Storing new data using a controller

Adding Additional Actions

Now that the basic features are in place, I can add actions that allow clients to replace and delete Product objects using the HTTP PUT and DELETE methods, as shown in Listing 19-12.

Listing 19-12. Adding Actions in the ProductsController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using WebApp.Models;
using System.Collections.Generic;
using Microsoft.Extensions.Logging;
using System.Linq;

namespace WebApp.Controllers {

    [Route("api/[controller]")]
    public class ProductsController: ControllerBase {
        private DataContext context;

        public ProductsController(DataContext ctx) {
            context = ctx;
        }

        [HttpGet]
        public IEnumerable<Product> GetProducts() {
            return context.Products;
        }

        [HttpGet("{id}")]
        public Product GetProduct(long id,
            [FromServices] ILogger<ProductsController> logger) {
            logger.LogDebug("GetProduct Action Invoked");
            return context.Products.Find(id);
        }
    }
}
```

```

[HttpPost]
public void SaveProduct([FromBody]Product product) {
    context.Products.Add(product);
    context.SaveChanges();
}

[HttpPut]
public void UpdateProduct([FromBody]Product product) {
    context.Products.Update(product);
    context.SaveChanges();
}

[HttpDelete("{id}")]
public void DeleteProduct(long id) {
    context.Products.Remove(new Product() { ProductId = id });
    context.SaveChanges();
}
}
}

```

The `UpdateProduct` action is similar to the `SaveProduct` action and uses model binding to receive a `Product` object from the request body. The `DeleteProduct` action receives a primary key value from the URL and uses it to create a `Product` that has a value only for the `ProductId` property, which is required because Entity Framework Core works only with objects, but web service clients typically expect to be able to delete objects using just a key value.

Restart ASP.NET Core and then use a different PowerShell command prompt to run the command shown in Listing 19-13, which tests the `UpdateProduct` action.

Listing 19-13. Updating an Object

```
Invoke-RestMethod http://localhost:5000/api/products -Method PUT -Body (@{ ProductId=1; Name="Green Kayak"; Price=275; CategoryId=1; SupplierId=1} | ConvertTo-Json) -ContentType "application/json"
```

The command sends an HTTP PUT request whose body contains a replacement object. The action method receives the object through the model binding feature and updates the database. Next, run the command shown in Listing 19-14 to test the `DeleteProduct` action.

Listing 19-14. Deleting an Object

```
Invoke-RestMethod http://localhost:5000/api/products/2 -Method DELETE
```

This command sends an HTTP DELETE request, which will delete the object whose `ProductId` property is 2. To see the effect of the changes, use the browser to request `http://localhost:5000/api/products`, which will send a GET request that is handled by the `GetProducts` action and produce the response shown in Figure 19-8.

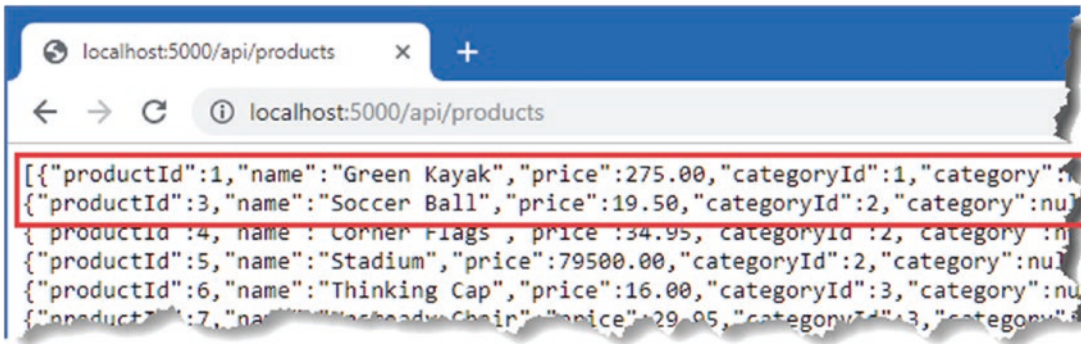


Figure 19-8. Updating and deleting objects

Improving the Web Service

The controller in Listing 19-14 re-creates all the functionality provided by the separate endpoints, but there are still improvements that can be made, as described in the following sections.

SUPPORTING CROSS-ORIGIN REQUESTS

If you are supporting third-party JavaScript clients, you may need to enable support for cross-origin requests (CORS). Browsers protect users by only allowing JavaScript code to make HTTP requests within the same origin, which means to URLs that have the same scheme, host, and port as the URL used to load the JavaScript code. CORS loosens this restriction by performing an initial HTTP request to check that the server will allow requests originating from a specific URL, helping prevent malicious code using your service without the user's consent.

ASP.NET Core provides a built-in service that handles CORS, which is enabled by adding the following statement to the `ConfigureServices` method in the `Startup` class:

```
...
Services.AddCors();
...
```

The options pattern is used to configure CORS with the `CorsOptions` class defined in the `Microsoft.AspNetCore.Cors.Infrastructure` namespace. See <https://docs.microsoft.com/en-gb/aspnet/core/security/cors?view=aspnetcore-3.1> for details.

Using Asynchronous Actions

The ASP.NET Core platform processes each request by assigning a thread from a pool. The number of requests that can be processed concurrently is limited to the size of the pool, and a thread can't be used to process any other request while it is waiting for an action to produce a result.

Actions that depend on external resources can cause a request thread to wait for an extended period. A database server, for example, may have its own concurrency limits and may queue up queries until they can be executed. The ASP.NET Core request thread is unavailable to process any other requests until the database produces a result for the action, which then produces a response that can be sent to the HTTP client.

This problem can be addressed by defining asynchronous actions, which allow ASP.NET Core threads to process other requests when they would otherwise be blocked, increasing the number of HTTP requests that the application can process simultaneously. Listing 19-15 revises the controller to use asynchronous actions.

■ **Note** Asynchronous actions don't produce responses any quicker, and the benefit is only to increase the number of requests that can be processed concurrently.

Listing 19-15. Asynchronous Actions in the ProductsController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using WebApp.Models;
using System.Collections.Generic;
using Microsoft.Extensions.Logging;
using System.Linq;
using System.Threading.Tasks;

namespace WebApp.Controllers {

    [Route("api/[controller]")]
    public class ProductsController: ControllerBase {
        private DataContext context;

        public ProductsController(DataContext ctx) {
            context = ctx;
        }

        [HttpGet]
        public IEnumerable<Product> GetProducts() {
            return context.Products;
        }

        [HttpGet("{id}")]
        public async Task<Product> GetProduct(long id) {
            return await context.Products.FindAsync(id);
        }

        [HttpPost]
        public async Task SaveProduct([FromBody]Product product) {
            await context.Products.AddAsync(product);
            await context.SaveChangesAsync();
        }

        [HttpPut]
        public async Task UpdateProduct([FromBody]Product product) {
            context.Update(product);
            await context.SaveChangesAsync();
        }

        [HttpDelete("{id}")]
        public async Task DeleteProduct(long id) {
            context.Products.Remove(new Product() { ProductId = id });
            await context.SaveChangesAsync();
        }
    }
}
```

Entity Framework Core provides asynchronous versions of some methods, such as `FindAsync`, `AddAsync`, and `SaveChangesAsync`, and I have used these with the `await` keyword. Not all operations can be performed asynchronously, which is why the `Update` and `Remove` methods are unchanged.

For some operations—including LINQ queries to the database—the `IAsyncEnumerable<T>` interface can be used, which denotes a sequence of objects that should be enumerated asynchronously and prevents the ASP.NET Core request thread from waiting for each object to be produced by the database, as explained in Chapter 5.

There is no change to the responses produced by the controller, but the threads that ASP.NET Core assigns to process each request are not necessarily blocked by the action methods.

Preventing Over-Binding

Some of the action methods use the model binding feature to get data from the response body so that it can be used to perform database operations. There is a problem with the `SaveProduct` action, which can be seen by using a PowerShell prompt to run the command shown in Listing 19-16.

Listing 19-16. Saving a Product

```
Invoke-RestMethod http://localhost:5000/api/products -Method POST -Body (@{ ProductId=100; Name="Swim Buoy";
Price=19.99; CategoryId=1; SupplierId=1} | ConvertTo-Json) -ContentType "application/json"
```

Unlike the command that was used to test the POST method in Listing 19-11, this command includes a value for the `ProductId` property. When Entity Framework Core sends the data to the database, the following exception is thrown:

```
...
Microsoft.Data.SqlClient.SqlException (0x80131904): Cannot insert explicit value for identity column in table
'Products' when IDENTITY_INSERT is set to OFF.
...
```

By default, Entity Framework Core configures the database to assign primary key values when new objects are stored. This means the application doesn't have to worry about keeping track of which key values have already been assigned and allows multiple applications to share the same database without the need to coordinate key allocation. The `Product` data model class needs a `ProductId` property, but the model binding process doesn't understand the significance of the property and adds any values that the client provides to the objects it creates, which causes the exception in the `SaveProduct` action method.

This is known as *over-binding*, and it can cause serious problems when a client provides values that the developer wasn't expecting. At best, the application will behave unexpectedly, but this technique has been used to subvert application security and grant users more access than they should have.

The safest way to prevent over-binding is to create separate data model classes that are used only for receiving data through the model binding process. Add a class file named `ProductBindingTarget.cs` to the `WebApp/Models` folder and use it to define the class shown in Listing 19-17.

Listing 19-17. The Contents of the `ProductBindingTarget.cs` File in the `WebApp/Models` Folder

```
namespace WebApp.Models {
    public class ProductBindingTarget {

        public string Name { get; set; }

        public decimal Price { get; set; }

        public long CategoryId { get; set; }

        public long SupplierId { get; set; }

        public Product ToProduct() => new Product() {
            Name = this.Name, Price = this.Price,
            CategoryId = this.CategoryId, SupplierId = this.SupplierId
        };
    }
}
```

The `ProductBindingTarget` class defines only the properties that the application wants to receive from the client when storing a new object. The `ToProduct` method creates a `Product` that can be used with the rest of the application, ensuring that the client can only provide properties for the `Name`, `Price`, `CategoryId`, and `SupplierId` properties. Listing 19-18 uses the binding target class in the `SaveProduct` action to prevent over-binding.

Listing 19-18. Using a Binding Target in the `ProductsController.cs` File in the `Controllers` Folder

```
...
[HttpPost]
public async Task SaveProduct([FromBody]ProductBindingTarget target) {
    await context.Products.AddAsync(target.ToProduct());
    await context.SaveChangesAsync();
}
...
```

Restart ASP.NET Core and repeat the command from Listing 19-16, and you will see the response shown in Figure 19-9. The client has included the `ProductId` value, but it is ignored by the model binding process, which discards values for read-only properties. (You may see a different value for the `ProductId` property when you run this example depending on the changes you made to the database before running the command.)

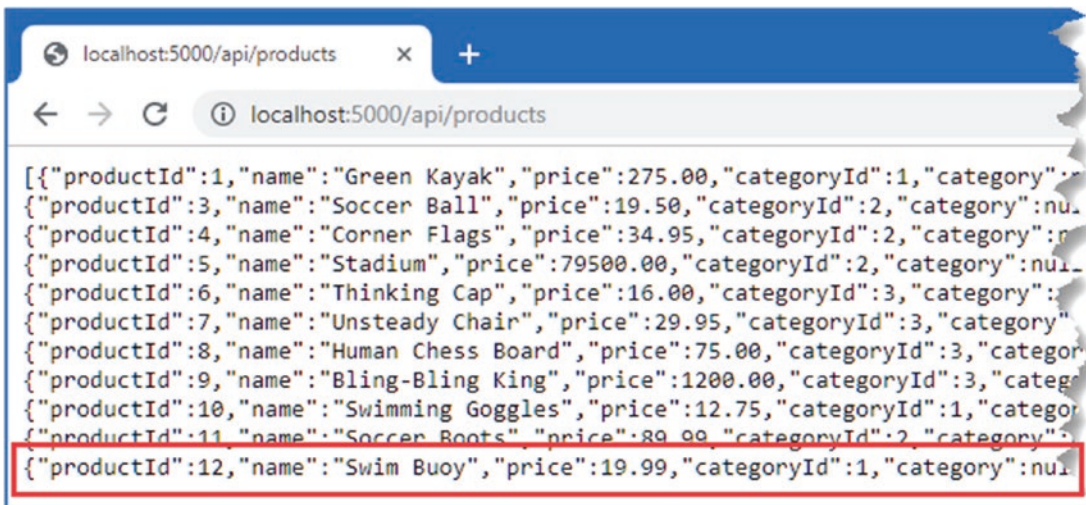


Figure 19-9. Discarding unwanted data values

Using Action Results

The MVC Framework sets the status code for responses automatically, but you won't always get the result you desire, in part because there are no firm rules for RESTful web services, and the assumptions that Microsoft makes may not match your expectations. To see an example, use a PowerShell command prompt to run the command shown in Listing 19-19, which sends a GET request to the web service.

Listing 19-19. Sending a GET Request

```
Invoke-WebRequest http://localhost:5000/api/products/1000 |Select-Object StatusCode
```

The `Invoke-WebRequest` command is similar to the `Invoke-RestMethod` command used in earlier examples but makes it easier to get the status code from the response. The URL requested in Listing 19-19 will be handled by the `GetProduct` action method, which will query the database for an object whose `ProductId` value is 1000, and the command produces the following output:

```

StatusCode
-----
        204

```

There is no matching object in the database, which means that the `GetProduct` action method returns `null`. When the MVC Framework receives `null` from an action method, it returns the 204 status code, which indicates a successful request that has produced no data. Not all web services behave this way, and a common alternative is to return a 404 response, indicating not found.

Similarly, the `SaveProducts` action will return a 200 response when it stores an object, but since the primary key isn't generated until the data is stored, the client doesn't know what key value was assigned.

■ **Note** There is no right or wrong when it comes to these kinds of web service implementation details, and you should pick the approaches that best suit your project and personal preferences. This section is an example of how to change the default behavior and not a direction to follow any specific style of web service.

Action methods can direct the MVC Framework to send a specific response by returning an object that implements the `IActionResult` interface, which is known as an *action result*. This allows the action method to specify the type of response that is required without having to produce it directly using the `HttpResponse` object.

The `ControllerBase` class provides a set of methods that are used to create action result objects, which can be returned from action methods. Table 19-7 describes the most useful action result methods.

Table 19-7. *Useful ControllerBase Action Result Methods*

Name	Description
<code>Ok</code>	The <code>IActionResult</code> returned by this method produces a 200 OK status code and sends an optional data object in the response body.
<code>NoContent</code>	The <code>IActionResult</code> returned by this method produces a 204 NO CONTENT status code.
<code>BadRequest</code>	The <code>IActionResult</code> returned by this method produces a 400 BAD REQUEST status code. The method accepts an optional model state object that describes the problem to the client, as demonstrated in the “Validating Data” section.
<code>File</code>	The <code>IActionResult</code> returned by this method produces a 200 OK response, sets the <code>Content-Type</code> header to the specified type, and sends the specified file to the client.
<code>NotFound</code>	The <code>IActionResult</code> returned by this method produces a 404 NOT FOUND status code.
<code>RedirectRedirectPermanent</code>	The <code>IActionResult</code> returned by these methods redirects the client to a specified URL.
<code>RedirectToRoute</code> <code>RedirectToRoutePermanent</code>	The <code>IActionResult</code> returned by these methods redirects the client to the specified URL that is created using the routing system, using convention routing, as described in the “Redirecting Using Route Values” sidebar.
<code>LocalRedirectLocal</code> <code>RedirectPermanent</code>	The <code>IActionResult</code> returned by these methods redirects the client to the specified URL that is local to the application.
<code>RedirectToActionRedirect</code> <code>ToActionPermanent</code>	The <code>IActionResult</code> returned by these methods redirects the client to an action method. The URL for the redirection is created using the URL routing system.
<code>RedirectToPageRedirect</code> <code>ToPagePermanent</code>	The <code>IActionResult</code> returned by these methods redirects the client to a Razor Page, described in Chapter 23.
<code>StatusCode</code>	The <code>IActionResult</code> returned by this method produces a response with a specific status code.

When an action method returns an object, it is equivalent to passing the object to the `Ok` method and returning the result. When an action returns `null`, it is equivalent to returning the result from the `NoContent` method. Listing 19-20 revises the behavior of the `GetProduct` and `SaveProduct` actions so they use the methods from Table 19-7 to override the default behavior for web service controllers.

Listing 19-20. Using Action Results in the `ProductsController.cs` File in the `Controllers` Folder

```
using Microsoft.AspNetCore.Mvc;
using WebApp.Models;
using System.Collections.Generic;
using Microsoft.Extensions.Logging;
using System.Linq;
using System.Threading.Tasks;

namespace WebApp.Controllers {

    [Route("api/[controller]")]
    public class ProductsController : ControllerBase {
        private DataContext context;

        public ProductsController(DataContext ctx) {
            context = ctx;
        }

        [HttpGet]
        public IEnumerable<Product> GetProducts() {
            return context.Products;
        }

        [HttpGet("{id}")]
        public async Task<IActionResult> GetProduct(long id) {
            Product p = await context.Products.FindAsync(id);
            if (p == null) {
                return NotFound();
            }
            return Ok(p);
        }

        [HttpPost]
        public async Task<IActionResult>
            SaveProduct([FromBody]ProductBindingTarget target) {
            Product p = target.ToProduct();
            await context.Products.AddAsync(p);
            await context.SaveChangesAsync();
            return Ok(p);
        }

        [HttpPut]
        public async Task UpdateProduct([FromBody]Product product) {
            context.Update(product);
            await context.SaveChangesAsync();
        }
    }
}
```

```

[HttpDelete("{id}")]
public async Task DeleteProduct(long id) {
    context.Products.Remove(new Product() { ProductId = id });
    await context.SaveChangesAsync();
}
}
}

```

Restart ASP.NET Core and repeat the command from Listing 19-19, and you will see an exception, which is how the Invoke-WebRequest command responds to error status codes, such as the 404 Not Found returned by the GetProduct action method.

To see the effect of the change to the SaveProduct action method, use a PowerShell command prompt to run the command shown in Listing 19-21, which sends a POST request to the web service.

Listing 19-21. Sending a POST Request

```
Invoke-RestMethod http://localhost:5000/api/products -Method POST -Body (@{Name="Boot Laces"; Price=19.99;
CategoryId=2; SupplierId=2} | ConvertTo-Json) -ContentType "application/json"
```

The command will produce the following output, showing the values that were parsed from the JSON data received from the web service:

```
productid : 13
name      : Boot Laces
price     : 19.99
categoryid : 2
category  :
supplierid : 2
supplier  :
```

Performing Redirections

Many of the action result methods in Table 19-7 relate to redirections, which direct the client to another URL. The most basic way to perform a direction is to call the Redirect method, as shown in Listing 19-22.

■ **Tip** The LocalRedirect and LocalRedirectPermanent methods throw an exception if a controller tries to perform a redirection to any URL that is not local. This is useful when you are redirecting to URLs provided by users, where an *open redirection attack* is attempted to redirect another user to an untrusted site.

Listing 19-22. Redirecting in the ProductsController.cs File in the Controllers Folder

```

using Microsoft.AspNetCore.Mvc;
using WebApp.Models;
using System.Collections.Generic;
using Microsoft.Extensions.Logging;
using System.Linq;
using System.Threading.Tasks;

namespace WebApp.Controllers {

    [Route("api/[controller]")]
    public class ProductsController : ControllerBase {
        private DataContext context;

```

```

    public ProductsController(DataContext ctx) {
        context = ctx;
    }

    // ...other action methods omitted for brevity...

    [HttpGet("redirect")]
    public IActionResult Redirect() {
        return Redirect("/api/products/1");
    }
}

```

The redirection URL is expressed as a string argument to the Redirect method, which produces a temporary redirection. Restart ASP.NET Core and use a PowerShell command prompt to run the command shown in Listing 19-23, which sends a GET request that will be handled by the new action method.

Listing 19-23. Testing Redirection

```
Invoke-RestMethod http://localhost:5000/api/products/redirect
```

The Invoke-RestMethod command will receive the redirection response from the web service and send a new request to the URL it is given, producing the following response:

```

productId : 1
name      : GreenKayak
price     : 275.00
categoryId : 1
category  :
supplierId : 1
supplier  :

```

Redirecting to an Action Method

You can redirect to another action method using the RedirectToAction method (for temporary redirections) or the RedirectToActionPermanent method (for permanent redirections). Listing 19-24 changes the Redirect action method so that the client will be redirected to another action method defined by the controller.

Listing 19-24. Redirecting to an Action the ProductsController.cs File in the Controllers Folder

```

using Microsoft.AspNetCore.Mvc;
using WebApp.Models;
using System.Collections.Generic;
using Microsoft.Extensions.Logging;
using System.Linq;
using System.Threading.Tasks;

namespace WebApp.Controllers {

    [Route("api/[controller]")]
    public class ProductsController : ControllerBase {
        private DataContext context;

        public ProductsController(DataContext ctx) {
            context = ctx;
        }
    }
}

```

```

// ...other action methods omitted for brevity...

[HttpGet("redirect")]
public IActionResult Redirect() {
    return RedirectToAction(nameof(GetProduct), new { Id = 1 });
}
}
}

```

The action method is specified as a string, although the `nameof` expression can be used to select an action method without the risk of a typo. Any additional values required to create the route are supplied using an anonymous object. Restart ASP.NET Core and use a PowerShell command prompt to repeat the command in Listing 19-23. The routing system will be used to create a URL that targets the specified action method, producing the following response:

```

productId : 1
name      : Kayak
price     : 100.00
categoryId : 1
category  :
supplierId : 1
supplier  :

```

If you specify only an action method name, then the redirection will target the current controller. There is an overload of the `RedirectToAction` method that accepts action and controller names.

REDIRECTING USING ROUTE VALUES

The `RedirectToRoute` and `RedirectToRoutePermanent` methods redirect the client to a URL that is created by providing the routing system with values for segment variables and allowing it to select a route to use. This can be useful for applications with complex routing configurations, and caution should be used because it is easy to create a redirection to the wrong URL. Here is an example of redirection with the `RedirectToRoute` method:

```

...
[HttpGet("redirect")]
public IActionResult Redirect() {
    return RedirectToRoute(new {
        controller = "Products", action = "GetProduct", Id = 1
    });
}
...

```

The set of values in this redirection relies on convention routing to select the controller and action method. Convention routing is typically used with controllers that produce HTML responses, as described in Chapter 21.

Validating Data

When you accept data from clients, you must assume that a lot of the data will be invalid and be prepared to filter out values that the application can't use. The data validation features provided for MVC Framework controllers are described in detail in Chapter 29, but for this chapter, I am going to focus on only one problem: ensuring that the client provides values for the properties that are required to store data in the database. The first step in model binding is to apply attributes to the properties of the data model class, as shown in Listing 19-25.

Listing 19-25. Applying Attributes in the ProductBindingTarget.cs File in the Models Folder

using System.ComponentModel.DataAnnotations;

```
namespace WebApp.Models {
    public class ProductBindingTarget {

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

        [Range(1, 1000)]
        public decimal Price { get; set; }

        [Range(1, long.MaxValue)]
        public long CategoryId { get; set; }

        [Range(1, long.MaxValue)]
        public long SupplierId { get; set; }

        public Product ToProduct() => new Product() {
            Name = this.Name, Price = this.Price,
            CategoryId = this.CategoryId, SupplierId = this.SupplierId
        };
    }
}
```

The Required attribute denotes properties for which the client must provide a value and can be applied to properties that are assigned null when there is no value in the request. The Range attribute requires a value between upper and lower limits and is used for primitive types that will default to zero when there is no value in the request.

Listing 19-26 updates the SaveProduct action to perform validation before storing the object that is created by the model binding process, ensuring that only objects that contain values for all four properties are decorated with the validation attributes.

Listing 19-26. Applying Validation in the ProductsController.cs File in the Controllers Folder

```
...
[HttpPost]
public async Task<IActionResult> SaveProduct([FromBody]ProductBindingTarget target) {
    if (ModelState.IsValid) {
        Product p = target.ToProduct();
        await context.Products.AddAsync(p);
        await context.SaveChangesAsync();
        return Ok(p);
    }
    return BadRequest(ModelState);
}
...
```

The ModelState property is inherited from the ControllerBase class, and the IsValid property returns true if the model binding process has produced data that meets the validation criteria. If the data received from the client is valid, then the action result from the Ok method is returned. If the data sent by the client fails the validation check, then the IsValid property will be false, and the action result from the BadRequest method is used instead. The BadRequest method accepts the object returned by the ModelState property, which is used to describe the validation errors to the client. (There is no standard way to describe validation errors, so the client may rely only on the 400 status code to determine that there is a problem.)

To test the validation, restart ASP.NET Core and use a new PowerShell command prompt to run the command shown in Listing 19-27.

Listing 19-27. Testing Validation

```
Invoke-WebRequest http://localhost:5000/api/products -Method POST -Body (@{Name="Boot Laces"} |
ConvertTo-Json) -ContentType "application/json"
```

The command will throw an exception that shows the web service has returned a 400 Bad Request response. Details of the validation errors are not shown because neither the `Invoke-WebRequest` command nor the `Invoke-RestMethod` command provides access to error response bodies. Although you can't see it, the body contains a JSON object that has properties for each data property that has failed validation, like this:

```
{
  "Price":["The field Price must be between 1 and 1000."],
  "CategoryId":["The field CategoryId must be between 1 and 9.223372036854776E+18."],
  "SupplierId":["The field SupplierId must be between 1 and 9.223372036854776E+18."]
}
```

You can see examples of working with validation messages in Chapter 29 where the validation feature is described in detail.

Applying the API Controller Attribute

The `ApiController` attribute can be applied to web service controller classes to change the behavior of the model binding and validation features. The use of the `FromBody` attribute to select data from the request body and explicitly checking the `ModelState.IsValid` property is not required in controllers that have been decorated with the `ApiController` attribute. Getting data from the body and validating data are required so commonly in web services that they are applied automatically when the attribute is used, restoring the focus of the code in the controller's action to dealing with the application features, as shown in Listing 19-28.

Listing 19-28. Using `ApiController` in the `ProductsController.cs` File in the `Controllers` Folder

```
using Microsoft.AspNetCore.Mvc;
using WebApp.Models;
using System.Collections.Generic;
using Microsoft.Extensions.Logging;
using System.Linq;
using System.Threading.Tasks;

namespace WebApp.Controllers {

    [ApiController]
    [Route("api/[controller]")]
    public class ProductsController : ControllerBase {
        private DataContext context;

        public ProductsController(DataContext ctx) {
            context = ctx;
        }

        [HttpGet]
        public IEnumerable<Product> GetProducts() {
            return context.Products;
        }

        [HttpGet("{id}")]
        public async Task<ActionResult> GetProduct(long id) {
            Product p = await context.Products.FindAsync(id);
            if (p == null) {
```

```

        return NotFound();
    }
    return Ok(p);
}

[HttpPost]
public async Task<IActionResult> SaveProduct(ProductBindingTarget target) {
    Product p = target.ToProduct();
    await context.Products.AddAsync(p);
    await context.SaveChangesAsync();
    return Ok(p);
}

[HttpPut]
public async Task UpdateProduct(Product product) {
    context.Update(product);
    await context.SaveChangesAsync();
}

[HttpDelete("{id}")]
public async Task DeleteProduct(long id) {
    context.Products.Remove(new Product() { ProductId = id });
    await context.SaveChangesAsync();
}

[HttpGet("redirect")]
public IActionResult Redirect() {
    return RedirectToAction(nameof(GetProduct), new { Id = 1 });
}
}
}

```

Using the `ApiController` attribute is optional, but it helps produce concise web service controllers.

Omitting Null Properties

The final change I am going to make in this chapter is to remove the null values from the data returned by the web service. The data model classes contain navigation properties that are used by Entity Framework Core to associate related data in complex queries, as explained in Chapter 20. For the simple queries that are performed in this chapter, no values are assigned to these navigation properties, which means that the client receives properties for which values are never going to be available. To see the problem, use a PowerShell command prompt to run the command shown in Listing 19-29.

Listing 19-29. Sending a GET Request

```
Invoke-WebRequest http://localhost:5000/api/products/1 | Select-Object Content
```

The command sends a GET request and displays the body of the response from the web service, producing the following output:

```
Content
```

```
-----
{"productId":1,"name":"Green Kayak","price":275.00,"categoryId":1,"category":null,"supplierId":1,"supplier":null}
```

The request was handled by the `GetProduct` action method, and the category and supplier values in the response will always be null because the action doesn't ask Entity Framework Core to populate these properties.

Projecting Selected Properties

The first approach is to return just the properties that the client requires. This gives you complete control over each response, but it can become difficult to manage and confusing for client developers if each action returns a different set of values. Listing 19-30 shows how the `Product` object obtained from the database can be projected so that the navigation properties are omitted.

Listing 19-30. Omitting Properties in the `ProductsController.cs` File in the `Controllers` Folder

```
...
[HttpGet("{id}")]
public async Task<IActionResult> GetProduct(long id) {
    Product p = await context.Products.FindAsync(id);
    if (p == null) {
        return NotFound();
    }
    return Ok(new {
        ProductId = p.ProductId, Name = p.Name,
        Price = p.Price, CategoryId = p.CategoryId,
        SupplierId = p.SupplierId
    });
}
...
```

The properties that the client requires are selected and added to an object that is passed to the `Ok` method. Restart ASP.NET Core and run the command from Listing 19-30, and you will receive a response that omits the navigation properties and their null values, like this:

Content

```
-----
{"productId":1,"name":"Green Kayak","price":275.00,"categoryId":1,"supplierId":1}
```

Configuring the JSON Serializer

The JSON serializer can be configured to omit properties whose value is null when it serializes objects. The serializer is configured using the options pattern in the `Startup` class, as shown in Listing 19-31.

Listing 19-31. Configuring the JSON Serializer in the `Startup.cs` File in the `WebApp` Folder

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Configuration;
using Microsoft.EntityFrameworkCore;
using WebApp.Models;
using Microsoft.AspNetCore.Mvc;
```

```

namespace WebApp {
    public class Startup {

        public Startup(IConfiguration config) {
            Configuration = config;
        }

        public IConfiguration Configuration { get; set; }

        public void ConfigureServices(IServiceCollection services) {
            services.AddDbContext<DataContext>(opts => {
                opts.UseSqlServer(Configuration[
                    "ConnectionStrings:ProductConnection"]);
                opts.EnableSensitiveDataLogging(true);
            });

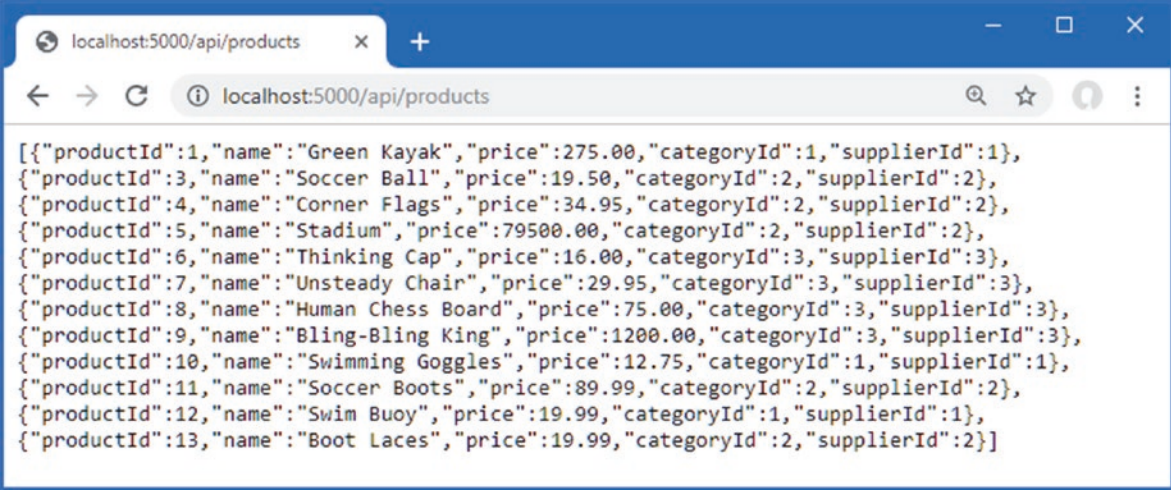
            services.AddControllers();
            services.Configure<JsonOptions>(opts => {
                opts.JsonSerializerOptions.IgnoreNullValues = true;
            });
        }

        public void Configure(IApplicationBuilder app, DataContext context) {
            app.UseDeveloperExceptionPage();
            app.UseRouting();
            app.UseMiddleware<TestMiddleware>();
            app.UseEndpoints(endpoints => {
                endpoints.MapGet("/", async context => {
                    await context.Response.WriteAsync("Hello World!");
                });
                endpoints.MapControllers();
            });
            SeedData.SeedDatabase(context);
        }
    }
}

```

The JSON serializer is configured using the `JsonSerializerOptions` property of the `JsonOptions` class, and null values are discarded when the `IgnoreNullValues` property is true.

This configuration change affects all JSON responses and should be used with caution, especially if any of your data model classes use null values to impart information to the client. To see the effect of the change, restart ASP.NET Core and use a browser to request `http://localhost:5000/api/products`, which will produce the response shown in Figure 19-10.

A screenshot of a web browser window. The address bar shows 'localhost:5000/api/products'. The main content area displays a JSON array of 13 product objects. Each object contains 'productId', 'name', 'price', 'categoryId', and 'supplierId' fields. The products listed include 'Green Kayak', 'Soccer Ball', 'Corner Flags', 'Stadium', 'Thinking Cap', 'Unsteady Chair', 'Human Chess Board', 'Bling-Bling King', 'Swimming Goggles', 'Soccer Boots', 'Swim Buoy', and 'Boot Laces'.

```
[{"productId":1,"name":"Green Kayak","price":275.00,"categoryId":1,"supplierId":1}, {"productId":3,"name":"Soccer Ball","price":19.50,"categoryId":2,"supplierId":2}, {"productId":4,"name":"Corner Flags","price":34.95,"categoryId":2,"supplierId":2}, {"productId":5,"name":"Stadium","price":79500.00,"categoryId":2,"supplierId":2}, {"productId":6,"name":"Thinking Cap","price":16.00,"categoryId":3,"supplierId":3}, {"productId":7,"name":"Unsteady Chair","price":29.95,"categoryId":3,"supplierId":3}, {"productId":8,"name":"Human Chess Board","price":75.00,"categoryId":3,"supplierId":3}, {"productId":9,"name":"Bling-Bling King","price":1200.00,"categoryId":3,"supplierId":3}, {"productId":10,"name":"Swimming Goggles","price":12.75,"categoryId":1,"supplierId":1}, {"productId":11,"name":"Soccer Boots","price":89.99,"categoryId":2,"supplierId":2}, {"productId":12,"name":"Swim Buoy","price":19.99,"categoryId":1,"supplierId":1}, {"productId":13,"name":"Boot Laces","price":19.99,"categoryId":2,"supplierId":2}]
```

Figure 19-10. Configuring the JSON serializer

Summary

In this chapter, I showed you how to use the MVC Framework to create RESTful web services. I explained that the MVC Framework is built on top of the ASP.NET Core platform and showed you how a single controller class can define multiple action methods, each of which can process a different combination of URL and HTTP method. I showed you how to use dependency injection to consume services, how to use model binding to access request data, how to validate request data, and how to take control of the responses that are returned to the client. In the next chapter, I describe the advanced features that ASP.NET Core provides for web services.

CHAPTER 20



Advanced Web Service Features

In this chapter, I describe advanced features that can be used to create RESTful web services. I explain how to deal with related data in Entity Framework Core queries, how to add support for the HTTP PATCH method, how to use content negotiations, and how to use OpenAPI to describe your web services. Table 20-1 puts this chapter in context.

Table 20-1. *Putting Advanced Web Service Features in Context*

Question	Answer
What are they?	The features described in this chapter provide greater control over how ASP.NET Core web services work, including managing the data sent to the client and the format used for that data.
Why are they useful?	The default behaviors provided by ASP.NET Core don't meet the needs of every project, and the features described in this chapter allow web services to be reshaped to fit specific requirements.
How are they used?	The common theme for the features in this chapter is altering the responses produced by action methods.
Are there any pitfalls or limitations?	It can be hard to decide how to implement web services, especially if they are consumed by third-party clients. The behavior of a web service becomes fixed as soon as clients start using a web service, which means that careful thought is required when using the features described in this chapter.
Are there any alternatives?	The features described in this chapter are optional, and you can rely on the default behaviors of ASP.NET Core web services.

Table 20-2 summarizes the chapter.

Table 20-2. *Chapter Summary*

Problem	Solution	Listing
Using relational data	Use the <code>Include</code> and <code>ThenInclude</code> methods in LINQ queries	4
Breaking circular references	Explicitly set navigation properties to null	5
Allowing clients to selectively update data	Support the HTTP PATCH method	6–9
Supporting a range of response data types	Support content formatting and negotiation	10–24
Documenting a web service	Use OpenAPI to describe the web service	25–29

Preparing for This Chapter

This chapter uses the WebApp project created in Chapter 18 and modified in Chapter 19. To prepare for this chapter, add a file named `SuppliersController.cs` to the `WebApp/Controllers` folder with the content shown in Listing 20-1.

Listing 20-1. The Contents of the SuppliersController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using WebApp.Models;
using System.Threading.Tasks;

namespace WebApp.Controllers {

    [ApiController]
    [Route("api/[controller]")]
    public class SuppliersController: ControllerBase {
        private DataContext context;

        public SuppliersController(DataContext ctx) {
            context = ctx;
        }

        [HttpGet("{id}")]
        public async Task<Supplier> GetSupplier(long id) {
            return await context.Suppliers.FindAsync(id);
        }
    }
}
```

The controller extends the ControllerBase class, declares a dependency on the DataContext service, and defines an action named GetSupplier that handles GET requests for the /api/[controller]/{id} URL pattern.

Dropping the Database

Open a new PowerShell command prompt, navigate to the folder that contains the WebApp.csproj file, and run the command shown in Listing 20-2 to drop the database.

■ **Tip** You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/apress/pro-asp.net-core-3>. See Chapter 1 for how to get help if you have problems running the examples.

Listing 20-2. Dropping the Database

```
dotnet ef database drop --force
```

Running the Example Application

Once the database has been dropped, select Start Without Debugging or Run Without Debugging from the Debug menu or use the PowerShell command prompt to run the command shown in Listing 20-3.

Listing 20-3. Running the Example Application

```
dotnet run
```

The database will be seeded as part of the application startup. Once ASP.NET Core is running, use a web browser to request <http://localhost:5000/api/suppliers/1>, which will produce the response shown in Figure 20-1.

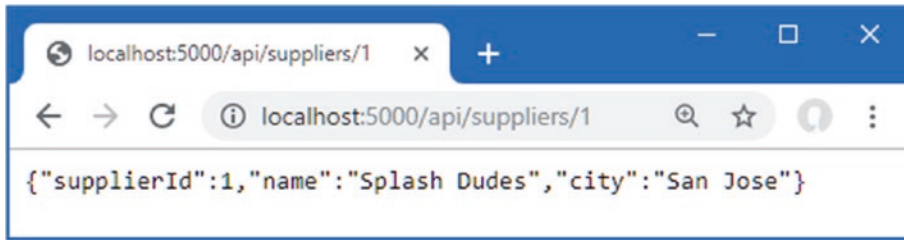


Figure 20-1. Running the example application

The response shows the `Supplier` object whose primary key matches the last segment of the request URL. In Chapter 19, the JSON serializer was configured to ignore properties with null values, which is why the response doesn't include the navigation property defined by the `Supplier` data model class.

Dealing with Related Data

Although this isn't a book about Entity Framework Core, there is one aspect of querying for data that most web services encounter. The data model classes defined in Chapter 18 include navigation properties, which Entity Framework Core can populate by following relationships in the database when the `Include` method is used, as shown in Listing 20-4.

Listing 20-4. Requesting Related Data in the `SuppliersController.cs` File in the `Controllers` Folder

```
using Microsoft.AspNetCore.Mvc;
using WebApp.Models;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;

namespace WebApp.Controllers {

    [ApiController]
    [Route("api/[controller]")]
    public class SuppliersController: ControllerBase {
        private DataContext context;

        public SuppliersController(DataContext ctx) {
            context = ctx;
        }

        [HttpGet("{id}")]
        public async Task<Supplier> GetSupplier(long id) {
            return await context.Suppliers
                .Include(s => s.Products)
                .FirstOrDefault(s => s.SupplierId == id);
        }
    }
}
```

The `Include` method tells Entity Framework Core to follow a relationship in the database and load the related data. In this case, the `Include` method selects the `Products` navigation property defined by the `Supplier` class, which causes Entity Framework Core to load the `Product` objects associated with the selected `Supplier` and assign them to the `Products` property.

Restart ASP.NET Core and use a browser to request `http://localhost:5000/api/suppliers/1`, which will target the `GetSupplier` action method. The request fails, and you will see the exception shown in Figure 20-2.

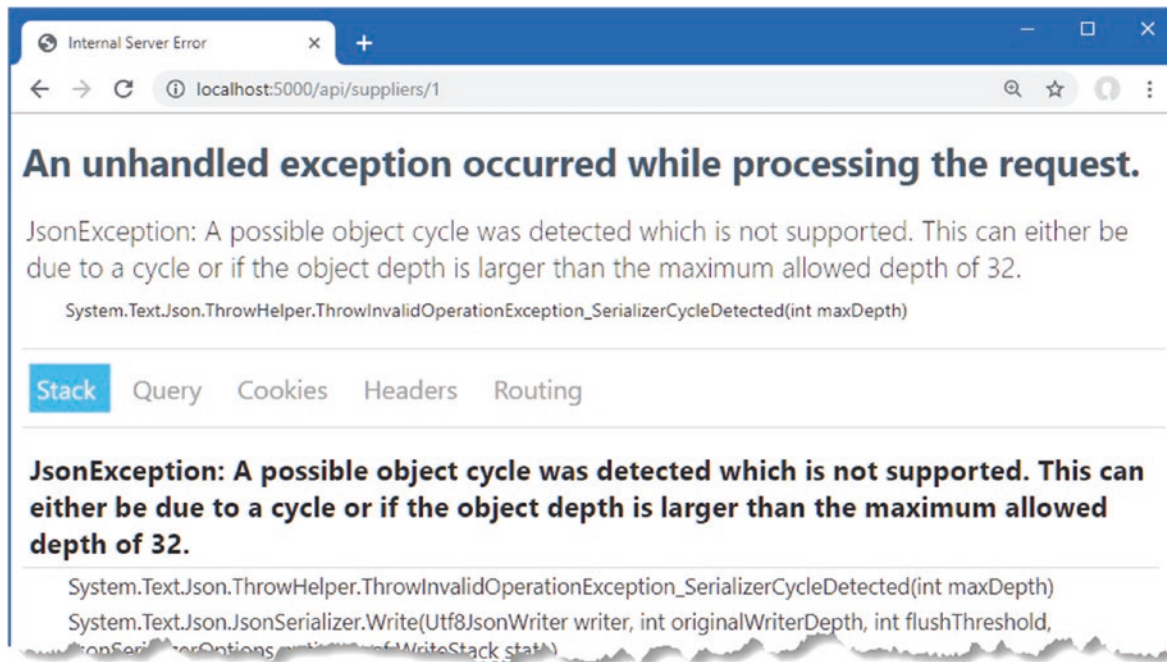


Figure 20-2. An exception caused by querying for related data

The JSON serializer has reported an “object cycle,” which means there is a circular reference in the data that is being serialized for the response.

Looking at the code in Listing 20-4, you might struggle to see why using the `Include` method has created a circular reference. The problem is caused by an Entity Framework Core feature that attempts to minimize the amount of data read from the database but that causes problems in ASP.NET Core applications.

When Entity Framework Core creates objects, it populates navigation properties with objects that have already been created by the same database context. This can be a useful feature in some kinds of applications, such as desktop apps, where a database context object has a long life and is used to make many requests over time. It isn’t useful for ASP.NET Core applications, where a new context object is created for each HTTP request.

Entity Framework Core queries the database for the `Product` objects associated with the selected `Supplier` and assigns them to the `Supplier.Products` navigation property. The problem is that Entity Framework Core then looks at each `Product` object it has created and uses the query response to populate the `Product.Supplier` navigation property as well. For an ASP.NET Core application, this is an unhelpful step to take because it creates a circular reference between the navigation properties of the `Supplier` and `Product` objects, as shown in Figure 20-3.

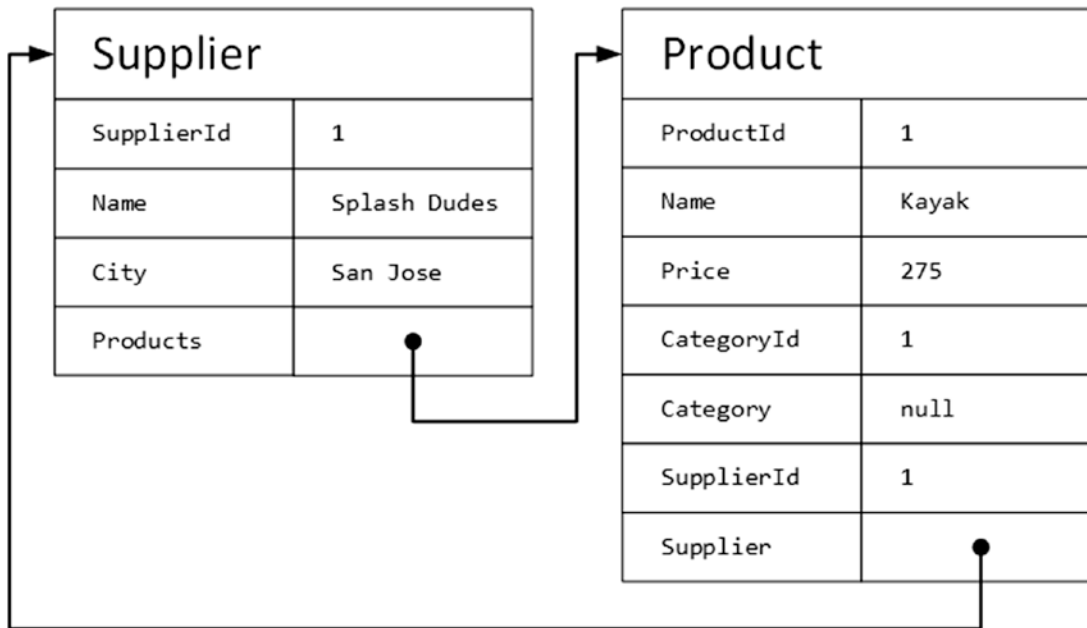


Figure 20-3. Understanding how Entity Framework Core uses related data

When the Supplier object is returned by the controller's action method, the JSON serializer works its way through the properties and follows the references to the Product objects, each of which has a reference back to the Supplier object, which it follows in a loop until the maximum depth is reached and the exception shown in Figure 20-2 is thrown.

Breaking Circular References in Related Data

There is no way to stop Entity Framework Core from creating circular references in the data it loads in the database. Preventing the exception means presenting the JSON serializer with data that doesn't contain circular references, which is most easily done by altering the objects after they have been created by Entity Framework Core and before they are serialized, as shown in Listing 20-5.

Listing 20-5. Breaking References in the SuppliersController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using WebApp.Models;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;

namespace WebApp.Controllers {

    [ApiController]
    [Route("api/[controller]")]
    public class SuppliersController: ControllerBase {
        private DataContext context;

        public SuppliersController(DataContext ctx) {
            context = ctx;
        }

        [HttpGet("{id}")]
        public async Task<Supplier> GetSupplier(long id) {
            Supplier supplier = await context.Suppliers.Include(s => s.Products)
                .FirstOrDefaultAsync(s => s.SupplierId == id);
        }
    }
}
```

```

        foreach (Product p in supplier.Products) {
            p.Supplier = null;
        };
    return supplier;
}
}
}

```

The foreach loop sets the Supplier property of each Product object to null, which breaks the circular references. Restart ASP.NET Core and request `http://localhost:5000/api/suppliers/1` to query for a supplier and its related products, which produces the response shown in Figure 20-4.

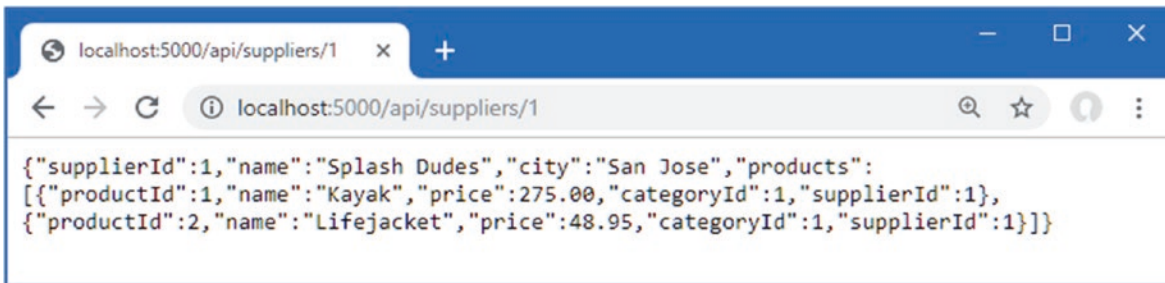


Figure 20-4. Querying for related data

Supporting the HTTP PATCH Method

For simple data types, edit operations can be handled by replacing the existing object using the PUT method, which is the approach I took in Chapter 19. Even if you only need to change a single property value in the Product class, for example, it isn't too much trouble to use a PUT method and include the values for all the other Product properties, too.

Not all data types are as easy to work with, either because they define too many properties or because the client has only received values for selected properties. The solution is to use a PATCH request, which sends just the changes to the web service rather than a complete replacement object.

Understanding JSON Patch

ASP.NET Core has support for working with the JSON Patch standard, which allows changes to be specified in a uniform way. The JSON Patch standard allows for a complex set of changes to be described, but for this chapter, I am going to focus on just the ability to change the value of a property.

I am not going to go into the details of the JSON Patch standard, which you can read at <https://tools.ietf.org/html/rfc6902>, but the client is going to send the web service JSON data like this in its HTTP PATCH requests:

```

[
  { "op": "replace", "path": "Name", "value": "Surf Co"},
  { "op": "replace", "path": "City", "value": "Los Angeles"},
]

```

A JSON Patch document is expressed as an array of operations. Each operation has an `op` property, which specifies the type of operation, and a `path` property, which specifies where the operation will be applied.

For the example application—and, in fact, for most applications—only the replace operation is required, which is used to change the value of a property. This JSON Patch document sets new values for the Name and City properties. The properties defined by the Supplier class not mentioned in the JSON Patch document will not be modified.

Installing and Configuring the JSON Patch Package

Support for JSON Patch isn't installed when a project is created with the Empty template. To install the JSON Patch package, open a new PowerShell command prompt, navigate to the folder that contains the `WebApp.csproj` file, and run the command shown in Listing 20-6. If you are using Visual Studio, you can install the package by selecting Project ► Manage NuGet Packages.

Listing 20-6. Installing the JSON Patch Package

```
dotnet add package Microsoft.AspNetCore.Mvc.NewtonsoftJson --version 3.1.1
```

The Microsoft implementation of JSON Patch relies on the third-party Newtonsoft JSON.NET serializer that was used in ASP.NET Core 2.x but that has been replaced with a bespoke JSON serializer in ASP.NET Core 3.x. Add the statement shown in Listing 20-7 to the `ConfigureServices` method of the `Startup` class to enable the old serializer.

Listing 20-7. Enabling the JSON.NET Serializer in the `Startup.cs` File in the WebApp Folder

```
...
public void ConfigureServices(IServiceCollection services) {
    services.AddDbContext<DataContext>(opts => {
        opts.UseSqlServer(Configuration[
            "ConnectionStrings:ProductConnection"]);
        opts.EnableSensitiveDataLogging(true);
    });

    services.AddControllers().AddNewtonsoftJson();

    services.Configure<MvcNewtonsoftJsonOptions>(opts => {
        opts.SerializerSettings.NullValueHandling
        = Newtonsoft.Json.NullValueHandling.Ignore;
    });

    //services.Configure<JsonOptions>(opts => {
    // opts.JsonSerializerOptions.IgnoreNullValues = true;
    //});
}
...
```

The `AddNewtonsoftJson` method enables the JSON.NET serializer, which replaces the standard ASP.NET Core serializer. The JSON.NET serializer has its own configuration class, `MvcNewtonsoftJsonOptions`, which is applied through the options pattern. Listing 20-7 sets the `NullValueHandling` value, which tells the serializer to discard properties with null values.

■ **Tip** See <https://www.newtonsoft.com/json> for details of the other configuration options available for the JSON.NET serializer.

Defining the Action Method

To add support for the PATCH method, add the action method shown in Listing 20-8 to the `SuppliersController` class.

Listing 20-8. Adding an Action in the `SuppliersController.cs` File in the Controller Folder

```
using Microsoft.AspNetCore.Mvc;
using WebApp.Models;
using System.Threading.Tasks;
using Microsoft.EntityFrameworkCore;
using Microsoft.AspNetCore.JsonPatch;
```

```

namespace WebApp.Controllers {

    [ApiController]
    [Route("api/[controller]")]
    public class SuppliersController : ControllerBase {
        private DataContext context;

        public SuppliersController(DataContext ctx) {
            context = ctx;
        }

        [HttpGet("{id}")]
        public async Task<Supplier> GetSupplier(long id) {
            Supplier supplier = await context.Suppliers.Include(s => s.Products)
                .FirstAsync(s => s.SupplierId == id);
            foreach (Product p in supplier.Products) {
                p.Supplier = null;
            };
            return supplier;
        }

        [HttpPatch("{id}")]
        public async Task<Supplier> PatchSupplier(long id,
            JsonPatchDocument<Supplier> patchDoc) {
            Supplier s = await context.Suppliers.FindAsync(id);
            if (s != null) {
                patchDoc.ApplyTo(s);
                await context.SaveChangesAsync();
            }
            return s;
        }
    }
}

```

The action method is decorated with the `HttpPatch` attribute, which denotes that it will handle HTTP requests. The model binding feature is used to process the JSON Patch document through a `JsonPatchDocument<T>` method parameter. The `JsonPatchDocument<T>` class defines an `ApplyTo` method, which applies each operation to an object. The action method in Listing 20-8 retrieves a `Supplier` object from the database, applies the JSON PATCH, and stores the modified object.

Restart ASP.NET Core and use a PowerShell command prompt to run the command shown in Listing 20-9, which sends an HTTP PATCH request with a JSON PATCH document that changes the value of the `City` property to Los Angeles.

Listing 20-9. Sending an HTTP PATCH Request

```

Invoke-RestMethod http://localhost:5000/api/suppliers/1 -Method PATCH -ContentType "application/json"
-Body '[{"op":"replace","path":"City","value":"Los Angeles"}]'

```

The `PatchSupplier` action method returns the modified `Supplier` object as its result, which is serialized and sent to the client in the HTTP response. You can also see the effect of the change by using a web browser to request `http://localhost:5000/suppliers/1`, which produces the response shown in Figure 20-5.

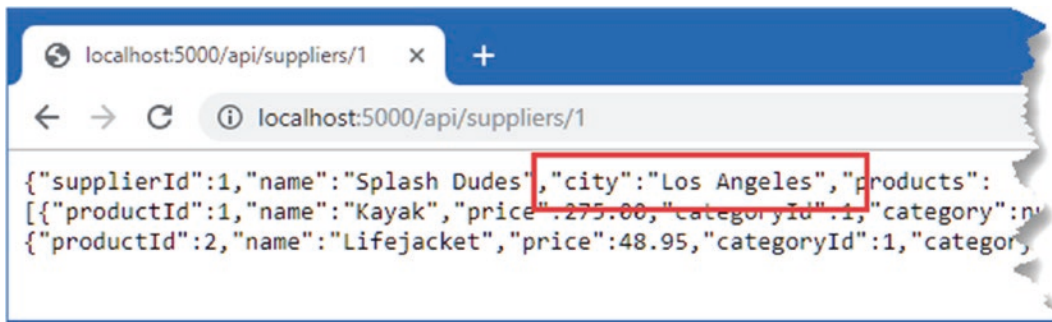


Figure 20-5. Updating using a PATCH request

Understanding Content Formatting

The web service examples so far have produced JSON results, but this is not the only data format that action methods can produce. The content format selected for an action result depends on four factors: the formats that the client will accept, the formats that the application can produce, the content policy specified by the action method, and the type returned by the action method. Figuring out how everything fits together can be daunting, but the good news is that the default policy works just fine for most applications, and you only need to understand what happens behind the scenes when you need to make a change or when you are not getting results in the format that you expect.

Understanding the Default Content Policy

The best way to get acquainted with content formatting is to understand what happens when neither the client nor the action method applies any restrictions to the formats that can be used. In this situation, the outcome is simple and predictable.

1. If the action method returns a string, the string is sent unmodified to the client, and the Content-Type header of the response is set to text/plain.
2. For all other data types, including other simple types such as int, the data is formatted as JSON, and the Content-Type header of the response is set to application/json.

Strings get special treatment because they cause problems when they are encoded as JSON. When you encode other simple types, such as the C# int value 2, then the result is a quoted string, such as "2". When you encode a string, you end up with two sets of quotes so that "Hello" becomes ""Hello"". Not all clients cope well with this double encoding, so it is more reliable to use the text/plain format and sidestep the issue entirely. This is rarely an issue because few applications send string values; it is more common to send objects in the JSON format. To see the default policy, add a class file named ContentController.cs to the WebApps/Controllers folder with the code shown in Listing 20-10.

Listing 20-10. The Contents of the ContentController.cs File in the Controllers Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Threading.Tasks;
using WebApp.Models;

namespace WebApp.Controllers {

    [ApiController]
    [Route("/api/[controller]")]
    public class ContentController : ControllerBase {
        private DataContext context;

        public ContentController(DataContext dataContext) {
            context = dataContext;
        }
    }
}
```

```

[HttpGet("string")]
public string GetString() => "This is a string response";

[HttpGet("object")]
public async Task<Product> GetObject() {
    return await context.Products.FirstAsync();
}
}
}

```

The controller defines actions that return string and object results. Restart ASP.NET Core and use a PowerShell prompt to run the command shown in Listing 20-11; this command sends a request that invokes the `GetString` action method, which returns a string.

Listing 20-11. Requesting a String Response

```
Invoke-WebRequest http://localhost:5000/api/content/string | select @{n='Content-Type';e={$_.Headers.
"Content-Type" }}, Content
```

This command sends a GET request to the `/api/content/string` URL and processes the response to display the `Content-Type` header and the content from the response. The command produces the following output, which shows the `Content-Type` header for the response:

Content-Type	Content
-----	-----
text/plain; charset=utf-8	This is a string response

Next, run the command shown in Listing 20-12, which sends a request that will be handled by the `GetObject` action method.

Listing 20-12. Requesting an Object Response

```
Invoke-WebRequest http://localhost:5000/api/content/object | select @{n='Content-Type';e={
$_.Headers."Content-Type" }}, Content
```

This command produces the following output, formatted for clarity, that shows that the response has been encoded as JSON:

Content-Type	Content
-----	-----
application/json; charset=utf-8	{"productId":1,"name":"Kayak", "price":275.00,"categoryId":1,"supplierId":1}

Understanding Content Negotiation

Most clients include an `Accept` header in a request, which specifies the set of formats that they are willing to receive in the response, expressed as a set of MIME types. Here is the `Accept` header that Google Chrome sends in requests:

```
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,
*/*;q=0.8
```

This header indicates that Chrome can handle the HTML and XHTML formats (XHTML is an XML-compliant dialect of HTML), XML, and the WEBP image format (which is the animated PNG image format).

The `q` values in the header specify relative preference, where the value is 1.0 by default. Specifying a `q` value for 0.9 for `application/xml` tells the server that Chrome will accept XML data but prefers to deal with HTML or XHTML. The `*/*` item tells the server that Chrome will accept any format, but its `q` value specifies that it is the lowest preference of the specified types. Putting this together means that the Accept header sent by Chrome provides the server with the following information:

1. Chrome prefers to receive HTML or XHTML data or WEBP and APNG images.
2. If those formats are not available, then the next most preferred format is XML.
3. If none of the preferred formats is available, then Chrome will accept any format.

You might assume from this that you can change the format produced by the ASP.NET Core application by setting the Accept header, but it doesn't work that way—or, rather, it doesn't work that way just yet because there is some preparation required.

To see what happens when the Accept header is changed, use a PowerShell prompt to run the command shown in Listing 20-13, which sets the Accept header to tell ASP.NET Core that the client is willing to receive only XML data.

Listing 20-13. Requesting XML Data

```
Invoke-WebRequest http://localhost:5000/api/content/object -Headers @{Accept="application/xml"} | select
@{n='Content-Type';e={$_.Headers."Content-Type"}}, Content
```

Here are the results, which show that the application has sent an `application/json` response:

Content-Type	Content
application/json; charset=utf-8	{"productId":1,"name":"Kayak", "price":275.00,"categoryId":1,"supplierId":1}

Including the Accept header has no effect on the format, even though the ASP.NET Core application sent the client a format that it hasn't specified. The problem is that, by default, the MVC Framework is configured to only use JSON. Rather than return an error, the MVC Framework sends JSON data in the hope that the client can process it, even though it was not one of the formats specified by the request Accept header.

Enabling XML Formatting

For content negotiation to work, the application must be configured so there is some choice in the formats that can be used. Although JSON has become the default format for web applications, the MVC Framework can also support encoding data as XML, as shown in Listing 20-14.

■ **Tip** You can create your own content format by deriving from the `Microsoft.AspNetCore.Mvc.Formatters.OutputFormatter` class. This is rarely used because creating a custom data format isn't a useful way of exposing the data in your application, and the most common formats—JSON and XML—are already implemented.

Listing 20-14. Enabling XML Formatting in the Startup.cs File in the WebApp Folder

```
...
public void ConfigureServices(IServiceCollection services) {
    services.AddDbContext<DataContext>(opts => {
        opts.UseSqlServer(Configuration[
            "ConnectionStrings:ProductConnection"]);
        opts.EnableSensitiveDataLogging(true);
    });
    services.AddControllers().AddNewtonsoftJson().AddXmlSerializerFormatters();
}
```



```

services.Configure<MvcNewtonsoftJsonOptions>(opts => {
    opts.SerializerSettings.NullValueHandling
        = Newtonsoft.Json.NullValueHandling.Ignore;
});
}
...

```

The XML Serializer has some limitations, including the inability to deal with Entity Framework Core navigation properties because they are defined through an interface. To create an object that can be serialized, Listing 20-15 uses `ProductBindingTarget` defined in Chapter 19.

Listing 20-15. Creating a Serializable Object in the `ContentController.cs` File in the `Controllers` Folder

```

using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Threading.Tasks;
using WebApp.Models;

namespace WebApp.Controllers {

    [ApiController]
    [Route("/api/[controller]")]
    public class ContentController : ControllerBase {
        private DataContext context;

        public ContentController(DataContext dataContext) {
            context = dataContext;
        }

        [HttpGet("string")]
        public string GetString() => "This is a string response";

        [HttpGet("object")]
        public async Task<ProductBindingTarget> GetObject() {
            Product p = await context.Products.FirstAsync();
            return new ProductBindingTarget() {
                Name = p.Name, Price = p.Price, CategoryId = p.CategoryId,
                SupplierId = p.SupplierId
            };
        }
    }
}

```

When the MVC Framework had only the JSON format available, it had no choice but to encode responses as JSON. Now that there is a choice, you can see the content negotiation process working more fully. Restart ASP.NET Core MVC and run the command in Listing 20-13 again to request XML data, and you will see the following output (from which I have omitted the namespace attributes for brevity):

Content-Type	Content
application/xml; charset=utf-8	<pre> <ProductBindingTarget> <Name>Kayak</Name> <Price>275.00</Price> <CategoryId>1</CategoryId> <SupplierId>1</SupplierId> </ProductBindingTarget> </pre>

Fully Respecting Accept Headers

The MVC Framework will always use the JSON format if the Accept header contains `*/*`, indicating any format, even if there are other supported formats with a higher preference. This is an odd feature that is intended to deal with requests from browsers consistently, although it can be a source of confusion. Run the command shown in Listing 20-16 to send a request with an Accept header that requests XML but will accept any other format if XML isn't available.

Listing 20-16. Requesting an XML Response with a Fallback

```
Invoke-WebRequest http://localhost:5000/api/content/object -Headers @{Accept="application/xml,*/*;q=0.8"} |
select @{n='Content-Type';e={$_.Headers."Content-Type" }}, Content
```

Even though the Accept header tells the MVC Framework that the client prefers XML, the presence of the `*/*` fallback means that a JSON response is sent. A related problem is that a JSON response will be sent when the client requests a format that the MVC Framework hasn't been configured to produce, which you can see by running the command shown in Listing 20-17.

Listing 20-17. Requesting a PNG Response

```
Invoke-WebRequest http://localhost:5000/api/content/object -Headers @{Accept="img/png"} | select @{n='Content-
Type';e={$_.Headers."Content-Type" }}, Content
```

The commands in Listing 20-16 and Listing 20-17 both produce this response:

Content-Type	Content
-----	-----
application/json; charset=utf-8	{"name":"Kayak","price":275.00, "categoryId":1,"supplierId":1}

In both cases, the MVC Framework returns JSON data, which may not be what the client is expecting. Two configuration settings are used to tell the MVC Framework to respect the Accept setting sent by the client and not send JSON data by default. To change the configuration, add the statements shown in Listing 20-18 to the Startup class.

Listing 20-18. Configuring Content Negotiation in the Startup.cs File in the WebApp Folder

```
...
public void ConfigureServices(IServiceCollection services) {
    services.AddDbContext<DataContext>(opts => {
        opts.UseSqlServer(Configuration[
            "ConnectionStrings:ProductConnection"]);
        opts.EnableSensitiveDataLogging(true);
    });

    services.AddControllers().AddNewtonsoftJson().AddXmlSerializerFormatters();

    services.Configure<MvcNewtonsoftJsonOptions>(opts => {
        opts.SerializerSettings.NullValueHandling
            = Newtonsoft.Json.NullValueHandling.Ignore;
    });

    services.Configure<MvcOptions>(opts => {
        opts.RespectBrowserAcceptHeader = true;
        opts.ReturnHttpNotAcceptable = true;
    });
}
...
```

The options pattern is used to set the properties of a `MvcOptions` object. Setting `RespectBrowserAcceptHeader` to `true` disables the fallback to JSON when the `Accept` header contains `*/*`. Setting `ReturnHttpNotAcceptable` to `true` disables the fallback to JSON when the client requests an unsupported data format.

Restart ASP.NET Core and repeat the command from Listing 20-16. Instead of a JSON response, the format preferences specified by the `Accept` header will be respected, and an XML response will be sent. Repeat the command from Listing 20-17, and you will receive a response with the 406 status code.

```
...
Invoke-WebRequest : The remote server returned an error: (406) Not Acceptable.
...
```

Sending a 406 code indicates there is no overlap between the formats the client can handle and the formats that the MVC Framework can produce, ensuring that the client doesn't receive a data format it cannot process.

Specifying an Action Result Format

The data formats that the MVC Framework can use for an action method result can be constrained using the `Produces` attribute, as shown in Listing 20-19.

■ **Tip** The `Produces` attribute is an example of a filter, which allows attributes to alter requests and responses. See Chapter 30 for more details.

Listing 20-19. Specifying a Data Format in the `ContentController.cs` File in the `Controllers` Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Threading.Tasks;
using WebApp.Models;

namespace WebApp.Controllers {

    [ApiController]
    [Route("/api/[controller]")]
    public class ContentController : ControllerBase {
        private DataContext context;

        public ContentController(DataContext dataContext) {
            context = dataContext;
        }

        [HttpGet("string")]
        public string GetString() => "This is a string response";

        [HttpGet("object")]
        [Produces("application/json")]
        public async Task<ProductBindingTarget> GetObject() {
            Product p = await context.Products.FirstAsync();
            return new ProductBindingTarget() {
                Name = p.Name, Price = p.Price, CategoryId = p.CategoryId,
                SupplierId = p.SupplierId
            };
        }
    }
}
```

The argument for the attribute specifies the format that will be used for the result from the action, and more than one type can be specified. The `Produces` attribute restricts the types that the MVC Framework will consider when processing an `Accept` header. To see the effect of the `Produces` attribute, use a PowerShell prompt to run the command shown in Listing 20-20.

Listing 20-20. Requesting Data

```
Invoke-WebRequest http://localhost:5000/api/content/object -Headers @{Accept="application/xml,application/json;q=0.8"} | select @{n='Content-Type';e={ $_.Headers."Content-Type" }}, Content
```

The `Accept` header tells the MVC Framework that the client prefers XML data but will accept JSON. The `Produces` attribute means that XML data isn't available as the data format for the `GetObject` action method and so the JSON serializer is selected, which produces the following response:

Content-Type	Content
-----	-----
application/json; charset=utf-8	{"name":"Kayak","price":275.00, "categoryId":1,"supplierId":1}

Requesting a Format in the URL

The `Accept` header isn't always under the control of the programmer who is writing the client. In such situations, it can be helpful to allow the data format for the response to be requested using the URL. This feature is enabled by decorating an action method with the `FormatFilter` attribute and ensuring there is a format segment variable in the action method's route, as shown in Listing 20-21.

Listing 20-21. Enabling Formatting in the `ContentController.cs` File in the `Controllers` Folder

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using System.Threading.Tasks;
using WebApp.Models;

namespace WebApp.Controllers {

    [ApiController]
    [Route("/api/[controller]")]
    public class ContentController : ControllerBase {
        private DataContext context;

        public ContentController(DataContext dataContext) {
            context = dataContext;
        }

        [HttpGet("string")]
        public string GetString() => "This is a string response";

        [HttpGet("object/{format?}")]
        [FormatFilter]
        [Produces("application/json", "application/xml")]
        public async Task<ProductBindingTarget> GetObject() {
            Product p = await context.Products.FirstAsync();
            return new ProductBindingTarget() {
                Name = p.Name, Price = p.Price, CategoryId = p.CategoryId,
                SupplierId = p.SupplierId
            };
        }
    }
}
```