

murach's

# **ASP.NET CORE MVC**

**Mary Delamater**

**Joel Murach**



**TRAINING & REFERENCE**

**murach's  
ASP.NET  
Core MVC**

**Mary Delamater**

**Joel Murach**



**MIKE MURACH & ASSOCIATES, INC.**

*4340 N. Knoll Ave. • Fresno, CA 93722*

[www.murach.com](http://www.murach.com) • [murachbooks@murach.com](mailto:murachbooks@murach.com)

## **Editorial team**

**Authors:** Mary Delamater  
Joel Murach

**Editor:** Anne Boehm

**Contributor:** John Baugh

**Production:** Juliette Baylon

## **Books for web developers**

*Murach's HTML5 and CSS3*  
*Murach's JavaScript and jQuery*  
*Murach's ASP.NET Web Programming with C#*  
*Murach's PHP and MySQL*  
*Murach's Java Servlets and JSP*

## **Books on programming languages**

*Murach's C#*  
*Murach's C++ Programming*  
*Murach's Java Programming*  
*Murach's Python Programming*  
*Murach's Visual Basic*

## **Books for database programmers**

*Murach's SQL Server for Developers*  
*Murach's MySQL*  
*Murach's Oracle SQL and PL/SQL for Developers*

**For more on Murach books,  
please visit us at [www.murach.com](http://www.murach.com)**

© 2020, Mike Murach & Associates, Inc.  
All rights reserved.

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1  
ISBN: 978-1-943872-49-7

# Contents

Introduction

xvii

## Section 1 Get off to a fast start

---

Chapter 1	An introduction to web programming with ASP.NET Core MVC	3
Chapter 2	How to develop a single-page MVC web app	37
Chapter 3	How to make a web app responsive with Bootstrap	81
Chapter 4	How to develop a data-driven MVC web app	129
Chapter 5	How to manually test and debug an ASP.NET Core web app	175

## Section 2 Master the essential skills

---

Chapter 6	How to work with controllers and routing	197
Chapter 7	How to work with Razor views	227
Chapter 8	How to transfer data from controllers	279
Chapter 9	How to work with session state and cookies	321
Chapter 10	How to work with model binding	359
Chapter 11	How to validate data	397
Chapter 12	How to use EF Core	439
Chapter 13	The Bookstore website	495

## Section 3 Add more skills as you need them

---

Chapter 14	How to use dependency injection and unit testing	559
Chapter 15	How to create tag helpers, partial views, and view components	603
Chapter 16	How to authenticate and authorize users	649
Chapter 17	How to use Visual Studio Code	703

## Reference Aids

---

Appendix A	How to set up Windows for this book	727
Appendix B	How to set up macOS for this book	735



# Expanded contents

## Section 1 Get off to a fast start

---

<b>Chapter 1 An introduction to web programming with ASP.NET Core MVC</b>	
<b>An introduction to web apps .....</b>	<b>4</b>
The components of a web app.....	4
How static web pages are processed.....	6
How dynamic web pages are processed .....	8
An introduction to the MVC pattern .....	10
<b>An introduction to ASP.NET Core MVC .....</b>	<b>12</b>
Three ASP.NET programming models for web apps.....	12
Some web components of .NET and .NET Core.....	14
An introduction to ASP.NET Core middleware .....	16
How state works in a web app.....	18
<b>Tools for working with ASP.NET Core MVC apps.....</b>	<b>20</b>
An introduction to Visual Studio.....	20
An introduction to Visual Studio Code .....	22
<b>How an ASP.NET Core MVC app works .....</b>	<b>24</b>
How coding by convention works.....	24
How a controller passes a model to a view .....	26
How a view uses Razor code, tag helpers, and Bootstrap CSS classes.....	28
How the Startup.cs file configures the middleware for an app.....	30
<b>Chapter 2 How to develop a single-page MVC web app</b>	
<b>How to create a Core MVC web app.....</b>	<b>38</b>
How to start a new web app .....	38
How to select a template .....	40
How to set up the MVC folders .....	42
How to add a controller.....	44
How to add a Razor view .....	46
How to configure an MVC web app .....	48
<b>How to run a web app and fix errors.....</b>	<b>50</b>
How to run a web app .....	50
How to find and fix errors .....	52
<b>How to work with a model.....</b>	<b>54</b>
How to add a model .....	54
How to add a Razor view imports page.....	56
How to code a strongly-typed view .....	58
How to handle GET and POST requests .....	60
How to work with a strongly-typed view.....	60
The Future Value app after handling GET and POST requests .....	62
<b>How to organize the files for a view .....</b>	<b>64</b>
How to add a CSS style sheet .....	64
How to add a Razor layout, view start, and view .....	66
The code for a Razor layout, view start, and view .....	68

<b>How to validate user input .....</b>	<b>70</b>
How to set data validation rules in the model .....	70
The model class with data validation .....	72
How to check the data validation.....	74
How to display validation error messages .....	74
The Future Value app after validating data.....	74
<b>Chapter 3     How to make a web app responsive with Bootstrap</b>	
<b>An introduction to responsive web design.....</b>	<b>82</b>
A responsive user interface.....	82
How to add client-side libraries such as Bootstrap and jQuery .....	84
How to manage client-side libraries with LibMan .....	86
How to enable client-side libraries.....	88
<b>How to get started with Bootstrap .....</b>	<b>90</b>
The classes of the Bootstrap grid system.....	90
How the Bootstrap grid system works.....	92
How to work with forms .....	94
How to work with buttons, images, and jumbotron.....	96
How to work with margins and padding .....	98
The code for the view of the Future Value app .....	100
<b>More skills for Bootstrap CSS classes.....</b>	<b>102</b>
How to format HTML tables .....	102
How to align and capitalize text .....	104
How to provide context.....	106
<b>More skills for Bootstrap components .....</b>	<b>108</b>
How to work with button groups .....	108
How to work with icons and badges .....	110
How to work with button dropdowns .....	112
How to work with list groups.....	114
How to work with alerts and breadcrumbs.....	116
<b>How to work with navigation bars.....</b>	<b>118</b>
How to create navs .....	118
How to create navbars.....	120
How to position navbars.....	122
<b>Chapter 4     How to develop a data-driven MVC web app</b>	
<b>An introduction to the Movie List app .....</b>	<b>130</b>
The pages of the app .....	130
The folders and files of the app.....	132
<b>How to use EF Core .....</b>	<b>134</b>
How to add EF Core to your project.....	134
How to create a DbContext class .....	136
How to seed initial data .....	138
How to add a connection string .....	140
How to enable dependency injection .....	140
How to use migrations to create the database .....	142
<b>How to work with data .....</b>	<b>144</b>
How to select data.....	144
How to insert, update, and delete data.....	146
How to view the generated SQL statements.....	146

<b>The Movie List app.....</b>	<b>148</b>
The Home controller .....	148
The Home/Index view .....	148
The Movie controller .....	150
The Movie/Edit view .....	152
The Movie/Delete view .....	154
<b>How to work with related data .....</b>	<b>156</b>
How to relate one entity to another.....	156
How to update the DbContext class and the seed data.....	158
How to use migrations to update the database .....	160
How to select related data and display it on the Movie List page .....	162
How to display related data on the Add and Edit Movie pages.....	164
<b>How to make user-friendly URLs .....</b>	<b>166</b>
How to make URLs lowercase with a trailing slash .....	166
How to add a slug.....	168
<b>Chapter 5</b>	<b>How to manually test and debug an ASP.NET Core web app</b>
<b>How to test an ASP.NET Core web app .....</b>	<b>176</b>
How to run a web app .....	176
How to use the browser's developer tools.....	178
How to use the Internal Server Error page .....	180
How to use the Exception Helper .....	182
<b>How to use the debugger.....</b>	<b>184</b>
How to use breakpoints.....	184
How to work in break mode.....	186
How to monitor variables and expressions .....	188
How to use tracepoints.....	190
<b>Section 2 Master the essential skills</b>	
<b>Chapter 6</b>	<b>How to work with controllers and routing</b>
<b>How to use the default route.....</b>	<b>198</b>
How to configure the default route .....	198
How the default route works.....	200
How to code a simple controller and its actions .....	202
How to code a controller that uses the id segment .....	204
<b>How to create custom routes.....</b>	<b>206</b>
How to include static content in a route.....	206
How to work with multiple routing patterns.....	208
<b>How to use attribute routing.....</b>	<b>210</b>
How to change the routing for an action.....	210
More skills for changing the routing for an action .....	212
How to change the routing for a controller.....	214
<b>Best practices for creating URLs.....</b>	<b>216</b>
<b>How to work with areas.....</b>	<b>218</b>
How to set up areas .....	218
How to associate controllers with areas .....	220

## **Chapter 7 How to work with Razor views**

<b>How to use Razor syntax .....</b>	<b>228</b>
How to work with code blocks and inline expressions.....	228
How to code inline loops .....	230
How to code inline conditional statements.....	232
<b>Essential skills for Razor views .....</b>	<b>234</b>
The starting folders and files for an app.....	234
How to code controllers that return views.....	236
How to create a default layout and enable tag helpers.....	238
How to use tag helpers to generate URLs for links.....	240
Three views that use the default layout and tag helpers .....	242
The three views displayed in a browser.....	244
<b>More skills for Razor views.....</b>	<b>246</b>
More tag helpers for generating URLs for links .....	246
How to format numbers in a view.....	248
<b>How to work with a model.....</b>	<b>250</b>
How to pass a model to a view.....	250
How to display model properties in a view .....	252
How to bind model properties to HTML elements .....	254
How to bind a list of items to a <select> element.....	256
How to display a list of model objects .....	258
<b>How to work with Razor layouts.....</b>	<b>262</b>
How to create and apply a layout.....	262
How to nest layouts .....	264
How to use view context .....	268
How to use sections.....	270
<b>The Guitar Shop website.....</b>	<b>272</b>
The user interface for customers.....	272
The user interface for administrators .....	272

## **Chapter 8**

### **How to transfer data from controllers**

<b>How to use ActionResult objects.....</b>	<b>280</b>
An introduction to ActionResult subtypes .....	280
How to return ActionResult objects .....	282
<b>How to use the ViewData and ViewBag properties.....</b>	<b>284</b>
How to use the ViewData property.....	284
How to use the ViewBag property.....	286
<b>The NFL Teams 1.0 app .....</b>	<b>288</b>
The user interface .....	288
The model layer.....	290
The Home controller.....	292
The layout.....	294
The Home/Index view .....	296
<b>How to work with view models .....</b>	<b>298</b>
How to create a view model .....	298
The updated Index() action method.....	300
The updated Home/Index view .....	300
<b>How to redirect a request.....</b>	<b>302</b>
How to use the ActionResult classes for redirection .....	302
How to use the Post-Redirect-Get pattern .....	304

<b>How to use the TempData property .....</b>	<b>306</b>
How to get started with TempData .....	306
How to use methods of the TempData dictionary .....	308
<b>The NFL Teams 2.0 app.....</b>	<b>310</b>
The user interface .....	310
The view model classes.....	312
The Details() action method.....	312
The Home/Index view .....	314
The Home/Details view.....	316
<b>Chapter 9 How to work with session state and cookies</b>	
<b>How ASP.NET MVC handles state .....</b>	<b>322</b>
Six ways to maintain state .....	322
An introduction to session state.....	322
<b>How to work with session state.....</b>	<b>324</b>
How to configure an app to use session state .....	324
How to work with session state items in a controller .....	326
How to get session state values in a view .....	326
How to use JSON to store objects in session state.....	328
How to extend the ISession interface.....	330
How to use a wrapper class.....	332
<b>The NFL Teams 3.0 app.....</b>	<b>334</b>
The user interface .....	334
The session classes.....	336
The Home controller.....	338
The layout.....	340
The Home/Index view .....	342
The Home/Details view .....	342
The Favorites controller.....	344
The Favorites/Index view .....	346
<b>How to work with cookies .....</b>	<b>348</b>
How to work with session cookies.....	348
How to work with persistent cookies.....	348
<b>The NFL Teams 4.0 app.....</b>	<b>350</b>
The cookies class .....	350
The updated session class .....	350
The updated Home controller .....	352
The updated Favorites controller .....	354
<b>Chapter 10 How to work with model binding</b>	
<b>An introduction to MVC model binding .....</b>	<b>360</b>
How to use controller properties to retrieve GET and POST data.....	360
How to use model binding to retrieve GET and POST data .....	362
How to use model binding with complex types.....	364
<b>Model binding in the NFL Teams app .....</b>	<b>366</b>
An action method that binds to primitive types .....	366
An action method that binds to complex types .....	366
<b>More skills for binding data .....</b>	<b>368</b>
Two ways to code a submit button .....	368
How to use a submit button to post a name/value pair.....	370
How to post an array to an action method.....	372
How to control the source of bound values .....	374
How to control which values are bound .....	376

<b>The ToDo List app .....</b>	<b>378</b>
The user interface .....	378
The entity classes .....	380
The database context class.....	382
A utility class for filtering.....	382
The Home controller.....	384
The layout.....	388
The Home/Index view .....	388
The Home/Add view.....	392
<b>Chapter 11 How to validate data</b>	
<b>How data validation works in MVC.....</b>	<b>398</b>
The default data validation provided by model binding.....	398
How to use data attributes for validation.....	400
A Registration page with data validation .....	402
<b>How to control the user experience.....</b>	<b>404</b>
How to format validation messages with CSS.....	404
How to check validation state and use code to set validation messages .....	406
How to display model-level and property-level validation messages.....	408
How to enable client-side validation.....	410
<b>How to customize server-side validation .....</b>	<b>412</b>
How to create a custom data attribute .....	412
How to pass values to a custom data attribute.....	414
How to check multiple properties .....	416
<b>How to customize client-side validation.....</b>	<b>418</b>
How to add data attributes to the generated HTML.....	418
How to add a validation method to the jQuery validation library .....	420
How to work with remote validation .....	422
<b>The Registration app .....</b>	<b>424</b>
The user interface and CSS .....	424
The Customer and RegistrationContext classes .....	426
The MinimumAgeAttribute class.....	428
The minimum-age JavaScript file.....	428
The Validation and Register controllers .....	430
The layout.....	432
The Register/Index view.....	434
<b>Chapter 12 How to use EF Core</b>	
<b>How to create a database from code .....</b>	<b>440</b>
How to code entity and DB context classes .....	440
How to configure the database.....	442
How to manage configuration files .....	444
EF commands for working with a database .....	446
How to use EF migration commands .....	448
<b>How to work with relationships .....</b>	<b>450</b>
How entities are related .....	450
How to configure a one-to-many relationship .....	452
How to configure a one-to-one relationship .....	454
How to configure a many-to-many relationship .....	456
How to control delete behavior .....	458
<b>The Bookstore database classes .....</b>	<b>460</b>
The entity classes .....	460
The context and configuration classes .....	460

<b>How to create code from a database .....</b>	<b>466</b>
How to generate DB context and entity classes .....	466
How to configure a generated DB context class .....	468
How to modify a generated entity class.....	470
<b>How to work with data in a database .....</b>	<b>472</b>
How to query data.....	472
How to work with projections and related entities .....	474
How to insert, update, and delete data.....	476
<b>How to handle concurrency conflicts.....</b>	<b>478</b>
How to check for concurrency conflicts .....	478
How handle a concurrency exception.....	480
<b>How to encapsulate your EF code .....</b>	<b>482</b>
How to code a data access class.....	482
How to use a generic query options class .....	484
How to use the repository pattern.....	486
How to use the unit of work pattern .....	488
<b>Chapter 13    The Bookstore website</b>	
<b>The user interface and folder structure.....</b>	<b>496</b>
The end user pages.....	496
The admin user pages .....	498
The folders and files.....	500
<b>Some general-purpose code .....</b>	<b>504</b>
Extension methods for sessions and strings .....	504
The generic QueryOptions class.....	506
The generic Repository class .....	508
<b>The paging and sorting of the Author Catalog .....</b>	<b>510</b>
The custom route and the GridDTO class .....	510
The RouteDictionary class.....	512
The GridBuilder class .....	514
The Author/List view model and the Author controller.....	516
The Author/List view .....	518
<b>The paging, sorting, and filtering of the Book Catalog.....</b>	<b>520</b>
The custom route and the BooksGridDTO class.....	520
The FilterPrefix class and the updated RouteDictionary class.....	522
The BooksGridBuilder class.....	524
The BookQueryOptions and BookstoreUnitOfWork classes.....	526
The Book/List view model and the Book controller .....	528
The Book/List view .....	530
<b>The Cart page.....</b>	<b>532</b>
Extension methods for cookies .....	532
The user interface .....	534
The model classes .....	534
The Cart controller.....	542
The Cart/Index view .....	544
<b>The book search of the Admin pages.....</b>	<b>546</b>
The user interface .....	546
The SearchData and SearchViewModel classes .....	548
The Search() action methods of the Book controller .....	550
The Delete() action method of the Genre controller.....	552

## **Section 3 Add more skills as you need them**

---

### **Chapter 14 How to use dependency injection and unit testing**

<b>How to use dependency injection (DI) .....</b>	<b>560</b>
How to configure your app for DI.....	560
How to use DI with controllers.....	562
How to use DI with an HttpContextAccessor object.....	564
How to use DI with action methods.....	566
How to use DI with views.....	568
<b>How to get started with unit testing.....</b>	<b>570</b>
How unit tests work .....	570
How to add an xUnit project to a solution .....	572
How to write a unit test.....	574
How to run a unit test.....	576
<b>How to test methods that have dependencies .....</b>	<b>578</b>
How to use a fake repository object.....	578
How to use a fake TempData object .....	580
<b>How to create fake objects with Moq.....</b>	<b>582</b>
How to work with mock objects .....	582
How to mock a repository object .....	584
How to mock a TempData object .....	584
How to mock an HttpContextAccessor object.....	586
<b>The Bookstore.Tests project.....</b>	<b>588</b>
The Test Explorer.....	588
The BookControllerTests class .....	590
The AdminBookControllerTests class .....	592
The CartTests class .....	596

### **Chapter 15 How to work with tag helpers, partial views, and view components**

<b>An introduction to tag helpers .....</b>	<b>604</b>
How to register and use tag helpers .....	604
How tag helpers compare to HTML helpers .....	606
<b>How to create custom tag helpers .....</b>	<b>608</b>
How to create a custom tag helper.....	608
How to create a tag helper for a non-standard HTML element.....	610
How to use extension methods with tag helpers.....	612
How to control the scope of a tag helper .....	614
How to use a tag helper to add elements.....	616
<b>More skills for custom tag helpers .....</b>	<b>618</b>
How to use properties with a tag helper .....	618
How to work with the model property that an element is bound to .....	620
How to use dependency injection with a tag helper .....	622
How to create a conditional tag helper .....	624
How to generate URLs in a tag helper.....	626
<b>How to work with partial views.....</b>	<b>628</b>
How to create and use a partial view.....	628
How to pass data to a partial view.....	630
<b>How to work with view components .....</b>	<b>632</b>
How to create and use a view component .....	632
How to pass data to a view component.....	634
How view components can simplify an app.....	636

<b>The Bookstore app .....</b>	<b>638</b>
The Book Catalog page.....	638
The updated ActiveNavbar tag helper .....	640
The layout.....	642
The Book/List view .....	644
<b>Chapter 16 How to authenticate and authorize users</b>	
<b>An introduction to authentication .....</b>	<b>650</b>
Three types of authentication .....	650
How individual user account authentication works.....	652
An introduction to ASP.NET Identity .....	654
How to restrict access to controllers and actions.....	656
<b>How to get started with Identity .....</b>	<b>658</b>
How to add Identity classes to the DB context .....	658
How to add Identity tables to the database .....	660
How to configure the middleware for Identity .....	662
How to add Log In/Out buttons and links to the layout.....	664
How to start the Account controller .....	666
<b>How to register a user .....</b>	<b>668</b>
The Register view model .....	668
The Account/Register view .....	670
The Register() action method for POST requests.....	672
The LogOut() action method for POST requests .....	672
<b>How to log in a user .....</b>	<b>674</b>
The Login view model .....	674
The Account/Login view .....	676
The LogIn() action method for POST requests .....	678
<b>How to work with roles .....</b>	<b>680</b>
Properties and methods for working with roles.....	680
The User entity and view model .....	682
The User controller and its Index() action method.....	684
The User/Index view.....	686
Other action methods of the User controller .....	690
The code that restricts access .....	692
How to seed roles and users.....	694
<b>More skills for working with Identity.....</b>	<b>696</b>
How to change a user's password .....	696
How to add more user registration fields .....	698
<b>Chapter 17 How to use Visual Studio Code</b>	
<b>How to work with existing projects.....</b>	<b>704</b>
How to install VS Code .....	704
How to open and close a project folder.....	704
How to view and edit files.....	706
How to run and stop a project.....	708
How to create the database for a project.....	710
A summary of .NET EF Core commands .....	712
<b>How to start a new project .....</b>	<b>714</b>
How to create a new project.....	714
How to add NuGet packages to a project.....	714
How to work with the folders and files .....	716
How to install and manage client-side libraries .....	718

<b>How to debug a project .....</b>	<b>720</b>
How to set a breakpoint .....	720
How to work in break mode.....	722

**Appendix A How to set up Windows for this book**

How to install Visual Studio.....	728
How to install the source code for this book.....	730
How to create the databases for this book .....	732

**Appendix B How to set up macOS for this book**

How to install Visual Studio.....	736
How to install the source code for this book .....	738
Problems and solutions when using macOS with this book.....	740
How to install and use DB Browser for SQLite.....	742

# Introduction

If you want to learn how to develop ASP.NET Core MVC web applications, you've chosen the right book. The only prerequisites are that you already know the basics of C# and HTML/CSS. If you're new to web development, our self-paced approach helps you build competence and confidence at every turn of the page. If you're an experienced web developer, this same self-paced approach lets you learn ASP.NET Core MVC faster and more thoroughly than you've ever learned a web development framework before.

Either way, when you're through, you'll have mastered all the skills you need for developing web applications at a professional level. You'll also find that this book is the best on-the-job reference that money can buy.

## What this book does

---

To make this book as effective as possible, it is divided into three sections:

- Section 1 presents a five-chapter course that's designed to get you off to a great start. It shows how to use ASP.NET Core MVC to develop a single-page MVC web application by the end of chapter 2, and it shows how to develop a multi-page MVC web application that uses a database by the end of chapter 4. Along the way, it shows how to use Bootstrap to create responsive web applications, and it shows how to test and debug your web applications, a part of the job that many books treat too lightly or too late. At that point, you're ready for rapid progress in the sections that follow.
- Section 2 presents essential skills that every ASP.NET Core MVC developer should have. To do that, this section reviews and expands on some skills that were introduced in section 1, such as routing, Razor views, model binding, data validation, and EF (Entity Framework) Core. It also adds new skills like working with session state and cookies. Then, it finishes by presenting a realistic website that shows how all these skills fit together.

- Section 3 presents more skills that you can learn as you need them. These skills take your web development to a professional level. To be specific, this section shows how work with dependency injection (DI), unit testing, custom tag helpers, partial views, view components, authentication, and authorization. It also shows how to use Visual Studio Code, an increasingly popular alternative to the Visual Studio IDE.

## **Why you'll learn faster and better with this book**

---

Like all our books, this one has features that you won't find in competing books. That's why we believe you'll learn faster and better with our book than with any other. Here are some of those features.

- Because section 1 presents a complete subset of ASP.NET Core MVC in just 5 chapters, you're ready for productive work right away. This section also uses a self-paced approach that lets experienced programmers move more quickly and beginners work at a pace that's right for them.
- Because sections 2 and 3 present the rest of the skills that you need for developing corporate and commercial web applications, you can go from beginner to professional in a single book.
- If you page through this book, you'll see that all of the information is presented in "paired pages," with the essential syntax, guidelines, and examples on the right page and the perspective and extra explanation on the left page. This helps you learn faster by reading less...and this is the ideal reference format when you need to refresh your memory about how to do something.
- To help you put the skills presented in this book into context, this book presents complete web applications that use these skills. These applications include the HTML, CSS, and JavaScript that runs on the client as well as the Razor and C# code that runs on the server. As we see it, studying applications like these is the best way to learn ASP.NET Core MVC.

## **What software you need**

---

If you're using Windows, the only software you need for this book is Visual Studio Community, which is available for free from Microsoft's website. This provides the Visual Studio IDE, .NET Core the C# language, a built-in web server called Kestrel, and a database server called SQL Server Express LocalDB, which is a scaled back version of SQL Server. For more information, please refer to appendix A.

If you're using macOS, the only software you need for this book is Visual Studio for Mac, which is available for free from Microsoft's website. This provides the Visual Studio IDE, .NET Core the C# language, and a built-in web server called Kestrel. However, it doesn't include SQL Server Express LocalDB. Instead, you can use SQLite database files, which don't require installing a

database server. Then, to view the database files, you may want to install DB Browser for SQLite, which you can download for free from the Internet. For more information, please refer to appendix B.

## How our downloadable files can help you learn

---

If you go to our website at [www.murach.com](http://www.murach.com), you can download all the files that you need for getting the most from this book. These files include:

- the source code for the applications presented in this book
- the starting code for the exercises at the ends of the chapters
- the solutions to these exercises

These files let you test, review, and copy the code for an application. If you have any problems with the exercises, the solutions are there to help you over the learning blocks, an essential part of the learning process. In some cases, the solutions may show you a more elegant way to handle a problem, even when you've come up with a solution that works. Here again, appendixes A and B show how to download and install these files on Windows and macOS systems.

## 3 companion books for ASP.NET Core programmers

---

As you read this book, you may discover that your C# skills aren't as strong as they ought to be. In that case, we recommend that you get a copy of *Murach's C#*. It will get you up-to-speed with the C# language, and it will show you how to work with the most useful .NET Core classes.

If you discover that your HTML and CSS skills aren't as strong as they ought to be, we recommend *Murach's HTML5 and CSS3*. Or, if you want to learn more about JavaScript and jQuery, we recommend *Murach's JavaScript and jQuery*.

If you're new to these subjects, these books will get you started fast. If you already have experience with these subjects, these books make it easy for you to learn new skills whenever you need them. And after you've used these books for training, they become the best on-the-job references you've ever used.

## Support materials for instructors and trainers

---

If you're a college instructor or corporate trainer who would like to use this book as a course text, we offer a full set of the support materials you need for a turnkey course. That includes:

- instructional objectives that help your students focus on the skills that they need to develop
- projects that let your students demonstrate how well they have mastered those skills
- test banks that let you measure how well your students have mastered those skills

- a complete set of PowerPoint slides that you can use to review and reinforce the content of the book

Instructors tell us that this is everything they need for a course without all the busywork that you get from other publishers.

To learn more about our instructor's materials, please go to our website at [www.murachforinstructors.com](http://www.murachforinstructors.com) if you're an instructor. If you're a trainer, please go to [www.murach.com](http://www.murach.com) and click on the *Courseware for Trainers* link, or contact Kelly at 1-800-221-5528 or [kelly@murach.com](mailto:kelly@murach.com).

Please remember, though, that the primary component for a successful ASP.NET Core MVC course is this book. Because your students will learn faster and more thoroughly when they use our book, they will have better questions and be more prepared when they come to class. And because our paired pages are so good for reference, your students will be able to review for tests and do their projects more efficiently.

## **Please let us know how this book works for you**

---

When we started writing this book, we had two goals. First, we wanted to make this the best-ever book for learning how to develop ASP.NET Core MVC web applications. To do that right, we knew we had to make the book easy enough for beginners and yet still teach the skills needed by professionals.

Second, we wanted to make this the best-ever book for experienced developers who want to add ASP.NET Core MVC to their skillsets. To do that right, we've carefully selected the content, organized it from simple to complex in each chapter, and packed the book full of sample web applications. That's why we believe that this book will help experienced developers learn ASP.NET Core MVC faster and better than ever. And when they're done, this book will be their best-ever on-the-job reference.

Now, we hope we've succeeded. We thank you for buying this book. We wish you all the best with your ASP.NET Core MVC programming. And if you have any comments, we always appreciate hearing from you.



Mary Delamater  
[maryd@techknowsolve.com](mailto:maryd@techknowsolve.com)



Joel Murach  
[joel@murach.com](mailto:joel@murach.com)

# Section 1

## Get off to a fast start

This section presents the essential skills for coding, testing, and debugging ASP.NET Core MVC (Model-View-Controller) web apps. After chapter 1 introduces you to the concepts and terms that you need to know for ASP.NET Core MVC programming, chapter 2 shows you how to develop a one-page web app. This includes writing the C# code for the model and controller classes as well as writing the HTML, CSS, and Razor code for the view files that are used to generate the user interface and present it to the user. That gets you off to a fast start!

Chapter 3 shows you how to use the CSS classes that are available from an open-source library known as Bootstrap to make an ASP.NET Core MVC app work well for all screen sizes. This is known as responsive design. Then, chapter 4 shows you how to develop a three-page Movie List app that works with data in a database. Finally, chapter 5 shows you how to manually test and debug ASP.NET Core MVC apps.

When you finish all five chapters, you'll be able to develop real-world apps of your own. You'll have a solid understanding of how ASP.NET Core MVC works. And you'll be ready for rapid progress as you read the other sections of the book.



# 1

# An introduction to web programming with ASP.NET Core MVC

This chapter introduces you to the basic concepts of web programming and ASP.NET Core MVC. Here, you'll learn how web apps work, how ASP.NET Core MVC apps work, and what software you need for developing these apps. When you finish this chapter, you'll have the background that you need for learning how to develop ASP.NET Core MVC apps.

<b>An introduction to web apps.....</b>	<b>4</b>
The components of a web app .....	4
How static web pages are processed.....	6
How dynamic web pages are processed .....	8
An introduction to the MVC pattern .....	10
<b>An introduction to ASP.NET Core MVC.....</b>	<b>12</b>
Three ASP.NET programming models for web apps.....	12
Some web components of .NET and .NET Core.....	14
An introduction to ASP.NET Core middleware .....	16
How state works in a web app .....	18
<b>Tools for working with ASP.NET Core MVC apps .....</b>	<b>20</b>
An introduction to Visual Studio.....	20
An introduction to Visual Studio Code .....	22
<b>How an ASP.NET Core MVC app works .....</b>	<b>24</b>
How coding by convention works .....	24
How a controller passes a model to a view.....	26
How a view uses Razor code, tag helpers, and Bootstrap CSS classes.....	28
How the Startup.cs file configures the middleware for an app .....	30
<b>Perspective .....</b>	<b>32</b>

## An introduction to web apps

---

A *web app*, also known as a *web application*, consists of a set of *web pages* that are generated in response to user requests. The Internet has many different types of web apps such as search engines, online stores, news sites, social sites, music streaming, video streaming, and games.

### The components of a web app

---

The diagram in figure 1-1 shows that web apps consist of *clients* and a *web server*. The clients are the computers and mobile devices that use the web apps. They often access the web pages through programs known as *web browsers*. The web server holds the files that generate the pages of a web app.

A *network* is a system that allows clients and servers to communicate. The *Internet* is a large network that consists of many smaller networks. In a diagram like the one in this figure, the “cloud” represents the network or Internet that connects the clients and servers.

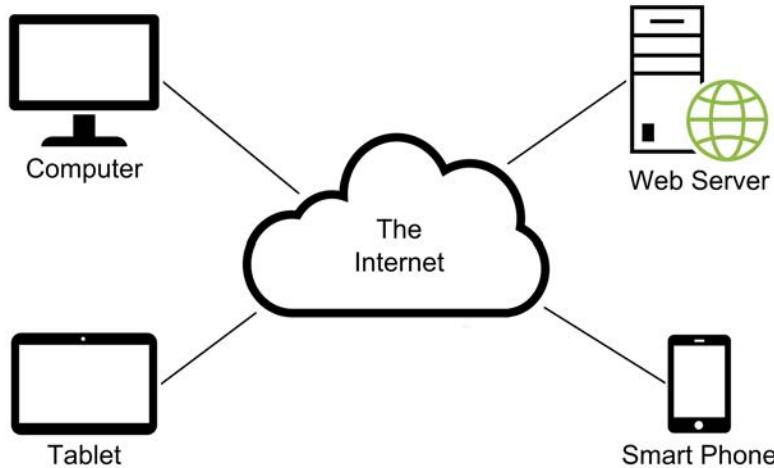
Networks can be categorized by size. A *local area network (LAN)* is a small network of computers that are near each other and can communicate with each other over short distances. Computers in a LAN are typically in the same building or adjacent buildings. This type of network is often called an *intranet*, and it can run web apps that are used throughout a company.

By contrast, a *wide area network (WAN)* consists of multiple LANs that have been connected. To pass information from one client to another, a router determines which network is closest to the destination and sends the information over that network. A WAN can be owned privately by one company or it can be shared by multiple companies.

An *Internet service provider (ISP)* is a company that owns a WAN that is connected to the Internet. An ISP leases access to its network to companies that need to be connected to the Internet. When you develop production web apps, you will often implement them through an ISP.

To access a web page from a browser, you can type a *URL (Uniform Resource Locator)* into the browser’s address area and press Enter. The URL starts with the *protocol*, which is usually HTTPS. It is followed by the *domain name* and the folder or directory *path* to the file that is requested. If the filename is omitted from the URL, the web server looks for a default file in the specified directory. The default files usually include index.html, index.htm, default.html, and default.htm.

## The components of a web app



## The components of an HTTP URL

`https://www.modulemedia.com/ourwork/index.html`

protocol      domain name      path      filename

### Description

- A *web app*, also known as a *web application*, consists of clients, a web server, and a network.
- The *clients* often use *web browsers* to request web pages from the web server. Today, the clients are often computers, smart phones, or tablets.
- The *web server* returns the pages that are requested to the browser.
- A *network* connects the clients to the web server.
- To request a page from a web server, the user can type the address of a web page, called a *URL (Uniform Resource Locator)*, into the browser's address area and then press the Enter key.
- A URL consists of the *protocol* (usually, HTTPS), *domain name*, *path*, and *filename*. If you omit the protocol, HTTPS is assumed. If you omit the filename, the web server typically looks for a file named index.html, index.htm, default.html, or default.htm.
- An *intranet* is a *local area network (LAN)* that connects computers that are near each other, usually within the same building.
- The *Internet* is a network that consists of many *wide area networks (WANs)*, and each of those consists of two or more LANs.
- The *cloud* refers to software and services that run on the Internet, instead of locally on your computer. This term implies that you don't have to understand how it works to be able to use it.
- An *Internet service provider (ISP)* owns a WAN that is connected to the Internet.

---

Figure 1-1 The components of a web app

## How static web pages are processed

---

A *static web page* is stored in a file that contains hard-coded content that doesn't change each time it is requested. This type of web page is sent directly from the web server to the web browser when the browser requests it. You can spot static pages in a web browser by looking at the extension in the address bar. If the extension is .htm or .html, the page is probably a static web page. Static web pages were common in the early days of web programming but are less common today.

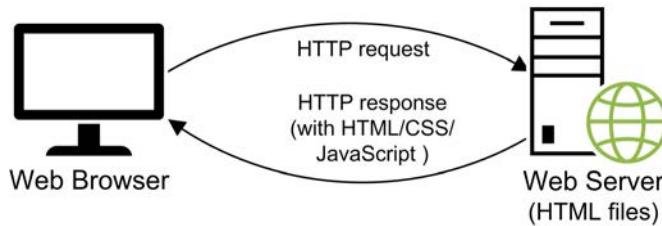
The diagram in figure 1-2 shows how a web server processes a request for a static web page. This process begins when a client requests a web page in a web browser. To do that, the user can either enter the URL of the page in the browser's address bar or click a link in the current page that specifies the next page to load.

In either case, the web browser builds a request for the web page and sends it to the web server. This request, known as an *HTTP request*, is formatted using the *Hypertext Transfer Protocol (HTTP)*, which lets the web server know which file is being requested. To keep the communication over the network secure, most web apps use an extension of HTTP known as *Hypertext Transfer Protocol Secure (HTTPS)*.

When the web server receives the HTTP request, it retrieves the requested file from the disk drive. This file contains the *HTML (Hypertext Markup Language)* for the requested page with references to any CSS or JavaScript files needed by the page. Then, the web server sends the HTML, CSS, and JavaScript back to the browser as part of an *HTTP response*.

When the browser receives the HTTP response, it *renders* (translates) the HTML, CSS, and JavaScript into a web page that is displayed in the browser. Then, the user can view the content. If the user requests another page, either by clicking a link or entering another URL into the browser's address bar, the process begins again.

## How a web server processes a static web page



### A simple HTTP request

```
GET / HTTP/1.1  
Host: www.example.com
```

### A simple HTTP response

```
HTTP/1.1 200 OK  
Content-Type: text/html  
Content-Length: 136  
Server: Apache/2.2.3
```

```
<html>  
<head>  
    <title>Example Web Page</title>  
</head>  
<body>  
    <p>This is a sample web page</p>  
</body>  
</html>
```

### Three protocols that web apps depend upon

- *Hypertext Transfer Protocol (HTTP)* is the protocol that web browsers and web servers use to communicate. It sets the specifications for HTTP requests and responses.
- *Hypertext Transfer Protocol Secure (HTTPS)* is an extension of the Hypertext Transfer Protocol (HTTP). It is used for secure communication over a network.
- *Transmission Control Protocol/Internet Protocol (TCP/IP)* is a suite of protocols that let two computers communicate over a network.

### Description

- *Hypertext Markup Language (HTML)* is used to design the pages of a web app.
- A *static web page* is built from an HTML file that's stored on the web server and doesn't change. The filenames for static web pages usually have .htm or .html extensions.
- When the user requests a static web page, the browser sends an *HTTP request* to the web server that includes the name of the file that's being requested.
- When the web server receives the request, it retrieves the HTML, CSS, and JavaScript for the requested file and sends it back to the browser as part of an *HTTP response*.
- When the browser receives the HTTP response, it *renders* the HTML, CSS, and JavaScript into a web page that is displayed in the browser.

---

Figure 1-2 How static web pages are processed

## How dynamic web pages are processed

---

A *dynamic web page* is a page that's generated by a program on an *application server*. This program uses the data that's sent with the HTTP request to generate the HTML that's returned to the server. For example, if the HTTP request includes a product code, the program can retrieve the data for that product from a *database server*. Then, it can insert that data into a web page and return it as part of an HTTP response.

The diagram in figure 1-3 shows how a web server processes a dynamic web page. The process begins when the user requests a page in a web browser. To do that, the user can click a link that specifies the dynamic page to load or click a button that submits a form that contains the data that the dynamic page should process.

In either case, the web browser builds an HTTP request and sends it to the web server. This request includes whatever data the app needs for processing the request. If, for example, the user has entered data into a form, that data is included in the HTTP request.

When the web server receives the HTTP request, the server examines the request to identify the application server that should process the request. The web server then forwards the request to that application server.

Next, the application server retrieves the appropriate program. It also loads any form data that the user submitted. Then, it executes the program. As the program executes, it generates the HTML, CSS, and JavaScript for the web page. If necessary, the program also requests data from a database server and uses that data as part of the web page it is generating.

When the program is finished, the application server sends the dynamically generated HTML back to the web server. Then, the web server sends the HTML, CSS, and JavaScript for the page back to the browser in an HTTP response.

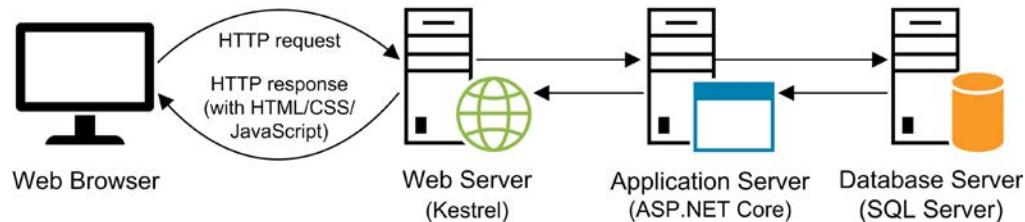
When the web browser receives the HTTP response, it renders the HTML/CSS/JavaScript and displays the web page. To do that, the web browser doesn't need to know whether this HTML is for a static page or a dynamic page. Either way, it just renders the HTML/CSS/JavaScript.

When the page is displayed, the user can view the content. Then, when the user requests another page, the process begins again. The process that begins with the user requesting a web page and ends with the server sending a response back to the client is called a *round trip*.

ASP.NET Core apps use ASP.NET Core as the application server. In addition, they typically use a cross-platform web server known as *Kestrel* and a *DBMS (database management system)* named Microsoft SQL Server running on the database server.

Prior to ASP.NET Core, ASP.NET apps typically used a Windows-only web server known as *Internet Information Services (IIS)*. And, as you would expect, they used ASP.NET as the application server.

## How a web server processes a dynamic web page



### Description

- A *dynamic web page* is a web page that's generated by a program running on a server.
- When a web server receives a request for a dynamic web page, it looks up the extension of the requested file and passes the request to the appropriate *application server* for processing.
- When the application server receives the request, it runs the appropriate program. Often, this program uses data that's sent in the HTTP request to get related data from a *database management system (DBMS)* running on a *database server*.
- When the application server finishes processing the data, it generates the HTML, CSS, and JavaScript for a web page and returns it to the web server. Then, the web server returns the HTML, CSS, and JavaScript to the web browser as part of an HTTP response.
- The process that starts when a client requests a page and ends when the page is returned to the browser is called a *round trip*.
- *Kestrel* is a new cross-platform web server that also functions as an application server for ASP.NET Core apps.
- *Internet Information Services (IIS)* is an older Windows-only web server that also functions as an application server for older ASP.NET apps.
- SQL Server is typically used as the database server for ASP.NET Core and ASP.NET apps.

---

Figure 1-3 How dynamic web pages are processed

## An introduction to the MVC pattern

---

The *MVC (Model-View-Controller) pattern* is commonly used to structure web apps that have significant processing requirements. Most modern web development frameworks today use some form of the MVC pattern. As you'll learn in this book, ASP.NET Core provides extensive support for the MVC pattern.

The MVC pattern divides an app into separate components where each component has a specific area of concern. This is known as *separation of concerns*. Although it requires a little more work to set up an app like this, it leads to many benefits. For example, it makes it easier to have different members of a team work on different components, it makes it possible to automate testing of individual components, and it makes it possible to swap out one component for another component. In short, it makes apps easier to code, test, debug, and maintain.

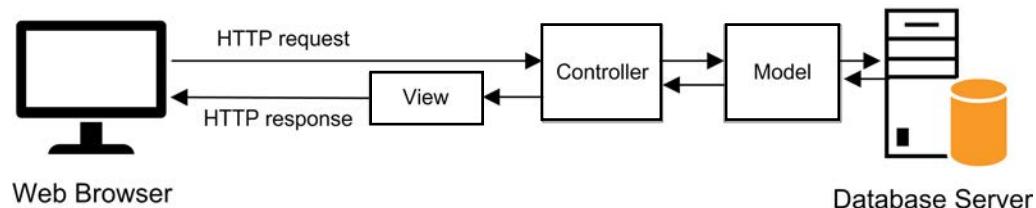
Figure 1-4 presents a diagram that shows the components of the MVC pattern and how they work together. To start, the *model* is in charge of data. Specifically, it gets and updates the data in a data store such as a database, it applies business rules to that data, and it validates that data.

The *view* is in charge of the user interface. Specifically, it creates the HTML, CSS, and JavaScript that the app sends to the browser in response to the browser's HTTP request.

The *controller* is in charge of controlling the flow of data between the model and the view. Specifically, it receives the HTTP request from the browser, determines what data to get from the model, and then sends the data from the model to the appropriate view.

It's important to note that each component should stick to its own area of concern and be as independent as possible. For example, the model should retrieve and validate data but it shouldn't have anything to do with formatting or displaying it. Similarly, the controller should move data between the model and the view but it shouldn't apply any business rules to it.

## The MVC pattern



## Components of the MVC pattern

- The *model* consists of the code that provides the data access and business logic.
- The *view* consists of the code that generates the user interface and presents it to the user.
- The *controller* consists of the code that receives requests from users, gets the appropriate data and stores it in the model, and passes the model to the appropriate view.

## Benefits of the MVC pattern

- Makes it easier to have different members of a team work on different components.
- Makes it possible to automate testing of individual components.
- Makes it possible to swap out one component for another component.
- Makes the app easier to maintain.

## Drawbacks of the MVC pattern

- Requires more work to set up.

## Description

- The *MVC (Model-View-Controller) pattern* is commonly used to structure web apps that have significant processing requirements.
- The MVC pattern breaks web apps into separate component parts. This is known as *separation of concerns*, and it leads to many benefits including making the app easier to code, test, debug, and maintain.

---

Figure 1-4 An introduction to the MVC pattern

## An introduction to ASP.NET Core MVC

Now that you understand some concepts that apply to most web apps, you're ready to learn some concepts that are more specific to ASP.NET Core MVC apps.

### Three ASP.NET programming models for web apps

Since 2002, Microsoft has developed many ASP.NET programming models. Figure 1-5 summarizes three of the most popular. Of these programming models, Web Forms is the oldest and most established, and Core MVC is the newest and most cutting edge.

In 2002, Microsoft released ASP.NET Web Forms. It provides for *RAD* (*Rapid Application Development*) by letting developers build web pages by working with a design surface in a way that's similar to the way that developers build desktop apps with Windows Forms.

This approach made a lot of sense back in 2002 when many developers were switching from desktop development to web development. However, this approach has many problems including poor performance, inadequate separation of concerns, lack of support for automated testing, and limited control over the HTML/CSS/JavaScript that's returned to the browser. In addition, Web Forms uses the ASP.NET Framework, which is proprietary and only runs on Windows.

In 2007, Microsoft released ASP.NET MVC. It uses the MVC pattern that's used by many other web development platforms. It fixes many of the perceived problems with web forms to provide better performance, separation of concerns, support for automated testing, and a high degree of control over the HTML/CSS/JavaScript that's returned to the browser. However, it uses the same proprietary, Windows-only ASP.NET Framework as Web Forms. As a result, it can't run on Internet servers that use other operating systems such as Linux, which is widely used for Internet servers.

In 2015, Microsoft released ASP.NET Core MVC. This programming model uses an MVC service to implement the MVC pattern. It provides all of the functionality of ASP.NET MVC, but with better performance, more modularity, and cleaner code. In addition, it's built on the new ASP.NET Core platform that's open source and can run on multiple platforms including Windows, macOS, and Linux. As a result, it can run on most Internet servers.

While developing the different versions of MVC and Core MVC, Microsoft used different names and version numbers. For example, ASP.NET Core 1.0 was originally going to be called ASP.NET 5. But then, Microsoft decided ASP.NET Core was a fundamentally different technology that should have a new name and number. Information on the Internet sometimes uses the old names and version numbers that were used during development, not the official names and numbers of the final release. So, be aware of this when you search for information about ASP.NET on the Internet.

## ASP.NET Web Forms

- Released in 2002.
- Provides for *RAD (Rapid Application Development)* by letting developers build web pages by working with a design surface in a way that's similar to Windows Forms.
- Has many problems including poor performance, inadequate separation of concerns, lack of support for automated testing, and limited control over the HTML/CSS/JavaScript that's returned to the browser.
- Uses the ASP.NET Framework, which is proprietary and only runs on Windows.

## ASP.NET MVC

- Released in 2007.
- Uses the MVC pattern that's used by many other web development platforms.
- Fixes many of the perceived problems with web forms to provide better performance, separation of concerns, support for automated testing, and a high degree of control over the HTML/CSS/JavaScript that's returned to the browser.
- Uses the same proprietary, Windows-only ASP.NET Framework as Web Forms.

## ASP.NET Core MVC

- Released in 2015.
- Uses a service to implement the MVC pattern that's used by many other web development platforms.
- Provides all of the functionality of ASP.NET MVC but with better performance, more modularity, and cleaner code.
- Is built on the open-source ASP.NET Core platform that can run on multiple platforms including Windows, macOS, and Linux.

## Description

- Since 2002, Microsoft has developed many ASP.NET programming models. Of these programming models, ASP.NET Core MVC is the newest.
- While developing the different versions of MVC and Core MVC, Microsoft used different names and version numbers.
- Information on the Internet sometimes uses the old names and version numbers that were used during development, not official names and numbers of the final release.
- There are breaking changes between ASP.NET and ASP.NET Core.
- There are breaking changes between different versions of ASP.NET Core such as between 2.0 and 3.0.

---

Figure 1-5 Three ASP.NET programming models for web apps

As you work with ASP.NET apps, you should be aware that there are breaking changes between ASP.NET and ASP.NET Core. Similarly, there are breaking changes between different versions of ASP.NET Core such as between 2.0 and 3.0. As a result, if you want to use code that's available from the Internet, you need to make sure that code is compatible with the version of ASP.NET that you're using.

## **Some web components of .NET and .NET Core**

---

Figure 1-6 shows some web components of the older *.NET Framework*, also referred to as just *.NET*. In addition, it shows some components of the newer *.NET Core* platform.

There are several important differences between .NET and .NET Core. Specifically, the .NET Core platform is open source and supports multiple operating systems including Windows, macOS, and Linux. By contrast, the .NET Framework is proprietary and only supports the Windows operating system. On the other hand, the .NET Framework supports all three programming models described in the previous figure. However, of these three programming models, the .NET Core platform only supports ASP.NET Core MVC.

So, when developing ASP.NET Core MVC apps, should you use the .NET Framework or .NET Core? Well, if you want to deploy your app to an Internet server that runs on Linux, you should use .NET Core. On the other hand, if you want to use components that are available from the .NET Framework but not from .NET Core, you should use the .NET Framework. For new development, it usually makes sense to choose .NET Core since that typically results in improved performance and greater flexibility in where you can deploy your app.

## Some web components of .NET and .NET Core



### Description

- The *.NET Framework*, also known as just *.NET*, only supports the Windows operating system.
- The *.NET Core* platform is open source and supports multiple operating systems including Windows, macOS, and Linux.
- ASP.NET Web Forms apps use services of the ASP.NET Framework, which uses services of the .NET Framework.
- ASP.NET MVC apps work by using services of the ASP.NET Framework.
- ASP.NET Core MVC apps work by using services of ASP.NET Core, which uses services of .NET Core or the .NET Framework.

Figure 1-6 Some web components of .NET and .NET Core

## An introduction to ASP.NET Core middleware

---

An ASP.NET Core app allows you to configure the *middleware* components that are in the HTTP request and response *pipeline*. This gives developers a lot of flexibility in how an app works.

Each middleware component can modify the HTTP request before passing it on to the next component in the pipeline. Similarly, each middleware component can modify the HTTP response before passing it to the next component in the pipeline.

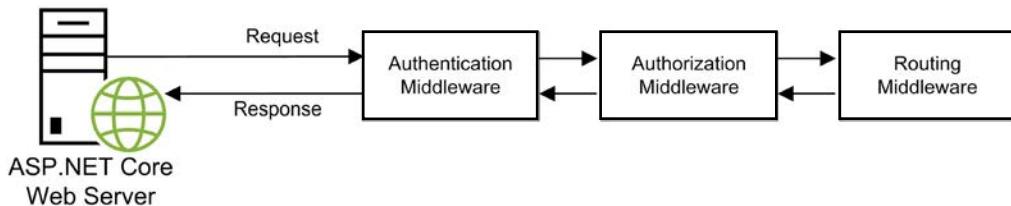
The diagrams in figure 1-7 should give you an idea of how middleware works. To start, the authentication middleware authenticates the request by confirming the identity of the client making the request. To do that, it may need to edit the content of the request. Then, it passes the request on to the authorization middleware.

If the authorization middleware determines that the client is authorized to make the request, it passes the request on to the routing middleware. This middleware uses a controller to generate the content for a response by working with the model and view layers. Then, it passes the response back to the web server. Along the way, the authorization and authentication middleware can edit the content of the response.

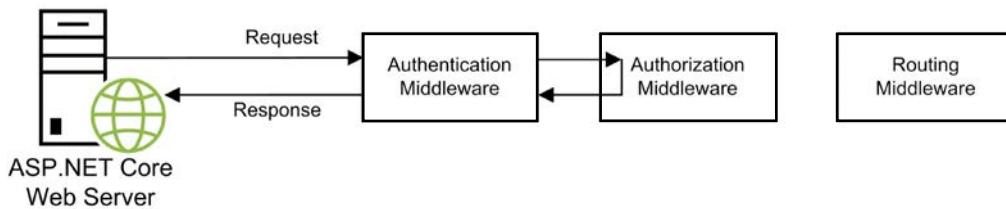
If the authorization middleware determines that the client is not authorized to make the request, it short circuits the request by passing a response back to the web server. Along the way, the authentication middleware can edit the content of the response.

An ASP.NET Core app typically has a number of middleware components configured such as static files middleware, authentication middleware, and endpoint routing middleware. Near the end of this chapter, you'll be introduced to the code that configures endpoint routing middleware for a simple ASP.NET Core MVC app. Most of the apps presented in this book configure this middleware. As you progress through this book, you'll learn how to configure additional middleware that's necessary to support the most important features of ASP.NET Core MVC.

## A request that makes it through all middleware in the pipeline



## A request that's short circuited by a middleware component in the pipeline



### Middleware can...

- Generate the content for a response
- Edit the content of a request
- Edit the content of a response
- Short circuit a request

### Description

- An ASP.NET Core app allows you to configure the *middleware* components that are in the HTTP request and response *pipeline*.

---

Figure 1-7 An introduction to ASP.NET Core middleware

## How state works in a web app

---

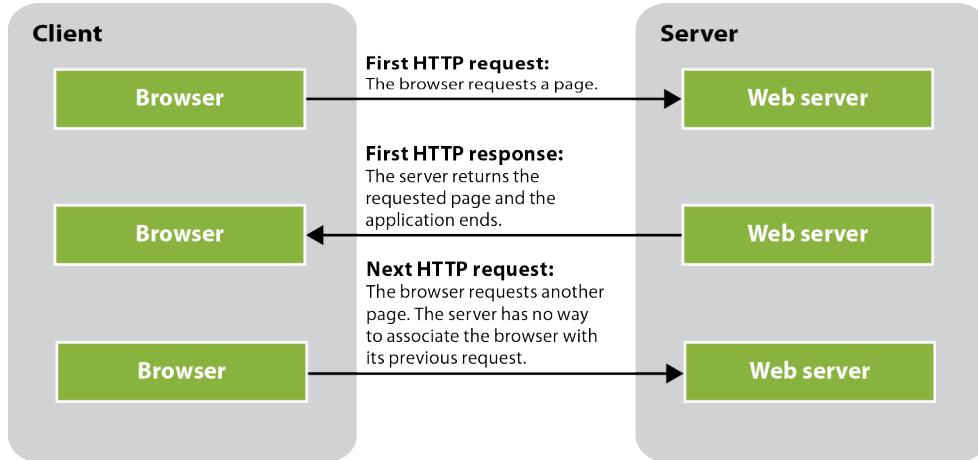
*State* refers to the current status of the properties, variables, and other data maintained by an app for a single user. As an app runs, it must maintain a separate state for each user currently accessing the app.

Unlike some other protocols, HTTP is a *stateless protocol*. That means that it doesn't keep track of state between round trips. Once a browser makes a request and receives a response, the app terminates and its state is lost as shown by the diagram in figure 1-8. As a result, when the web browser makes a subsequent request, the web server has no way to associate the current request with the previous request.

ASP.NET Web Forms attempted to hide the stateless nature of a web app from developers by automatically maintaining state. This often resulted in large amounts of data being transferred with each request and response, which led to poor performance.

ASP.NET Core MVC does not attempt to hide the stateless nature of a web app from the developer. Instead, it provides features to handle state in a way that give developers control over each HTTP request and response. When used wisely, this can lead to excellent performance.

## Why state is difficult to track in a web app



## Concepts

- *State* refers to the current status of the properties, variables, and other data maintained by an app for a single user.
- HTTP is a *stateless protocol*. That means that it doesn't keep track of state between round trips. Once a browser makes a request and receives a response, the app terminates and its state is lost.

## Description

- ASP.NET Web Forms attempted to hide the stateless nature of a web app from developers by automatically maintaining state. This led to poor performance.
- ASP.NET Core MVC does not attempt to hide the stateless nature of a web app from developers. Instead, it provides features to handle state in a way that gives developers control over each HTTP request and response.

Figure 1-8 How state works in a web app

## Tools for working with ASP.NET Core MVC apps

---

Now that you know some of the concepts behind ASP.NET Core MVC, you're ready to learn more about the tools that you can use to develop ASP.NET Core MVC apps. Some developers prefer to use an *integrated development environment (IDE)*, which is a tool that provides all of the functionality that you need for developing an app in one place. Other developers prefer to use a code editor to enter and edit code and a command line to compile and run it. There are pros and cons to each approach. Fortunately, an excellent IDE and code editor are both available for free to ASP.NET Core MVC developers.

### An introduction to Visual Studio

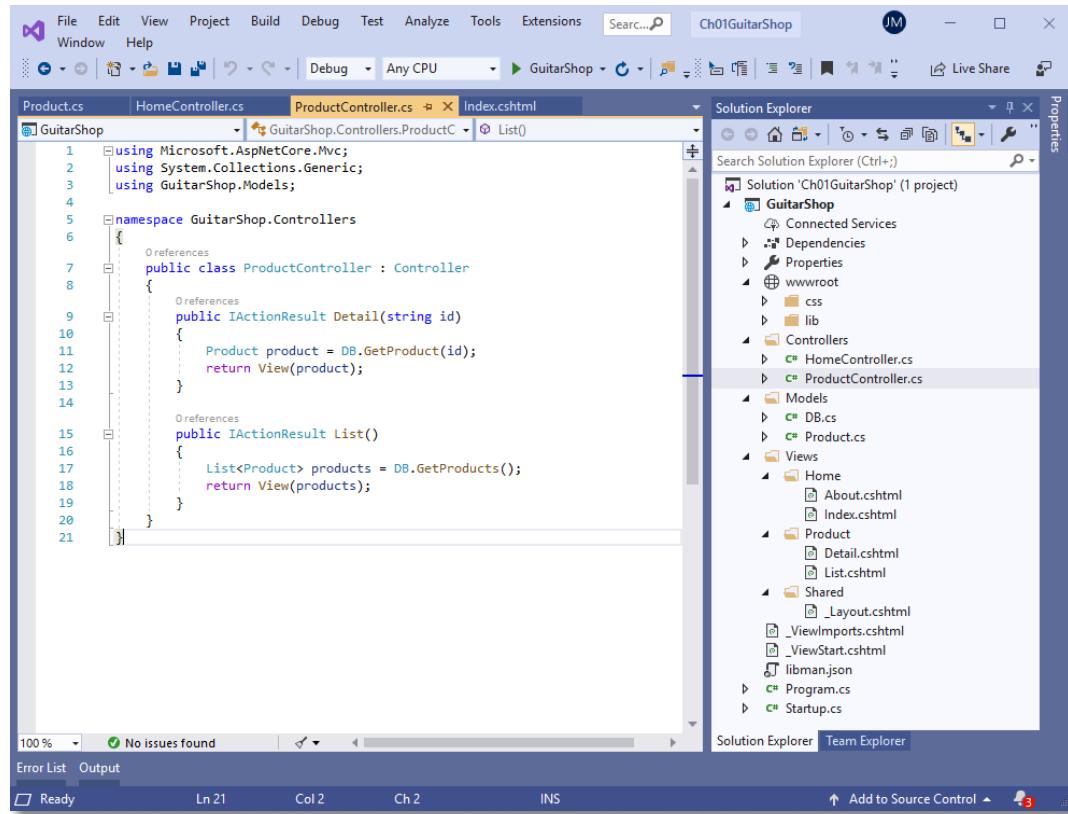
---

*Visual Studio*, also known just as VS, is the most popular IDE for developing ASP.NET Core apps. Microsoft provides a Community Edition that's available for free and runs on Windows, as well as a free version that runs on macOS.

Figure 1-9 shows Visual Studio after it has opened an ASP.NET Core MVC app and displayed the code for one of its controllers. In addition, this figure lists some of the features provided by Visual Studio. For example, you can compile and run an app with a single keystroke.

Visual Studio is an established product that has been around for decades, and it's a great tool for learning. That's why we present it throughout this book, starting in the next chapter.

## Visual Studio with an ASP.NET Core MVC app



## Features

- IntelliSense code completion makes it easy to enter code.
- Automatic compilation allows you to compile and run an app with a single keystroke.
- Integrated debugger makes it easy to find and fix bugs.
- Runs on Windows and macOS.

## Description

- An *Integrated Development Environment (IDE)* is a tool that provides all of the functionality that you need for developing web apps.
- *Visual Studio*, also known as VS, is the most popular IDE for ASP.NET Core web development.
- Starting in the next chapter, this book shows how to use Visual Studio to develop ASP.NET Core MVC apps.

Figure 1-9 An introduction to Visual Studio

## An introduction to Visual Studio Code

---

*Visual Studio Code*, also known as *VS Code*, is a *code editor* that's becoming popular with Microsoft developers. Like Visual Studio, VS Code can be used to develop all types of .NET apps, including ASP.NET Core apps. Since it doesn't provide as many features as an IDE like Visual Studio, some developers find VS Code easier to use. In addition, VS Code typically starts and runs faster than Visual Studio, especially on slower computers.

Figure 1-10 shows VS Code after it has opened the same app as the previous figure. If you compare these two figures, you'll see that they look very similar. In short, both provide an Explorer window that lets you navigate through the files for an app, and both provide a code editor for editing the code for an app. The main difference is that Visual Studio provides more features than VS Code. That's either good or bad, depending on how you look at it.

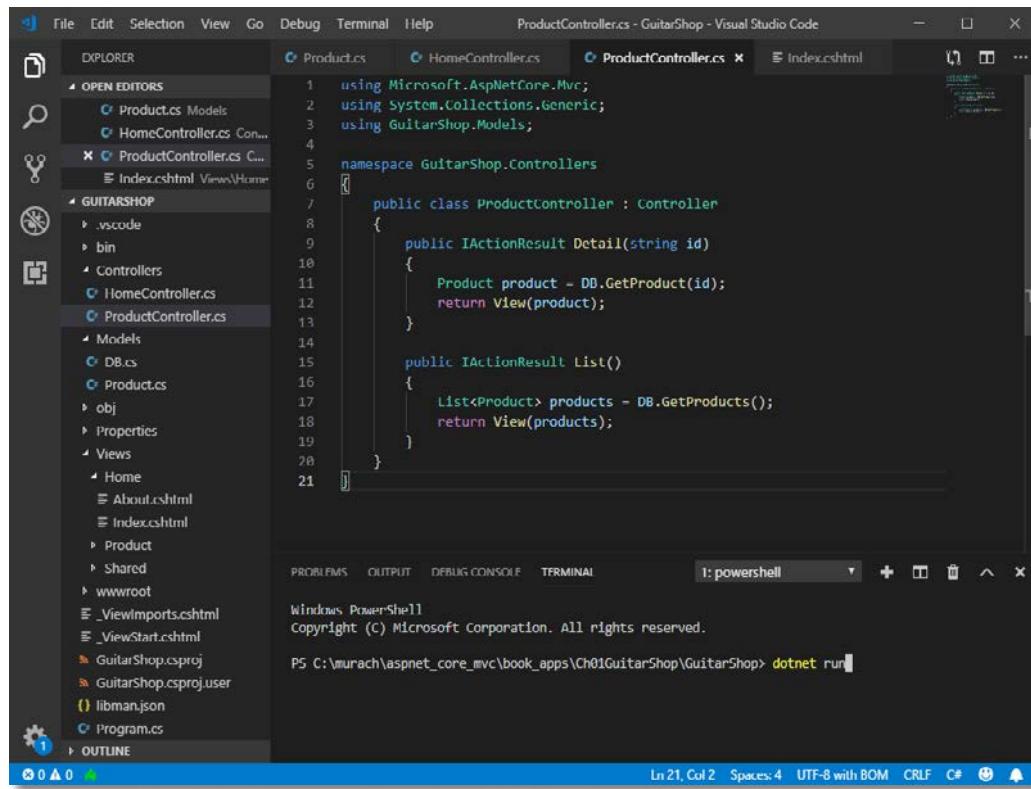
This figure also lists some of the features provided by VS Code. If you compare this list of features with the list of features in the previous figure, you'll see that they're mostly the same. The main difference is that VS Code runs on Linux, and Visual Studio does not. So, even though Visual Studio provides more features, VS Code provides all of the most essential features that make it easy to develop ASP.NET Core apps.

When you use VS Code, you can use its Terminal window to use a *command line* to enter and execute the commands that build and run an app. With Windows, for example, you typically use the command line known as Windows PowerShell. In this figure, the Windows PowerShell command line is displayed across the bottom of the code editor window.

When you choose between Visual Studio and VS Code, you may want to consider that VS Code has a less restrictive license than the Community Edition of Visual Studio and adheres to a truly open-source, open-use model. As a result, you may want or need to use VS Code if licensing for commercial development becomes an issue.

As mentioned in the previous figure, this book shows how to use Visual Studio, starting in the next chapter. However, chapter 17 shows how to use VS Code. If you prefer using a code editor and command line to using a more full-featured IDE, you can use VS Code. Then, you can refer to chapter 17 whenever you need help with a task. This should make it easy to use VS Code with this book.

## Visual Studio Code with an ASP.NET Core MVC app



## Features

- IntelliSense code completion makes it easy to enter code.
- Automatic compilation allows you to compile and run an app with a single keystroke.
- Integrated debugger makes it easy to find and fix bugs.
- Runs everywhere (Windows, macOS, and Linux).

## Description

- *Visual Studio Code*, also known as *VS Code*, is a *code editor* that you can use to work with ASP.NET Core apps.
- When you use VS Code, you can use its Terminal window to use a *command line* to enter and execute the commands that build and run an app.
- VS Code has a less restrictive license than the Community Edition of Visual Studio and adheres to a truly open-source, open-use model.
- Chapter 17 shows how to use VS Code to develop ASP.NET Core MVC apps.

---

Figure 1-10 An introduction to Visual Studio Code

## How an ASP.NET Core MVC app works

Now that you've been introduced to some general concepts and tools for working with ASP.NET Core MVC, you're ready to learn more about how an ASP.NET Core MVC app works. To do that, the next few figures present the overall structure of an ASP.NET Core MVC app as well as some key snippets of code. This should give you a general idea of how an ASP.NET Core MVC app works. Then, in the next chapter, you'll learn all of the details for coding such an app.

### How coding by convention works

ASP.NET Core MVC uses a software design paradigm known as *convention over configuration*, or *coding by convention*. This reduces the amount of configuration that developers need to do if they follow certain conventions.

That's why figure 1-11 shows some of the folders and files for an MVC web app that follows the standard MVC conventions. And that's why this figure lists some of these conventions.

To start, the top-level folder for a web app is known as its *root folder*, also known as the *root directory*. In this figure, the *GuitarShop* folder is the root folder. Within the root folder, you typically use .cs files to store the C# classes that define controllers and models.

All controller classes should be stored in a folder named *Controllers* or one of its subfolders. In this figure, the *Controllers* folder contains the files for two classes named *HomeController* and *ProductController*. Both of these classes have a suffix of "Controller". This isn't required for controllers to work, but it's a standard convention that makes it easy for other programmers to quickly identify these classes as controller classes.

All model classes should be stored in a folder named *Models* or one of its subfolders. In this figure, the *Models* folder contains a single file for a model class named *Product*.

All view files should be stored in a folder named *Views* or one of its subfolders. In addition, the subfolders of the *Views* folder should correspond to the controller classes. For example, the *Views/Home* folder should store the view files for the *HomeController* class, and the *Views/Product* folder should store the view files for the *ProductController* class. These view files (.cshtml files) contain *Razor views* that define views for the app. In this figure, the views for the app are defined by view files such as the *Home/Index.cshtml*, *Home/About.cshtml*, *Product/Detail.cshtml*, and *Product/List.cshtml* files.

All static files such as image files, CSS files, and JavaScript files should be stored in a folder named *wwwroot*. The static files for an app can include CSS libraries such as Bootstrap or JavaScript libraries such as jQuery. In addition, they can include custom CSS or JavaScript files that override the code in these libraries.

## Some of the folders and files for a web app

```
GuitarShop
  /Controllers
    /HomeController.cs
    /ProductController.cs
  /Models
    /Product.cs
  /Views
    /Home
      /About.cshtml
      /Index.cshtml
    /Product
      /Detail.cshtml
      /List.cshtml
  /wwwroot
    /css
      custom.css
    /images
    /js
      custom.js
    /lib
      /bootstrap
      /jquery
  /Startup.cs
```

## Some naming conventions for an ASP.NET Core MVC app

- All controller classes should be stored in a folder named Controllers or one of its subfolders.
- All model classes should be stored in a folder named Models or one of its subfolders.
- All view files should be stored in a folder named Views or one of its subfolders.
- All static files such as image files, CSS files, and JavaScript files should be stored in a folder named wwwroot or one of its subfolders.
- All controller classes should have a suffix of “Controller”.

## Description

- ASP.NET Core MVC uses a software design paradigm known as *convention over configuration*, or *coding by convention*. This reduces the amount of configuration that developers need to do if they follow certain conventions.
- The top-level folder for a web app is known as its *root folder* or *root directory*.
- You typically use C# classes (.cs files) to define controllers and models.
- You typically use *Razor views* (.cshtml files) to define views.
- The static files for an app can include CSS libraries such as Bootstrap or JavaScript libraries such as jQuery. In addition, they can include custom CSS or JavaScript files that override the code in these libraries.

---

Figure 1-11 How ASP.NET Core MVC uses coding by convention

## How a controller passes a model to a view

---

In ASP.NET Core MVC, a *model* is a C# class that defines the data objects and business rules for the app. Figure 1-12 begins by showing the code for a model named Product. This class is stored in the GuitarShop.Models namespace, and it defines a simple Product object.

In the Product class, you shouldn't have any trouble understanding the ID, Name, and Price properties. However, the Slug property is a read-only property that's created by replacing all spaces in the product's name with dashes. This property can be used in the URL that's used to display the product. You'll learn more about using slugs in chapter 4.

In ASP.NET Core MVC, a *controller* is a C# class that typically inherits the Microsoft.AspNetCore.Mvc.Controller class. In this figure, the ProductController class inherits this class and defines two actions. In ASP.NET Core MVC, an *action* is a method of a controller that returns an *action result*. To do that, a method can use the built-in View() method to return a type of action result known as a *view result* that's created by merging the model (if there is one) into the HTML code specified in the view file.

In this figure, for example, the Detail() method is an action. It starts by using the parameter named id to get a Product object that corresponds to that id from the database. This Product object is the model for the view. Then, it passes this model to the view, so the view can display its data.

Similarly, the List() method starts by getting the model, which is a List of Product objects, from the database. Then, it passes that List of Product objects to the view for display.

### The code for a model class named Product

```
namespace GuitarShop.Models
{
    public class Product
    {
        public int ProductID { get; set; }
        public string Name { get; set; }
        public decimal Price { get; set; }
        public string Slug => Name.Replace(' ', '-');
    }
}
```

### The code for a controller class named ProductController

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

namespace GuitarShop.Controllers
{
    public class ProductController : Controller
    {
        public IActionResult Detail(string id)
        {
            Product product = DB.GetProduct(id);
            return View(product); // passes model to Product/Detail view
        }

        public IActionResult List()
        {
            List<Product> products = DB.GetProducts();
            return View(products); // passes model to Product/List view
        }
    }
}
```

### Description

- A *model* is a C# class that defines the data objects and business rules for the app.
- With ASP.NET Core MVC, a *controller* is a C# class that typically inherits the Microsoft.AspNetCore.Mvc.Controller class.
- With ASP.NET Core MVC, an *action* is a method of a controller that returns an *action result*.
- An action method can use the View() method to return a type of action result known as a *view result* that's created by merging the model (if there is one) into the corresponding view file.

---

Figure 1-12 How a controller passes a model to a view

## How a view uses Razor code, tag helpers, and Bootstrap CSS classes

---

Most of a typical view file consists of HTML elements. In figure 1-13, for example, the code uses an `<h1>` element to display a level-1 heading. It uses the `<table>`, `<tr>`, and `<td>` elements to display a table that has three rows with two columns. And it uses an `<a>` element to display a link that's formatted to look like a button.

However, a view file can also contain *Razor code* that allows you to use C# to get data from the model, format it, and display it. Razor code begins with an @ sign. In this figure, all of the Razor code is highlighted. As you can see, there are several different ways to work with Razor code.

To start, the `@model` directive specifies the class for the model, and the `@Model` property allows you to access a model object that's created from the specified class. In this figure, the `@model` directive at the top of the class specifies that this view uses the `Product` class as the model. Then, it uses the `@Model` property to get data from the `ProductID`, `Name`, and `Price` properties and insert this data into the HTML elements. In addition, it uses the `ToString()` method of the `Price` property to format the number as currency with 2 decimal places.

After the `@model` directive, there's an @ sign followed by braces `({})` that identifies a block of C# statements. In this figure, the block of statements contains a single statement that uses the built-in `ViewBag` property to set the title of the web page. However, if necessary, you could add other C# statements to this code block.

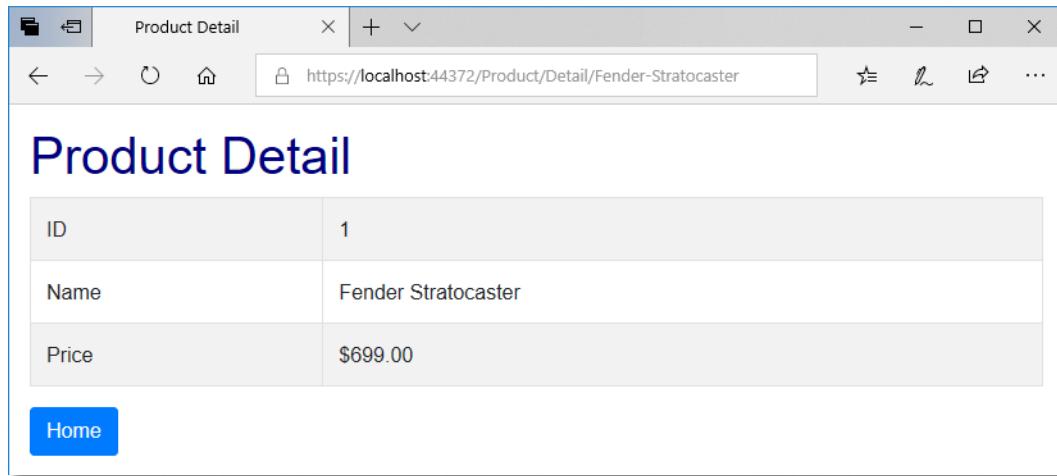
In a view, all HTML attributes that start with “asp-” are *tag helpers* that can make it easier to code HTML attributes. Tag helpers are defined by C# classes, and many are available from ASP.NET Core MVC. As a result, you can use them to make coding HTML attributes easier. In this figure, for example, the `<a>` element uses the `asp-controller` and `asp-action` tag helpers to specify the controller and action method for the link. Alternately, you could code an `href` attribute that specified a URL to the correct controller and action. However, this isn't as flexible and easy to maintain as using the tag helpers.

In a view, it's common to use the `class` attribute of an HTML element to specify CSS classes from *Bootstrap*, a popular open-source CSS library that's often used with ASP.NET Core MVC. In this figure, for example, the `<table>` element uses the `class` attribute to specify three CSS classes: `table`, `table-bordered`, and `table-striped`. That's why the table that's displayed in the browser looks so professional. Similarly, the `<a>` element uses the `class` attribute to specify two CSS classes: `btn` and `btn-primary`. That's why the link that's displayed in the browser looks like a button.

## The code for a view file named Product/Detail.cshtml

```
@model Product
 @{
    ViewBag.Title = "Product Detail";
}
<h1>Product Detail</h1>
<table class="table table-bordered table-striped">
    <tr>
        <td>ID</td><td>@Model.ProductID</td>
    </tr>
    <tr>
        <td>Name</td><td>@Model.Name</td>
    </tr>
    <tr>
        <td>Price</td><td>@Model.Price.ToString("C2")</td>
    </tr>
</table>
<a asp-controller="Home" asp-action="Index"
    class="btn btn-primary">Home</a>
```

## The view displayed in a browser



## Description

- Most of a typical view file consists of standard HTML elements.
- The @model directive specifies the class for the model, and the @Model property allows you to access the model object that's passed to the view from the controller.
- The @ sign followed by braces ({} ) identifies a block of C# statements. Within the block, you can code one or more C# statements.
- All HTML attributes that start with “asp-” are *tag helpers*. Tag helpers are defined by C# classes and make it easier to work with HTML elements. Many tag helpers are built into ASP.NET Core MVC.
- The class attribute of an HTML element can specify CSS classes from *Bootstrap*, a popular open-source CSS library that's often used with ASP.NET Core MVC.

---

Figure 1-13 How a view uses Razor code, tag helpers, and Bootstrap CSS classes

## How the Startup.cs file configures the middleware for an app

---

Figure 1-14 begins by showing the contents of the Startup.cs file for the Guitar Shop app. This file contains the code that configures the middleware that's used by the app. This builds the middleware pipeline for the app.

The Startup.cs file begins by importing the necessary Microsoft namespaces for building a hosting environment for the app. Then, it defines a class named Startup that contains two methods.

The ConfigureServices() method contains the code that adds services to the app. For a simple ASP.NET Core MVC app like the Guitar Shop app, you only need to use the AddControllersWithViews() method to add services that support controllers, models, views, tag helpers, and so on.

The Configure() method contains the code that configures the services that have been added. In this figure, the first statement specifies that the middleware should display exception pages that are designed for developers, not end users. That's typically what you want when you're developing an app. The second statement specifies that the app can use the static files that are in the wwwroot directory such as the Bootstrap library. The third statement marks the position in the middleware pipeline where a routing decision is made for a URL. And the fourth statement marks the position in the pipeline where the routing decision is executed.

To start, the UseEndpoints() method maps the controllers and their actions to a pattern for request URLs that's known as the default route. This route identifies the Home controller as the default controller and the Index() action method as the default action. As a result, if the request URL specifies the root folder (/), the app executes the Index() action method of the Home controller. However, if a request URL specifies /Product/Detail, the app executes the Detail() action method of the Product controller.

The default route also identifies a third segment of the URL that's a parameter named id, and it uses a question mark (?) to indicate that this parameter is optional. As a result, if a request URL specifies /Product/Detail/Fender-Stratocaster, the app passes an id parameter of “Fender-Stratocaster” to the Detail action of the Product controller. This allows the Product controller to retrieve data about the product that has a slug of “Fender-Stratocaster”.

## The Startup.cs file

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.DependencyInjection;

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

        public void Configure(IApplicationBuilder app)
        {
            app.UseDeveloperExceptionPage();
            app.UseStaticFiles();
            app.UseRouting();
            app.UseEndpoints(endpoints =>
            {
                endpoints.MapControllerRoute(
                    name: "default",
                    pattern: "{controller=Home}/{action=Index}/{id?}");
            });
        }
    }
}
```

## How request URLs map to controllers and actions by default

Request URL	Controller	Action
<a href="http://localhost">http://localhost</a>	Home	Index
<a href="http://localhost/Home">http://localhost/Home</a>	Home	Index
<a href="http://localhost/Home/About">http://localhost/Home/About</a>	Home	About
<a href="http://localhost/Product&gt;List">http://localhost/Product/List</a>	Product	List
<a href="http://localhost/Product/Detail">http://localhost/Product/Detail</a>	Product	Detail

## Description

- The Startup.cs file contains the code that configures the middleware that's used by the app. In other words, it builds the middleware pipeline for the app.
- The ConfigureServices() method contains the code that adds services to the app.
- The Configure() method contains the code that identifies which services to use and provides additional configuration if necessary.
- By convention, the routing system identifies the Home controller as the default controller and the Index() action method as the default action.

---

Figure 1-14 How the Startup.cs file configures the middleware for an app

# Perspective

---

Now that you've read this chapter, you should have a general understanding of how ASP.NET Core MVC apps work and what software you need for developing these apps. With that as background, you're ready to gain valuable hands-on experience by learning how to develop an ASP.NET Core MVC app as shown in the next chapter.

## Terms

---

web app	Kestrel
web application	Internet Information Services (IIS)
web page	MVC (Model-View-Controller)
client	pattern
web browser	model
web server	view
network	controller
URL (Uniform Resource Locator)	separation of concerns
protocol	RAD (Rapid Application Development)
domain name	.NET
path	.NET Framework
filename	.NET Core
intranet	middleware
Internet	pipeline
LAN (local area network)	state
WAN (wide area network)	stateless protocol
cloud	IDE (Integrated Development Environment)
ISP (Internet service provider)	VS (Visual Studio)
HTTP (Hypertext Transfer Protocol)	VS Code (Visual Studio Code)
HTTPS (Hypertext Transfer Protocol Secure)	code editor
TCP/IP (Transmission Control Protocol/Internet Protocol)	command line
HTML (Hypertext Markup Language)	convention over configuration
static web page	coding by convention
HTTP request	root folder
HTTP response	root directory
dynamic web page	Razor view
application server	action
database management system (DBMS)	action result
database server	view result
round trip	tag helper
	Bootstrap

## Summary

---

- A *web app*, also known as a *web application*, consists of a set of *web pages* that are run by clients, a web server, and a network. *Clients* often use *web browsers* to request web pages from the web server. The *web server* returns the requested pages.
- A *local area network (LAN)*, or *intranet*, connects computers that are near each other. By contrast, the *Internet* consists of many *wide area networks (WANs)*.
- One way to access a web page is to type a *URL (Uniform Resource Locator)* into the address area of a browser and press Enter. A URL consists of the *protocol* (usually, HTTPS), *domain name*, *path*, and *filename*.
- To request a web page, the web browser sends an *HTTP request* to the web server. Then, the web server gets the HTML/CSS/JavaScript for the requested page and sends it back to the browser in an *HTTP response*. Last, the browser *renders* the HTML/CSS/JavaScript into a web page.
- A *static web page* is a page that is the same each time it's retrieved. By contrast, the HTML/CSS/JavaScript for a *dynamic web page* is generated by a server-side program, so its HTML/CSS/JavaScript can change from one request to another. Either way, HTML/CSS/JavaScript is returned to the browser.
- For ASP.NET Core MVC apps, the application server is ASP.NET Core, the web server is usually *Kestrel*, and the database server usually runs a *database management system (DBMS)* like SQL Server.
- One way to develop ASP.NET apps is to use Web Forms. This is similar to using Windows Forms and encourages *Rapid Application Development (RAD)*.
- Another way to develop ASP.NET apps is to use ASP.NET Core *MVC (Model-View-Controller)*. It provides better *separation of concerns*, which leads to many benefits including making the app easier to code, test, debug, and maintain.
- The older *.NET Framework*, also known as just *.NET*, only supports the Windows operating system. The newer *.NET Core* platform is open source and supports Windows, macOS, and Linux.
- An ASP.NET Core app allows you to configure the *middleware* components that are in the HTTP request and response *pipeline*.
- HTTP is called a *stateless protocol* because it doesn't keep track of the data (*state*) between *round trips*. However, ASP.NET Core provides features to handle state in a way that gives developers control over each HTTP request and response.

- *Visual Studio*, also known as just *VS*, is an *integrated development environment (IDE)* that provides all of the functionality that you need for developing web apps.
- *Visual Studio Code*, also known as *VS Code*, is a *code editor* that you can use to develop ASP.NET Core apps.
- With ASP.NET Core MVC, you typically use *Razor views* (.cshtml files) to define the user interface and present it to the user.
- With ASP.NET Core MVC, an *action* is a method of a controller that returns an *action result*. To do that, it's common to use the `View()` method to return a type of action result known as a *view result* that's typically created by merging the model into the corresponding view file.
- All HTML attributes that start with “asp-” are *tag helpers*. Tag helpers are defined by C# classes and make it easier to work with HTML elements. Many tag helpers are built into ASP.NET Core MVC.
- The `class` attribute of an HTML element can specify CSS classes from *Bootstrap*, a popular open-source CSS library that's often used with ASP.NET Core MVC.

## Before you do the exercises for this book...

Before you do the exercises for this book, you should install the software that's required for this book, and you should download the source code for this book. Appendixes A (Windows) and B (macOS) show how to do that.

### Exercise 1-1 Use Visual Studio to run the Guitar Shop app

In this exercise, you'll run the Guitar Shop app. This will test whether you've successfully installed the software and source code for this book.

#### Start Visual Studio and open the Guitar Shop app

1. Start Visual Studio.
2. From the menu system, select the File→Open→Project/Solution item. Or, if you are in the Start window for Visual Studio, you can click the “Open a project or solution” button. In the dialog box that's displayed, navigate to this folder:

`/aspnet_core_mvc/book_apps/Ch01GuitarShop`

Then, select the Ch01GuitarShop.sln file and click the Open button.

#### Run the Guitar Shop app

3. Press Ctrl+F5 to run the app. That should display the Home page for the Guitar Shop app in Visual Studio's default web browser. If you get messages about trusting and installing an SSL certificate, you can click Yes. And if a web page is displayed indicating that the connection is not private, you can click the link to proceed.
4. Click the “View Fender Stratocaster” link. This should display the Product Detail page for that product.
5. Click the Home button to return to the Home page.
6. Click the “View Products” link. This should display a list of products.
7. Click the View link for the product named Gibson Les Paul. This should display the Product Detail page for that product.
8. Close the browser tab or window for the app, and then switch back to Visual Studio.
9. In the Solution Explorer, expand the Controllers, Models, and Views folders and review some of the code.

#### Close the Guitar Shop app and exit Visual Studio

10. Use the File→Close Solution command to close the solution.
11. Exit Visual Studio.

## Exercise 1-2 Use Visual Studio Code to run the Guitar Shop app (optional)

This exercise is optional. However, if you want to use Visual Studio Code instead of Visual Studio, or if you just want to see how VS Code compares to VS, you can use this exercise to take VS Code for a test drive. But first, you should install VS Code as described in chapter 17.

### Start VS Code and open the Guitar Shop app

1. Start VS Code.
2. From the menus, select the File→Open Folder item. In the dialog box that's displayed, navigate to this folder:  
`/aspnet_core_mvc/book_apps/Ch01GuitarShop/GuitarShop`  
Then, click the Select Folder button. This should open the GuitarShop project that's in this folder.
3. If you get any error messages, click on the appropriate buttons to fix them. VS Code should be able to do this for you.

### Run the Guitar Shop app

4. Press Ctrl+F5 to run the app. That should display the Home page for the Guitar Shop app in your default web browser. If you get an error that says something like “Configured debug type ‘coreclr’ is not supported”, you need to install the “C# for Visual Studio Code (powered by OmniSharp)” extension under Extensions at the bottom of the sidebar.
5. Click the “View Fender Stratocaster” link. This should display the Product Detail page for that product.
6. Click the Home button to return to the Home page.
7. Click the “View Products” link. This should display a list of products.
8. Click the View link for the product named Gibson Les Paul. This should display the Product Detail page for that product.
9. Close the browser tab or window for the app, and then switch back to VS Code.
10. In the Explorer window, expand the Controllers, Models, and Views folders and review some of the code.

### Close the Guitar Shop app and exit VS Code

11. Select the File→Close Folder item to close the folder. This should close the Guitar Shop project that's in this folder.
12. Exit VS Code.

# 2

## How to develop a single-page MVC web app

In the last chapter, you were introduced to the basic concepts of web programming and ASP.NET Core MVC. Now, this chapter shows you how to develop a single-page ASP.NET Core MVC web app that calculates the future value of a series of investments. To do that, this chapter shows how to use Visual Studio because it's the most established tool for working with .NET apps. However, if you prefer to use Visual Studio Code instead, you can refer to chapter 17 to learn how to work with Visual Studio Code.

<b>How to create a Core MVC web app .....</b>	<b>38</b>
How to start a new web app.....	38
How to select a template.....	40
How to set up the MVC folders .....	42
How to add a controller .....	44
How to add a Razor view.....	46
How to configure an MVC web app .....	48
<b>How to run a web app and fix errors .....</b>	<b>50</b>
How to run a web app .....	50
How to find and fix errors.....	52
<b>How to work with a model .....</b>	<b>54</b>
How to add a model .....	54
How to add a Razor view imports page.....	56
How to code a strongly-typed view .....	58
How to handle GET and POST requests .....	60
How to work with a strongly-typed view .....	60
The Future Value app after handling GET and POST requests.....	62
<b>How to organize the files for a view .....</b>	<b>64</b>
How to add a CSS style sheet .....	64
How to add a Razor layout, view start, and view .....	66
The code for a Razor layout, view start, and view .....	68
<b>How to validate user input.....</b>	<b>70</b>
How to set data validation rules in the model .....	70
The model class with data validation .....	72
How to check the data validation .....	74
How to display validation error messages.....	74
The Future Value app after validating data.....	74
<b>Perspective .....</b>	<b>76</b>

## How to create a Core MVC web app

---

This chapter starts by showing how to create a new ASP.NET Core MVC web app. This includes all the skills you need to set up the folders and files for a simple MVC app that uses a controller to pass some data to a view that displays the data on a web page.

### How to start a new web app

---

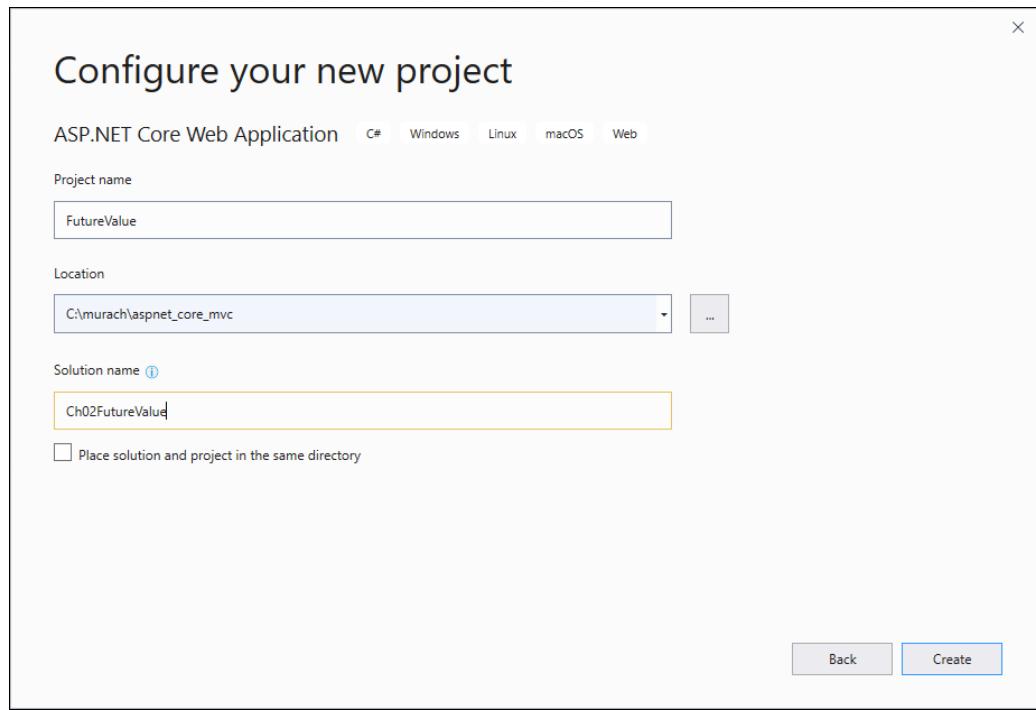
To start a new web app, you can use the procedure shown in figure 2-1. To create an ASP.NET Core web app, you can choose the ASP.NET Core Web Application item from the first dialog that's displayed (not shown in this figure.). Then, you can use the dialog shown in this figure to specify the name of the project and the location of the project. In this figure, the name of the project is FutureValue. To specify the location of the project, you typically click the Browse button to select a different folder or use the Location drop-down list to select a location you've used recently. In this figure, the folder is C:\murach\aspnet\_core\_mvc.

If the “Place solution and project in the same directory” box is unchecked, Visual Studio creates a folder for the solution and a subfolder for the project. In this figure, this check box is unchecked and a name of Ch02FutureValue is specified for the solution. As a result, Visual Studio creates a folder for the solution named Ch02FutureValue and a subfolder for the project named FutureValue.

Although this figure shows how to create a web app that runs on .NET Core, which is cross-platform, it's also possible to create a web app that runs on the Windows-only .NET Framework. To do that, you just select the ASP.NET Web Application (.NET Framework) template instead of the ASP.NET Core Web Application template in step 3 of this procedure.

So, when would you want to create a web app that runs on the ASP.NET Framework instead of ASP.NET Core? As a general rule, you probably wouldn't want to do that for new development. However, it's possible that you may want to use the ASP.NET Framework so you can use legacy components that aren't available from ASP.NET Core.

## The dialog for starting a new web app



## How to start a new ASP.NET Core MVC web app

1. Start Visual Studio.
2. From the menu system, select File→New→Project.
3. Select the ASP.NET Core Web Application item and click the Next button.
4. Enter a project name.
5. Specify the location (folder). To do that, you can click the Browse button.
6. If necessary, edit the solution name and click the Create button.
7. Use the resulting dialog to select the Web Application (Model-View-Controller) template or the Empty template.

### Description

- If the “Place solution and project in the same directory” box is unchecked, Visual Studio creates a folder for the solution and a subfolder for the project. Otherwise, these files are stored in the same folder.

---

Figure 2-1 How to start a new web app

## How to select a template

---

When starting an ASP.NET Core web app, Visual Studio provides several *templates* that you can use. These templates are displayed by the dialog shown in figure 2-2. In general, we recommend using the Web Application (Model-View-Controller) template because it makes it easy to start an MVC web app. However, if you want to manually build your web app from scratch, you can use the Empty template. Either template should work fine for the Future Value app presented in this chapter.

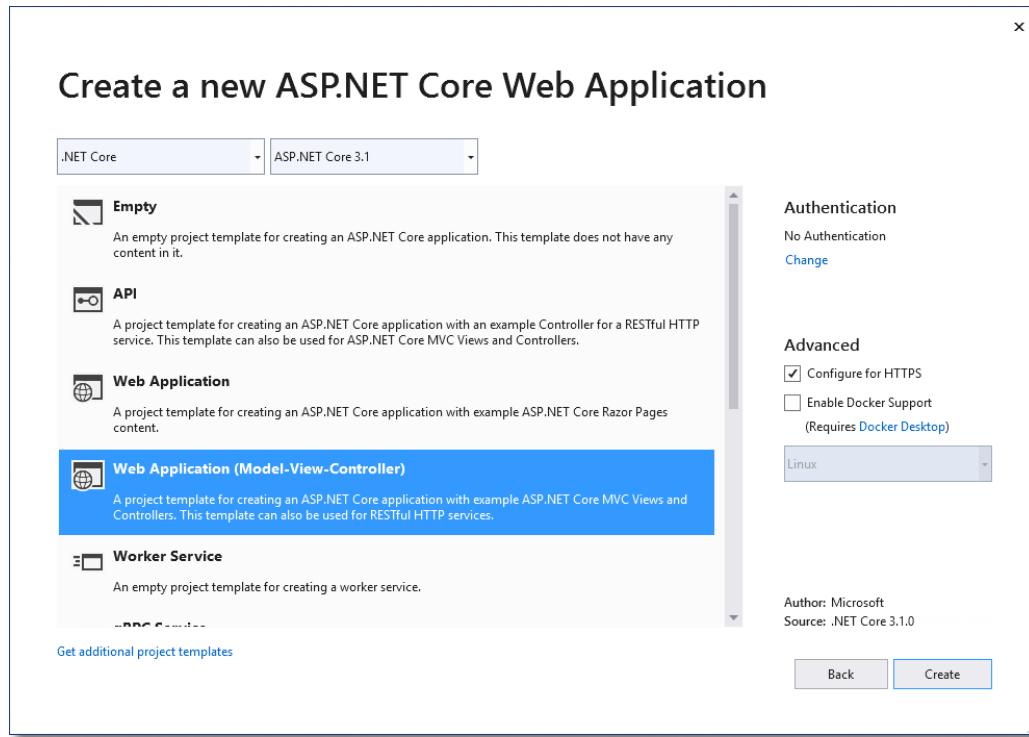
The template you choose determines the folders and files that Visual Studio adds to the project when it creates your web app. In this book, you'll learn how to use the two templates summarized in this figure.

The MVC template sets up the starting folders and files for an ASP.NET Core MVC web app, including a configuration file that configures the default routing for the app. When you use this template, you typically start by deleting files and code that you don't need. Then, as you develop the app, you add the files and code you do need.

The Empty template provides two starting files for an ASP.NET Core app. When you use this template, you must manually add the folders and files for an MVC web app and configure the middleware for the request pipeline. Although we recommend using the MVC template for this chapter, you can also use the Empty template if you prefer to add the folders and files you need instead of deleting the ones that you don't need.

Although you won't learn how to use any of the other templates in this book, you might want to experiment with them. For example, the API template sets up the starting folders and files for a RESTful web service. Also, the Web Application template sets up an ASP.NET Core app that uses Razor pages, which is a different approach than using the MVC pattern with controllers and Razor views.

## The dialog that displays the project templates



## The templates presented in this book

Template	Contains...
MVC	Starting folders and files for an ASP.NET Core MVC web app.
Empty	Two starting files for an ASP.NET Core app.

## Description

- When starting an ASP.NET Core web app, Visual Studio provides several *templates* that you can use.
- For this chapter, we recommend using the Web Application (Model-View-Controller) template, also known as the MVC template, because it makes it easy to start an ASP.NET Core MVC web app.
- If you want to manually build your web app from scratch, you can use the Empty template.

Figure 2-2 How to select a template

## How to set up the MVC folders

---

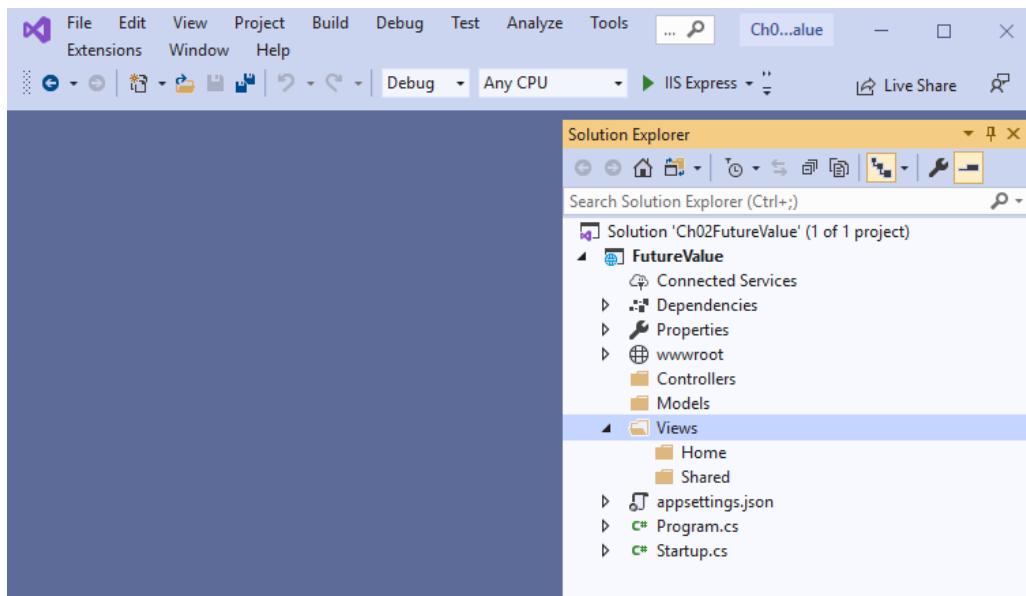
The procedures in figure 2-3 show how to set up the folders for an MVC web app. This works whether you started the web app from the MVC template or the Empty template.

If you started from the MVC template, you can delete all files from the folders named Models, Views, and Controllers. In addition, you can delete all files from the Home and Shared subfolders of the Views folder. This is an excellent approach when you're getting started.

Alternately, you can leave these files and edit them as described later in this chapter. However, this approach leaves extra files and code that can lead to errors, and it doesn't give you practice adding the files described in this chapter. As a result, it's better to use this approach later, after you've learned more about developing MVC web apps.

If you started from the Empty template, you need to add the folders named Models, Views, and Controllers. Then, you need to add the Home and Shared subfolders of the Views folder.

## Visual Studio after the folders have been set up for an MVC web app



### How to delete unnecessary files from the MVC template

1. Expand the Controllers folder and delete all files in that folder.
2. Expand the Models folder and delete all files in that folder.
3. Expand the Views folder and its subfolders and delete all files in those folders, but don't delete the folders.

### How to add folders to the Empty template

1. Add the Controllers, Models, and Views folders.
2. Within the Views folder, add the Home and Shared folders.

### Description

- To add a folder, you can right-click a node and select Add→New Folder.
- To delete a folder or file, you can right-click the folder or file and select Delete.

Figure 2-3 How to set up the MVC folders

## How to add a controller

---

The procedure in figure 2-4 shows how to add a controller file to a web app. In addition, it shows the code for the controller after it has been edited so it's a good starting point for the Future Value app presented in this chapter.

A *controller* is a C# class that inherits from the Controller class that's available from the Microsoft.AspNetCore.Mvc namespace. When you develop an MVC app, it's common to place controller classes in a namespace that consists of the project name, a dot, and the name of the folder that stores the controllers. In this figure, for example, the HomeController class is stored in the FutureValue.Controllers namespace. If you follow the procedure in this figure, this is where the HomeController class is placed by default.

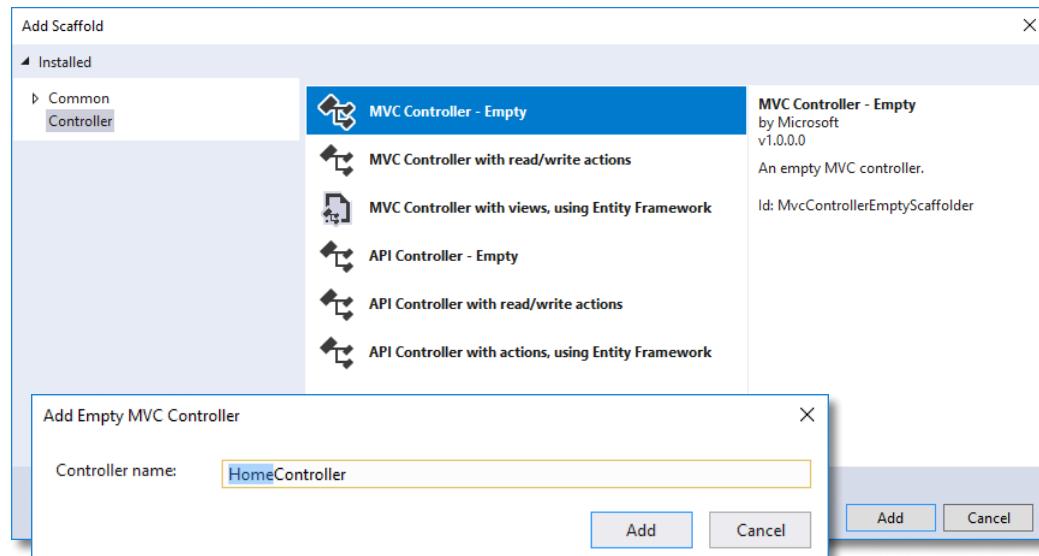
If a method of a controller runs in response to HTTP action verbs such as GET or POST, the method is known as an *action method*, or an *action*. In this figure, for example, the Index() method is an action because it runs in response to an HTTP GET or POST request. You'll learn more about how this works later.

In this figure, the Index() action begins by setting two properties of the ViewBag property that's automatically available to controllers and views. To do that, the first statement sets the Name property of the ViewBag to a string value of "Mary". Then, the second statement sets the FV property to a decimal value of 99999.99. This works because the ViewBag property uses dynamic properties to get and set values. As a result, you can dynamically create a property by specifying any property name that you want.

After the Index() action has stored some data in the ViewBag, it uses the View() method to return a ViewResult object for the view associated with the action method. For the Index() action of the Home controller, this returns a ViewResult object for the view in the Views/Home/Index.cshtml file like the one shown in the next figure. This works because a ViewResult object is a type of IActionResult object. As a result, the Index() method can return a ViewResult object.

Because specifying the IActionResult interface as the return type for an action method allows you to return any type of action result, it provides a flexible way to code an action method. Then, if you later decide to return a different type of action result, you can do that. However, if you know that you are definitely going to return a ViewResult object, you can change the return type of the method to ViewResult. Some programmers think this makes your code easier to read.

## The dialogs for adding a controller



## How to add a file for a controller

1. Right-click the Controllers folder and select Add→Controller.
2. In the Add Scaffold dialog, select “MVC Controller – Empty” and click Add.
3. In the Add Empty MVC Controller dialog, name the controller and click Add.

## The HomeController.cs file

```
using Microsoft.AspNetCore.Mvc;

namespace FutureValue.Controllers
{
    public class HomeController : Controller
    {
        public IActionResult Index()
        {
            ViewBag.Name = "Mary";
            ViewBag.FV = 99999.99;
            return View();
        }
    }
}
```

## Description

- A method of a *controller* that runs in response to HTTP action verbs such as GET or POST is known as an *action method*, or an *action*.
- The ViewBag property is automatically available to controllers and views. It uses dynamic properties to get and set values.
- The View() method returns a ViewResult object for the view associated with an action method.

---

Figure 2-4 How to add a controller

## How to add a Razor view

---

The procedure in figure 2-5 shows how to add a Razor view to a web app. In addition, it shows the code for the view after it has been edited so it's a good starting point for the Future Value app presented in this chapter.

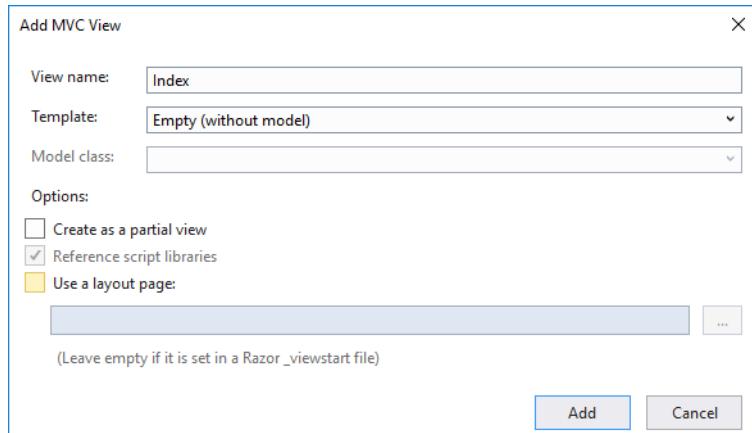
A *Razor view* contains both C# and HTML code. That's why its file extension is .cshtml. In ASP.NET Core MVC, the *Razor view engine* uses server-side code to embed C# code within the HTML. Razor code is preceded by the @ sign.

To execute one or more C# statements, you can declare a *Razor code block* by coding the @ sign followed by a pair of curly braces ({}). In this figure, for example, the Index.cshtml file begins with a code block that contains a single C# statement. This statement sets the Layout property that's available to all views to null. This indicates that the view doesn't have a Razor layout as described later in this chapter.

To evaluate a C# expression and display its result, you can code a *Razor expression* by coding the @ sign and then coding the expression. In this figure, for example, the view uses Razor expressions to access the ViewBag property that's available to all views and display the two properties that were set by the controller in the previous figure. Here, the first expression just displays the Name property. However, the second expression gets the FV property and calls the ToString() method to convert the decimal value to a string that uses the currency format with 2 decimal places. To do that, this code supplies a format specifier of "C2".

Besides the Razor code, the rest of the code for this view consists of simple HTML elements. As a result, if you have some experience with HTML, you shouldn't have any trouble understanding this page. If you don't understand the HTML on this page, you need to learn basic HTML skills like the ones presented in the first eight chapters of *Murach's HTML5 and CSS3*.

## The dialog for adding a Razor view



### How to add a view to the Views/Home folder

1. In the Solution Explorer, right-click the Views/Home folder and select Add→View.
2. In the resulting dialog, enter Index as the name of the view.
3. If necessary, select the “Empty (without model)” template.
4. Deselect the “Use a layout page” checkbox.
5. Click the Add button.

### The Home/Index.cshtml view

```

@{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Home Page</title>
</head>
<body>
    <h1>Future Value Calculator</h1>
    <p>Customer Name: @ ViewBag.Name</p>
    <p>Future Value: @ ViewBag.FV.ToString("C2")</p>
</body>
</html>

```

### Description

- A *Razor view* contains both C# code and HTML. That's why its file extension is .cshtml.
- In ASP.NET Core MVC, the *Razor view engine* uses server-side code to embed C# code within HTML elements.
- To execute one or more C# statements, you can declare a *Razor code block* by coding the @ sign followed by a pair of curly braces ({}).
- To evaluate a C# expression and display its result, you can code a *Razor expression* by coding the @ sign and then coding the expression.

---

Figure 2-5 How to add a Razor view

## How to configure an MVC web app

---

Figure 2-6 shows how to configure a simple MVC web app like the Future Value app presented in this chapter. To do that, you can edit the Startup.cs file so it configures the middleware for the HTTP request pipeline correctly. If you’re starting from the MVC template, you can do that by deleting all the extra statements that aren’t necessary for a simple app like the Future Value app. Or, if you’re starting from the Empty template, you can do that by adding the statements shown in this figure.

Within the ConfigureServices() method, you need to make sure to include a statement that calls the AddControllersWithViews() method. This enables the services required by the controllers and Razor views of an MVC app. However, if your code contains a statement that calls the AddRazorPages() method, you can delete that statement. That’s because Razor pages are typically used by apps that don’t use the MVC pattern. As a result, they aren’t presented in this book.

Within the Configure() method, you typically want to include all of the statements shown in this figure. These statements configure the services that your app is using.

To start, this code checks whether the web hosting environment is a development environment. If so, the middleware handles exceptions by displaying a web page that’s designed for developers, not end users. That’s typically what you want when you’re in a development environment as described throughout this book.

However, if you deploy the app to a production environment, the middleware handles exceptions by displaying a page that the developer customizes for end users. That’s typically what you want for a production environment. In addition, this code calls the UseHsts() method to configure the middleware to send HTTP Strict Transport Security (HSTS) headers to clients, which is a recommended practice for production apps.

After the if statement, the next four statements configure the middleware components that are the same for development and production environments. Of these statements, it’s important to note that the UseEndpoints() method sets the default controller for the app to the Home controller, and it sets the default action to the Index() action. As a result, when the app starts, it calls the Index() action method of the Home controller. This displays the Index view, which is usually what you want.

## The Startup.cs file after it has been edited

```
using Microsoft.AspNetCore.Builder;
using Microsoft.AspNetCore.Hosting;
using Microsoft.Extensions.DependencyInjection;
using Microsoft.Extensions.Hosting;

namespace FutureValue
{
    public class Startup
    {
        // Use this method to add services to the container.
        public void ConfigureServices(IServiceCollection services)
        {
            services.AddControllersWithViews();
        }

        // Use this method to configure the HTTP request pipeline.
        public void Configure(IApplicationBuilder app,
                             IWebHostEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
            else
            {
                app.UseExceptionHandler("/Home/Error");
                app.UseHsts();
            }

            app.UseHttpsRedirection();
            app.UseStaticFiles();

            app.UseRouting();
            app.UseEndpoints(endpoints =>
            {
                endpoints.MapControllerRoute(
                    name: "default",
                    pattern: "{controller=Home}/{action=Index}/{id?}");
            });
        }
    }
}
```

## Description

- The Startup.cs file contains the code that configures the middleware for the HTTP request pipeline.
- The Configure() method begins by checking whether the web hosting environment is a development environment. If so, it configures the middleware for a development environment. Otherwise, it configurations the middleware for a production environment.
- The UseEndpoints() method in this figure sets the default controller for the app to the Home controller, and it sets the default action to the Index() action. As a result, when the app starts, it calls the Index() action method of the Home controller.

---

Figure 2-6 How to configure an MVC web app

## How to run a web app and fix errors

After you write the C# and HTML code shown in the previous figures, you need to test it to be sure it works properly. If it does, the middleware for the HTTP request pipeline is configured correctly, the controller is setting some data, and the view is displaying that data. However, if you encounter any errors, you need to fix them and test the app again. For now, you can do that with the basic skills presented in the next two figures. Later, in chapter 5, you'll learn more skills for testing and debugging.

### How to run a web app

To run a web app, you can use one of the techniques presented in figure 2-7. For example, you can press Ctrl+F5 to run the web app without the debugger. Then, you can stop the app by clicking the close button in the browser's upper right corner.

However, you can also run the web app with the debugger by pressing F5. When you do that, you can use Visual Studio's debugger as described in chapter 5. Then, you can stop the app by clicking the Stop Debugging button in the Debug toolbar.

When you run an app, you need to decide whether to run it on the older Windows-only IIS Express server or the newer cross-platform Kestrel server. Since the Kestrel server runs faster than the IIS Express server, it's excellent for getting started with ASP.NET Core development. As a result, we recommend that you use it with this book.

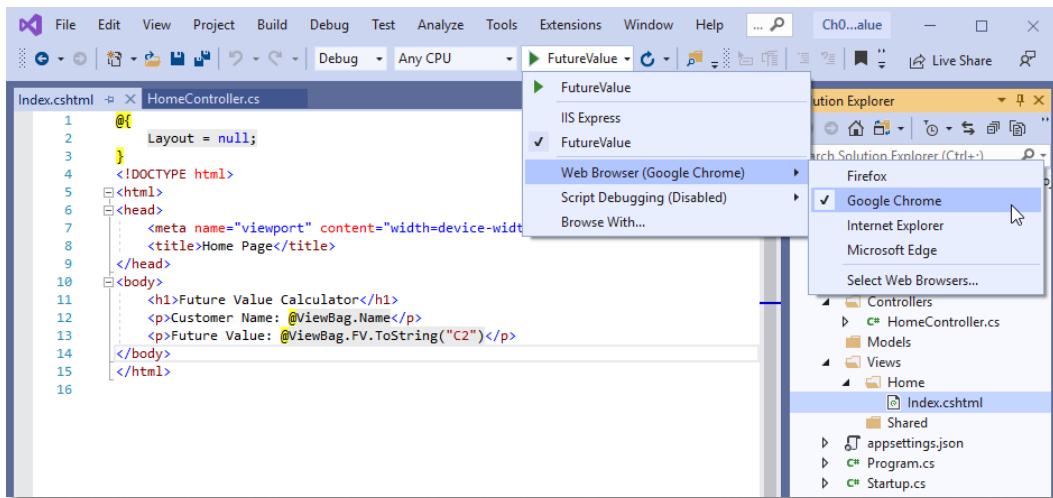
To use the Kestrel server, click the drop-down list to the right of the Start button in the toolbar and select the item for the project's name. In other words, don't select the IIS Express item that's usually selected by default. If you select the Kestrel server and run the app, Visual Studio starts Kestrel and uses a console window to display information about the status of each HTTP request. To stop the server, you can close this window.

In addition to testing whether the web app runs correctly, you should also check whether it displays correctly in different browsers. Visual Studio makes it easy to change the default browsers for this purpose by providing a drop-down browser list. After you use that list to change the default browser, you can run the web app again to test it in that browser.

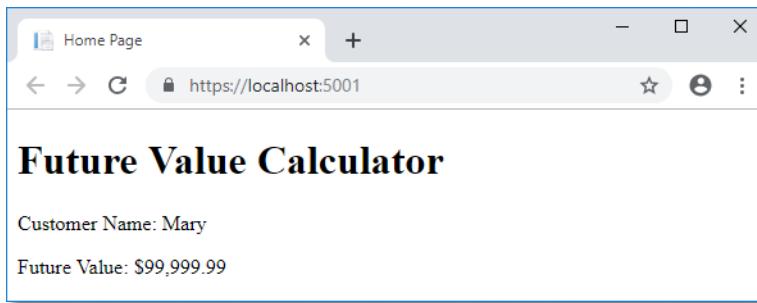
Before Visual Studio runs an app, it builds the project by compiling the necessary code. Then, if the code compiles without errors, Visual Studio runs the app and displays the starting page in your default browser. At that point, you can test the app to make sure that it works correctly. For now, the app is working correctly if it displays a web page that looks like the one shown in this figure that displays a customer name of "Mary" and a future value of \$99,999.99.

The first time you run a web app, you may get a series of dialogs with security warnings that indicate that you are about to install an SSL certificate. ASP.NET Core needs this certificate to configure a development environment, so you can click Yes to install this certificate. Then, if a web page is displayed indicating that it may not be safe to proceed, you can click the link or button to proceed.

## The Start button drop-down list in Visual Studio



## The Future Value app in the Chrome browser



### Description

- To run an app in the default browser, press **Ctrl+F5**. This starts the app without debugging.
- To stop an app, click the close button in the browser's upper right corner.
- To change the default browser for the app, display the drop-down list for the Start button, select the Web Browser item, and select the default web browser from the list.
- By default, Visual Studio uses the IIS Express web server. To change the web server to the Kestrel server, display the drop-down list for the Start button and select the project's name.
- When Visual Studio runs the app on the Kestrel server, it uses a console window to display information about the server. To stop the server, you can close the command line window.
- If you press **F5** or click the Start button in the toolbar, Visual Studio starts the app with debugging. This is another way to run an app that's especially useful if you need to debug an app as described in chapter 5. Then, to stop the app, you can click the Stop Debugging button in the Debug toolbar.

Figure 2-7 How to run a web app

## How to find and fix errors

---

If any errors are detected as part of the compilation, Visual Studio opens the Error List window and displays the errors as shown in figure 2-8. These errors can consist of *syntax errors* that have to be corrected before the app can be compiled, as well as warning messages. In this figure, just one error message and no warning messages are displayed.

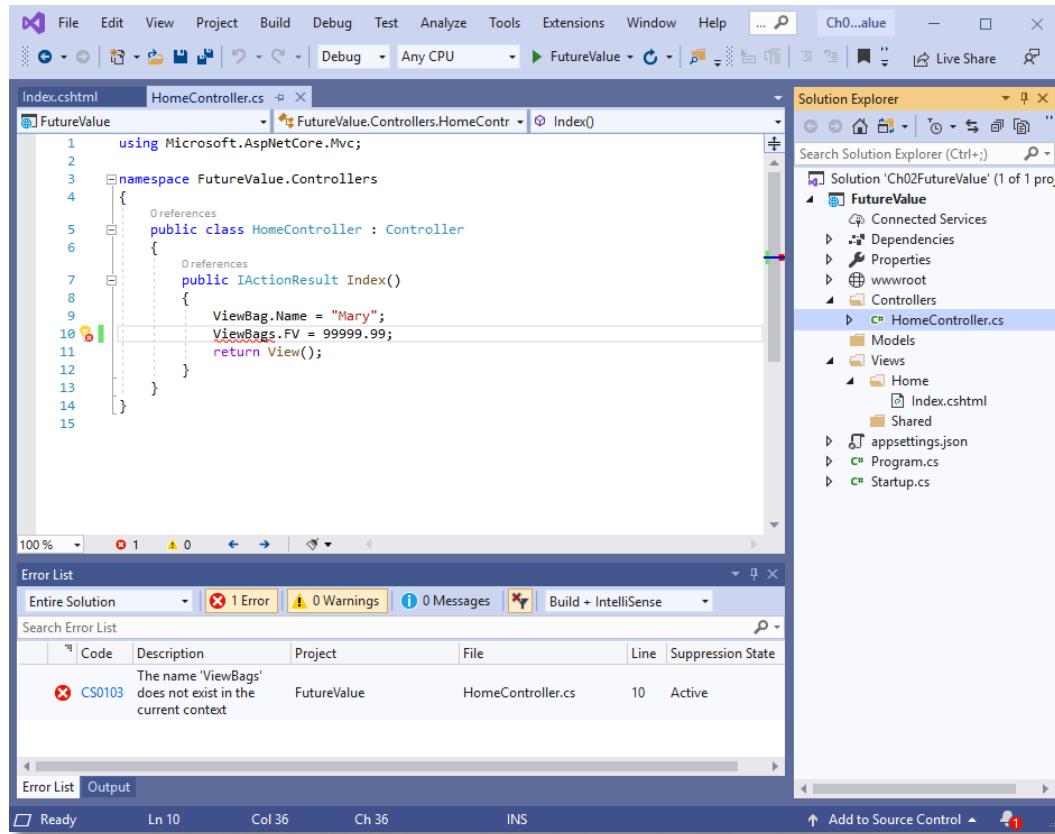
To fix an error, you can double-click it in the Error List window. This moves the cursor into the code editor and to the line of code that caused the error. By moving from the Error List window to the code editor for all of the messages, you should be able to find the coding problems and fix them. In this figure, the error message accurately indicates that the name ViewBags doesn't exist. That's because the name of the property that's available to controllers and views is ViewBag, not ViewBags.

Keep in mind, though, that the error message might not be accurate, and its link might not jump to the line of code that's causing the problem. For example, it's common to need to fix a related statement such as a statement that declares a variable. Still, the error message and the line of code that it links to should help you find the statement that's causing the problem.

After you fix all of the compilation errors and run the app, you may still encounter an *exception*. That happens when ASP.NET Core can't execute one of the compiled C# statements correctly at runtime. Then, if you're running the app without debugging, ASP.NET Core MVC displays a description of the exception in the web browser. At that point, you can stop the app. Then, you can fix the problem and test again.

Alternately, if you're running the app with debugging, ASP.NET Core MVC switches to the code editor and highlights the statement that caused the exception. At that point, you can stop the app by clicking on the Stop Debugging button in the Debug toolbar. Then, you can fix the problem and test again.

## Visual Studio with the Error List window displayed



### Description

- If a *syntax error* is detected when you attempt to build and run an app, a dialog asks whether you want to continue by running the last successful build. If you click No, the app isn't run and an Error List window is displayed.
- The Error List window provides information about the errors in your app.
- To go to the statement that caused a syntax error, double-click the error in the Error List window. This should help you find the cause of the error.
- If a compiled statement can't be executed when you run a web app, an *exception* occurs. Then, you can use the information that's displayed in the browser to attempt to fix this exception, or you can debug the exception as described in chapter 5.

Figure 2-8 How to find and fix errors

## How to work with a model

---

Once you’re sure that the controller and view are working correctly, you’re ready to add a model to your app. Then, you can modify the controller and view to work with this model. When you’re done, the app should get data from the user, store that data in the model, use the model to perform a calculation, and display the result of the calculation. Along the way, you’ll learn a lot about how an ASP.NET Core MVC app works.

### How to add a model

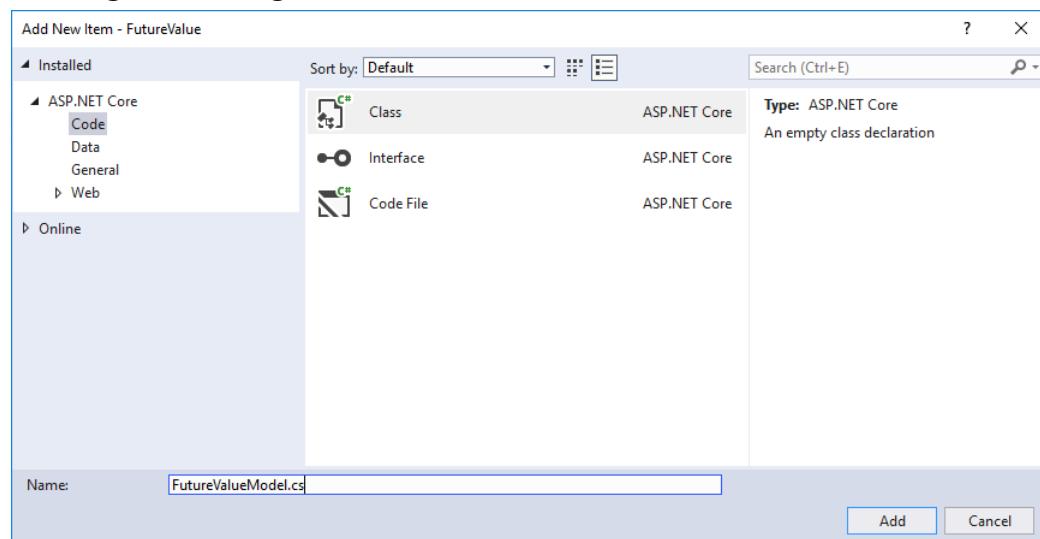
---

A *model* is a regular C# class that stores a model of the data for a page and is typically stored in the Models folder. As a result, to add a model to your app, you just need to add a C# class to the Models folder as shown in figure 2-9. In this figure, the model class has a name of FutureValueModel. Here, the “Model” suffix is optional.

To keep the name of the model short, some programmers would prefer to drop the “Model” suffix and give the model a name of FutureValue. However, a model can’t have the same name as a namespace, and this particular model is stored in the FutureValue namespace. As a result, this class uses the “Model” suffix to create a name for the model that doesn’t conflict with the name of the namespace.

The model shown in this figure is a standard C# class. It provides three properties that can be used to get and set the monthly investment, yearly interest rate, and number of years for a future value. In addition, it provides a method named CalculateFutureValue() that calculates and returns the future value for the specified properties. To do that, this method converts the yearly values to monthly values and uses a loop to calculate the future value.

## The dialog for adding a class



### How to add a file for a model class

1. In the Solution Explorer, right-click the Models folder and select Add→Class.
2. In the resulting dialog, enter the name of the class, and click the Add button.

### The FutureValueModel class with three properties and a method

```
namespace FutureValue.Models
{
    public class FutureValueModel
    {
        public decimal MonthlyInvestment { get; set; }
        public decimal YearlyInterestRate { get; set; }
        public int Years { get; set; }
        public decimal CalculateFutureValue()
        {
            int months = Years * 12;
            decimal monthlyInterestRate = YearlyInterestRate / 12 / 100;
            decimal futureValue = 0;
            for (int i = 0; i < months; i++)
            {
                futureValue = (futureValue + MonthlyInvestment) *
                    (1 + monthlyInterestRate);
            }
            return futureValue;
        }
    }
}
```

### Description

- A *model* is a regular C# class that models the data for the app. The class for a model is typically stored in the Models folder.
- A model can't have the same name as the namespace.

---

Figure 2-9 How to add a model

## How to add a Razor view imports page

---

A *Razor view imports page* makes it easier to work with models and tag helpers. As a result, most web apps include this page.

The procedure in figure 2-10 shows how to add a Razor view imports page to your web app. This adds a file named \_ViewImports.cshtml to your app that contains Razor directives that are applied to all views in your app.

To give you an idea of how a Razor view imports page works, this figure shows the code for the Razor view imports page of the Future Value app. Here, the first line imports the namespace for your model classes. That way, you can use classes from that namespace in your views without fully qualifying those classes. Then, the second line enables all tag helpers that are available from the ASP.NET Core MVC framework. That way, you can use these tag helpers in your views.

If you don't import the namespace for a model class, you can still use the model in your views. However, you'll need to fully qualify its name like this:

```
@model FutureValue.Models.FutureValueModel
```

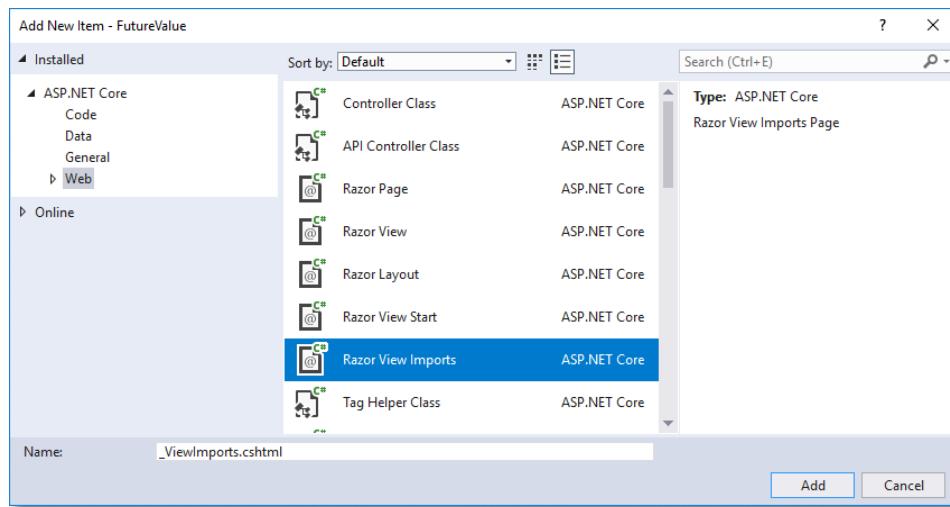
As a result, you typically want to include a Razor view imports page that imports the model. That way, you can specify the name of the model like this:

```
@model FutureValueModel
```

The next figure shows how this works.

If you don't enable the tag helpers, you can still use them in your views. However, you need to add a @tagTagHelper directive to the top of each view that uses tag helpers. As a result, it typically makes sense to include this directive in a Razor view imports page. That way, you don't have to specify this directive for each view.

## The dialog for adding a Razor view imports page



### How to add a Razor view imports page

1. In the Solution Explorer, right-click the Views folder and select Add→New Item.
2. In the resulting dialog, select the Installed→ASP.NET Core→Web category, select the Razor View Imports item, and click the Add button.

### The Views/\_ViewImports.cshtml file for the Future Value app

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

### A Razor view imports page makes it easier to work with...

- Model classes.
- Tag helpers.

### Description

- Most apps include a *Razor view imports page* that makes it easier to work with your model classes and the tag helpers that are available from ASP.NET Core MVC.

---

Figure 2-10 How to add a Razor view imports page

## How to code a strongly-typed view

---

You use the @model directive to *bind* the model to the view. This kind of view is called a *strongly-typed view*. In figure 2-11, for example, the @model directive at the top of the view binds the view to the model class named FutureValueModel.

After binding the model to the view, this view uses the asp-for *tag helper* to *bind* HTML elements to the corresponding properties of the object. In particular, this tag helper binds the MonthlyInvestment, YearlyInterestRate, and Years properties to corresponding <label> and <input> elements in the view. As a result, when the user enters values into the <input> elements and clicks the Calculate button, ASP.NET Core MVC automatically updates the model with the values entered by the user. Then, the controller can access those values as shown in the next figure.

The asp-for tag helper automatically generates attributes for these HTML elements. For example, it generates the name and id attributes that MVC needs to be able to access these HTML elements. It also generates a type attribute that indicates the type of field to display.

The asp-action tag helper also generates an attribute. In particular, it generates the action attribute for the <form> element. Instead of using this tag helper, you could just specify an href attribute like this:

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

However, using the asp-action tag helper makes your code more flexible and easier to maintain.

In this figure, the form only uses the asp-action tag helper to specify the action. As a result, MVC uses the Index() action method of the current controller, which is the Home controller. However, if you want to call an action from another controller, you can use the asp-controller tag helper to specify the name of that controller. As you progress through this book, you'll see plenty of examples of that.

The code in this figure uses the asp-for tag helper to access the properties of the model object. Since this tag helper is designed to bind a model object to HTML elements, you can access the properties of the model object just by specifying their names.

However, if you want to access a property of the model object outside of an asp-for tag helper, you must start by coding the @Model property (not the @model directive) to access the model object. Then, you access any property or method from that object. For example, you can use the @Model property to access the MonthlyInvestment property of the FutureValueModel model object like this:

```
<div>@Model.MonthlyInvestment</div>
```

Before you go on to the next figure, note that this view includes a <style> element within its <head> element. To save space, this <style> element just contains a comment that indicates that it includes all of the same CSS styles shown in figure 2-14. These styles apply some basic formatting to the <body>, <h1>, <label>, and <div> elements so this page appears as shown later in this chapter.

## Common tag helpers for forms

Tag helper	HTML tags	Description
asp-for	<label> <input>	Binds the HTML element to the specified model property.
asp-action	<form> <a>	Specifies the action for the URL. If no controller is specified, MVC uses the current controller.
asp-controller	<form> <a>	Specifies the controller for the URL.

## A strongly-typed Index view with tag helpers

```
@model FutureValueModel
@{
    Layout = null;
}
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>Future Value Calculator</title>
    <style>
        /* all of the CSS styles from figure 2-14 go here */
    </style>
</head>
<body>
    <h1>Future Value Calculator</h1>
    <form asp-action="Index" method="post">
        <div>
            <label asp-for="MonthlyInvestment">Monthly Investment:</label>
            <input asp-for="MonthlyInvestment" />
        </div>
        <div>
            <label asp-for="YearlyInterestRate">Yearly Interest Rate:</label>
            <input asp-for="YearlyInterestRate" />
        </div>
        <div>
            <label asp-for="Years">Number of Years:</label>
            <input asp-for="Years" />
        </div>
        <div>
            <label>Future Value:</label>
            <input value="@ViewBag.FV.ToString("C2")" readonly>
        </div>
        <button type="submit">Calculate</button>
        <a asp-action="Index">Clear</a>
    </form>
</body>
</html>
```

### Description

- You use the @model directive to *bind* the model to the view. This kind of view is called a *strongly-typed* view.
- ASP.NET Core MVC *tag helpers* are used to automatically generate *attributes* for some HTML elements. They are also used to *bind* HTML elements to the properties of the object that's the *model* for the view.

Figure 2-11 How to code a strongly-typed view

## How to handle GET and POST requests

---

Figure 2-12 begins by showing how to use the `HttpGet` and `HttpPost` attributes to create one `Index()` method that handles an HTTP GET request and another `Index()` method that handles an HTTP POST request. This is a common pattern in web development.

For example, it's common for a GET request to display a blank input form to the user. That happens by default when an ASP.NET Core MVC app starts, and it happens when a link like the Clear link on the Future Value form is clicked. Then, when the user clicks the submit button, the app sends a POST request to the same URL to process the data entered by the user. If you look back at figure 2-11, you'll see that the `method` attribute of the `<form>` element determines the type of request that's sent. In this case, it's a POST request.

In MVC, you can use overloaded action methods to handle both GET and POST requests for a page. In this figure, for example, the first `Index()` method doesn't accept any arguments. However, the second `Index()` method accepts a `FutureValueModel` object as an argument. Since each `Index()` method has a unique signature, you can use HTTP attributes to specify the HTTP verb for each method.

If you don't provide a unique signature for each version of the action method, you'll get a compiler error. For example, what if both versions of the action method need to specify the model as a parameter? In that case, you can solve the issue by specifying a dummy parameter like this:

```
public IActionResult Index(FutureValueModel model,  
    string dummy)
```

Here, the second argument isn't used by the `Index()` method. However, it provides a unique signature for the method.

## How to work with a strongly-typed view

---

When an action method handles a POST request from a strongly-typed view, MVC uses the data stored in the POST request to set the properties of the model object. In this figure, for example, MVC automatically sets the properties of the model object that's passed to the POST version of the `Index()` method.

As a result, the action method can use the model object to work with the posted data. In this figure, the code just calls the `CalculateFutureValue()` method from the model to get the result of the future value calculation. However, this shows that the other three properties of the model were set automatically, which is what you want.

In addition, the POST version of the `Index()` method can use the `View()` method to pass the model on to the view. In this figure, that's what the second statement does. That way, the strongly-typed view shown in the previous figure can display the correct values for the properties of the model.

## Two attributes that indicate the HTTP verb an action method handles

Attribute	Description
<code>HttpGet</code>	Specifies that the action method handles a GET request.
<code>HttpPost</code>	Specifies that the action method handles a POST request.

## Two methods you can use to return a view from a controller

Method	Description
<code>View()</code>	Returns the view that corresponds to the current controller and action.
<code>View(model)</code>	Passes the specified model to the view that corresponds to the current controller and action so the view can bind to the model.

## An overloaded Index() action method that handles GET and POST requests

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

public class HomeController : Controller
{
    [HttpGet]
    public IActionResult Index()
    {
        ViewBag.FV = 0;
        return View();
    }

    [HttpPost]
    public IActionResult Index(FutureValueModel model)
    {
        ViewBag.FV = model.CalculateFutureValue();
        return View(model);
    }
}
```

### Description

- A common pattern in web development is for the same URL to handle HTTP GET and POST requests. In particular, it's common to use a GET request for a URL to display a blank input form to the user. Then, a POST request for the same URL can process the data that's submitted when the user fills out the form and submits it.
- In MVC, you can use overloaded action methods to handle both GET and POST requests for a page. When you do, you use HTTP attributes to indicate which action method handles which request.
- When an action method handles a POST request from a strongly-typed view, MVC uses the data stored in the POST request to set the properties of the model object. Then, the action method can use the model object to work with the posted data, and it can use the `View()` method to pass the model on to the view.

---

Figure 2-12 How to handle GET and POST requests

## The Future Value app after handling GET and POST requests

---

Figure 2-13 shows the Future Value app that has been presented so far in this chapter after it has handled GET and POST requests.

When this app starts, it sends a GET request to the Index() action of the Home controller. As a result, the app displays a screen like the first one shown in this figure. This page doesn't contain values for the first three fields, and it displays a value of \$0.00 for the fourth field, which is a read-only field.

When the user enters data in the form and clicks the Calculate button, the app sends a POST request to the Index() action of the Home controller. As a result, the app calculates the future value and displays it in the fourth text field as shown by the second screen in this figure.

At this point, the user can edit the values and click the Calculate button again to calculate and display a different future value. Or, the user can click the Clear link. Then, the app sends a GET request to the Index() action of the Home controller. Since this GET request doesn't include a model, it clears the form as shown by the first screen.

### The Future Value app after a GET request

A screenshot of a web browser window titled "Future Value Calculator". The address bar shows "https://localhost:5001". The page content is titled "Future Value Calculator". It contains four input fields: "Monthly Investment" (empty), "Yearly Interest Rate" (empty), "Number of Years" (empty), and "Future Value" (\$0.00). Below the inputs are two buttons: "Calculate" and "Clear".

### The Future Value app after a POST request

A screenshot of a web browser window titled "Future Value Calculator". The address bar shows "https://localhost:5001". The page content is titled "Future Value Calculator". The input fields now contain values: "Monthly Investment" (100), "Yearly Interest Rate" (3), "Number of Years" (3), and "Future Value" (\$3,771.46). Below the inputs are two buttons: "Calculate" and "Clear".

### Description

- When the Future Value app starts, it sends a GET request to the Index() action of the Home controller.
- When the user clicks the Clear link, the app sends a GET request to the Index() action of the Home controller.
- When the user clicks the Calculate button, the app sends a POST request to the Index() action of the Home controller. If the user has filled out the form correctly, this automatically sets the three properties of the model object.

Figure 2-13 The Future Value app after handling GET and POST requests

## How to organize the files for a view

So far, the view for the Future Value app consists of a single view file. That's adequate for a single-page web app like the Future Value app presented in this chapter. However, most web apps consist of multiple pages. When that's the case, it makes sense to split the view for a web app into multiple files. That way, you can store the HTML elements and CSS styles that are common to multiple pages in their own files. Then, you can use the common HTML elements and CSS styles in other pages as shown in the next three figures.

If it's adequate to store the view for a single-page app in a single file, why does this chapter show how to split the view for the Future Value app into multiple files? Well, in the real world, multi-page apps are more common than single-page apps. Even for a simple app like this Future Value app, you might want to add pages such as an About page or a Contact Us page. As a result, it often makes sense to set up your app to support multiple pages, even if it's currently a single-page app.

## How to add a CSS style sheet

Figure 2-14 shows how to add a file for a *CSS style sheet*. This provides a way to store the formatting for multiple web pages in a single external file.

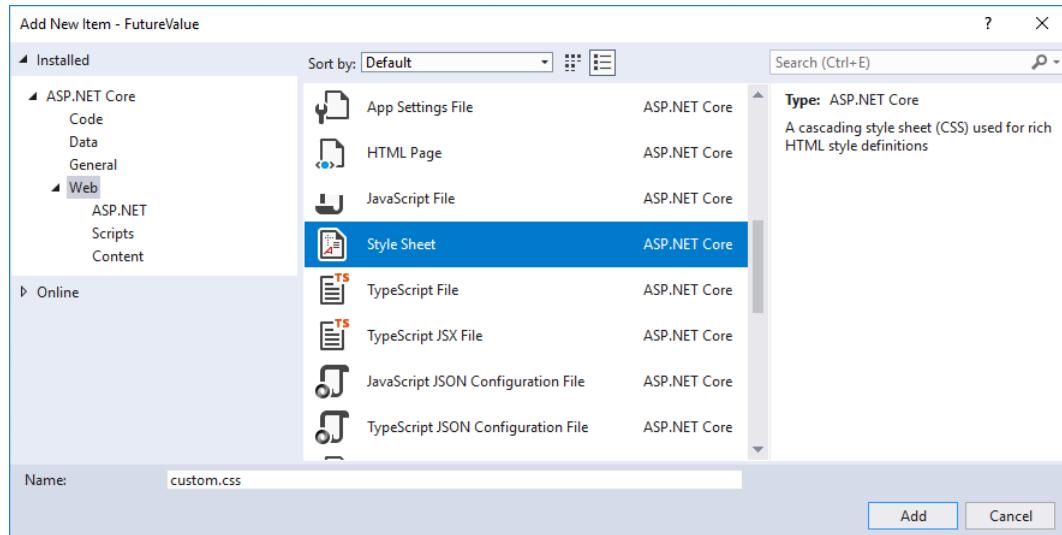
When you add a style sheet to a project, you typically add it to the css folder of the wwwroot folder. In this figure, for example, a style sheet named custom.css is added to the wwwroot/css folder. If your project is based on the MVC template, this folder should already exist. However, if it doesn't exist, you need to create it.

This figure also shows the styles that are stored in the custom.css file. These styles format the Future Value app so it looks the way it does in the previous figure. If you're familiar with CSS, you shouldn't have any trouble understanding this code.

The four style rules in this figure select elements by type. These are referred to as type selectors. To code a type selector, you just code the name of the element. As a result, the first style rule in this group selects the <body> element, the second selects all <h1> elements, the third selects all <label> elements, and the fourth selects all <div> elements.

Each style rule also includes one or more declarations enclosed in braces that specify the formatting for the selected element. In this figure, the declarations for the <body> element set the font family for all elements nested within that element and set the padding for that element. The declarations for the <h1> element set the top margin and color for all <h1> elements on the page. The declarations for the <label> element cause it to be displayed on the same line as the following element and set the width and padding for all <label> elements on the page. And the declaration for the <div> element sets the bottom margin for all <div> elements on the page.

## The dialog for adding a CSS style sheet



### How to add a CSS style sheet

1. If the wwwroot/css folder doesn't exist, create it.
2. Right-click the wwwroot/css folder and select Add→New Item.
3. Select the ASP.NET Core→Web category, select the Style Sheet item, enter a name for the CSS file, and click the Add button.

### The custom.css file for the Future Value app

```
body {
    padding: 1em;
    font-family: Arial, Helvetica, sans-serif;
}

h1 {
    margin-top: 0;
    color: navy;
}

label {
    display: inline-block;
    width: 10em;
    padding-right: 1em;
}

div {
    margin-bottom: .5em;
}
```

### Description

- A *CSS style sheet* provides a way to store the formatting for multiple web pages in a single external file.

---

Figure 2-14 How to add a CSS style sheet

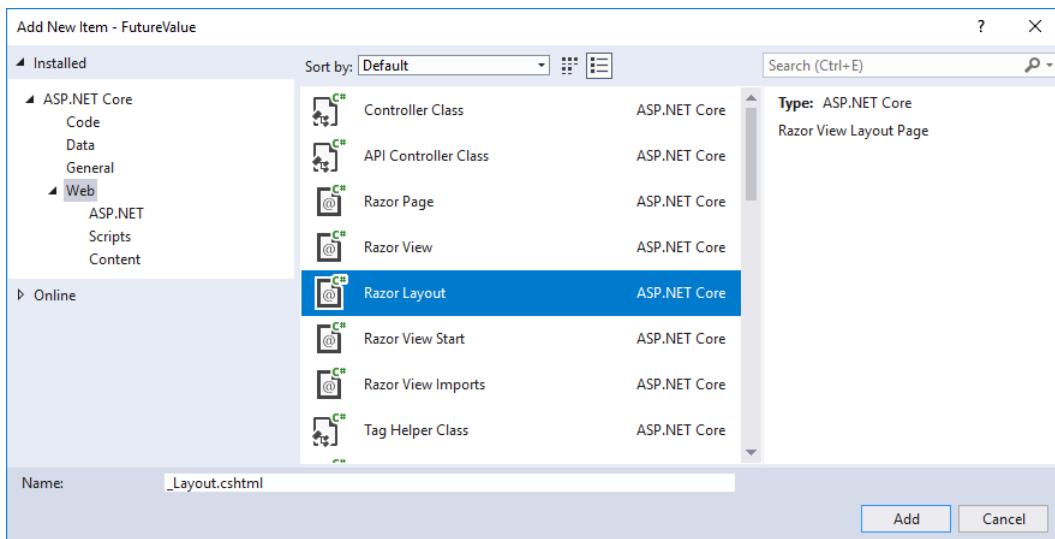
## How to add a Razor layout, view start, and view

---

When you create a multi-page web app, it's common to have headers, footers, and navigation bars that are displayed on all or most pages of a web app. In other words, it's common to have HTML elements that are common to all pages. In that case, it's a good practice to store the elements that are common to multiple pages in separate files. This allows you to keep a consistent look across all pages, and it makes your app easier to maintain.

To store elements that are common to multiple pages in a separate file, you can add a *Razor layout* to your web app as described in figure 2-15. A *Razor layout* provides a way to store elements that are common to multiple web pages in a single file. Then, it usually makes sense to add a *Razor view start* to specify the default layout for the views of your web app. Finally, you can add a *Razor view* to provide a way to store elements that are unique to a web page. If necessary, you can override the default layout for any views as described in the next figure.

## The dialog for adding a Razor layout, view start, or view



### How to add a Razor layout

1. Right-click the Views/Shared folder and select AddNew Item.
2. Select the ASP.NET Core→Web category, select the Razor Layout item, and click the Add button.

### How to add a Razor view start

1. Right-click the Views folder (not the Views/Shared folder) and select Add→New Item.
2. Select the ASP.NET Core→Web category, select the Razor View Start item, and click the Add button.

### How to add a Razor view

1. Right-click the folder for the view (Views/Home, for example) and select Add→View.
2. Use the dialog from figure 2-5 to specify the name for the view.
3. If you're using a layout that has a view start, select the “Use a layout page” item but don't specify a name for the layout page.

### Description

- A *Razor layout* provides a way to store elements that are common to multiple web pages in a single file.
- A *Razor view start* lets you specify the default Razor layout for the Razor views of a web app.
- A *Razor view* provides a way to store elements that are unique to a web page.

Figure 2-15 How to add a Razor layout, view start, and view

## The code for a Razor layout, view start, and view

Figure 2-16 shows the code for a Razor layout, view start, and view for the Future Value app. If you study this code, you should see how it all fits together to form the same strongly-typed view as the one presented earlier in this chapter. The only differences are that it has been split up into the three Razor files shown in this figure and it uses the external CSS file shown earlier in this chapter.

The code for the layout is stored in a Razor file named `_Layout`. This code stores the HTML elements that are common to all pages such as the `<html>`, `<head>`, and `<body>` elements. In addition, it uses Razor code to do some processing. For example, it uses Razor code to get the title for the page from the `Title` property of the `ViewBag` property that's automatically available to all layouts and views. In addition, it uses Razor code to call the `RenderBody()` method that's available to all layouts. This inserts the code from any view file that uses this layout.

The code for the layout also uses a `<link>` element to link to the CSS style sheet shown earlier in this chapter. To do that, it specifies a `rel` attribute of “stylesheet” and an `href` attribute of “`~/css/custom.css`”. As a result, all of the pages of the web app use the styles from that style sheet.

The code for the view start is stored in a Razor file named `_ViewStart`. This code defines a block of C# statements that are executed before the view is rendered. In this example, the block contains a single statement that sets the `Layout` property to “`_Layout`”. In other words, it sets the default layout for all views in the app to the `_Layout` view shown in the first example.

The code for the view is stored in a Razor file named `Index`. This code works much like the strongly-typed view presented earlier in this chapter. The main difference is that it uses a Razor code block to set the title for the page. To do that, it sets the `Title` property of the `ViewBag` object to “Future Value Calculator”. As a result, the layout for the page can get this property and display it as the title of the page.

In general, it's considered a good practice to use a view start to set the default layout for all the views in your app. However, if necessary, you can use the `Layout` property of a view to override the default layout. To do that, you can add a statement below the statement that sets the title for the page like this:

```
@{
    ViewBag.Title = "Future Value Calculator";
    Layout = "_LayoutCalculator";
}
```

Here, the page is using a hypothetical Razor layout named `_LayoutCalculator` that's designed especially for all of the calculator pages of the web app.

For now, that's all you need to know about layout pages. Later, in chapter 7, you'll learn more about working with layouts and views.

## The Views/Shared/\_Layout.cshtml file

```
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
    <link rel="stylesheet" href="~/css/custom.css" />
</head>
<body>
    <div>
        @RenderBody()
    </div>
</body>
</html>
```

## The Views/\_ViewStart.cshtml file

```
@{
    Layout = "_Layout";
}
```

## The Views/Home/Index.cshtml file

```
@model FutureValueModel
@{
    ViewBag.Title = "Future Value Calculator";
}
<h1>Future Value Calculator</h1>
<form asp-action="Index" method="post">
    <div>
        <label asp-for="MonthlyInvestment">Monthly Investment:</label>
        <input asp-for="MonthlyInvestment" />
    </div>
    <div>
        <label asp-for="YearlyInterestRate">Yearly Interest Rate:</label>
        <input asp-for="YearlyInterestRate" />
    </div>
    <div>
        <label asp-for="Years">Number of Years:</label>
        <input asp-for="Years" />
    </div>
    <div>
        <label>Future Value:</label>
        <label>@ViewBag.FV.ToString("c2")</label>
    </div>
    <button type="submit">Calculate</button>
    <a asp-action="Index">Clear</a>
</form>
```

## Description

- You can use the Razor file named \_ViewStart to set the default layout for all the views in your app. However, if necessary, you can use the Layout property of a view to override the default layout.

---

Figure 2-16 The code for a Razor layout, view start, and view

## How to validate user input

At this point, the Future Value app works correctly if the user enters valid data. However, if the user enters invalid data and clicks the Calculate button, the app just displays a future value of 0 without displaying any error messages to indicate that the user has entered invalid data. When you code a web app, you typically want to display messages like these if the user enters invalid data. Fortunately, ASP.NET Core MVC makes it easy to validate data and display error messages as shown in the next three figures. This is known as *data validation*, and it's an important part of developing most apps.

### How to set data validation rules in the model

The first step in validating the data that a user enters is to set data *validation rules* in the model as described in figure 2-17. To start, you can import the DataAnnotations namespace. Then, you can use the *validation attributes* from that namespace to set the data validation rules.

The table in this figure describes two of the most common validation attributes. First, you can code the Required attribute above a property to indicate that a value is required for that property. Second, you can code the Range attribute above a property to indicate that the value for that property must be within the specified range of values.

When you code the Required attribute, the data type for the property must be nullable. To make a non-nullable data type nullable, you can code a question mark (?) after the data type as shown in the first example. Then, if the user doesn't enter a value for this property, the MVC framework generates a default error message.

The second example shows that you can code two validation attributes on the same property. In this example, both the Required and Range attributes are coded above the property. Here, the Required attribute works the same as it did in the first example. In addition, the Range attribute specifies a minimum range of 1 and a maximum range of 500. As a result, if the user doesn't enter a value that's within the specified range, the MVC framework generates a default error message.

Although the default error messages generated by the MVC framework are adequate in some cases, it's a good practice to specify user-friendly error messages as shown in the third example. To do that, you can pass an argument named ErrorMessage as the last argument of the attribute. In this example, the Required attribute specifies an error message of "Please enter a monthly investment amount." This is more user-friendly than the default message of "The field MonthlyInvestment is required." Similarly, the Range attribute specifies an error message of "Monthly investment amount must be between 1 and 500." This is more user-friendly than the default message of "The field MonthlyInvestment must be between 1 and 500."

## How to import the DataAnnotations namespace

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

### Two common validation attributes

Attribute	Description
<b>Required</b>	Indicates that a value is required for the property.
<b>Range(min, max)</b>	Indicates that the value for the property must be within a specified range of values.

### A model property with a validation attribute

```
[Required]
public decimal? MonthlyInvestment { get; set; }
```

The default error message if the property isn't set

The field `MonthlyInvestment` is required.

### A model property with two validation attributes

```
[Required]
[Range(1, 500)]
public decimal? MonthlyInvestment { get; set; }
```

The default error message if the property isn't in a valid range

The field `MonthlyInvestment` must be between 1 and 500.

### A model property with user-friendly error messages

```
[Required(ErrorMessage = "Please enter a monthly investment amount.")]
[Range(1, 500, ErrorMessage =
    "Monthly investment amount must be between 1 and 500.")]
public decimal? MonthlyInvestment { get; set; }
```

### Description

- The process of checking data to make sure it's valid is known as *data validation*.
- You can use the *validation attributes* of the DataAnnotations namespace to add *validation rules* to your model.
- For the Required attribute to work properly, the data type for the property must be nullable.
- If you don't specify an error message, the data validation attributes generate a default error message.
- To specify a custom error message, you can pass an argument named `ErrorMessage` as the last argument of the attribute.

---

Figure 2-17 How to set data validation rules in the model

## The model class with data validation

---

Figure 2-18 shows the entire model class after data validation attributes have been added to its three properties. In addition, since these properties now use nullable data types, the return type for the CalculateFutureValue() method must also be nullable. Other than that, you shouldn't have much trouble understanding this model class. All three of the properties specify both the Required and Range attributes. In addition, all three of the properties specify a user-friendly error message.

## The model class with data validation attributes

```
using System.ComponentModel.DataAnnotations;

namespace FutureValue.Models
{
    public class FutureValueModel
    {
        [Required(ErrorMessage = "Please enter a monthly investment.")]
        [Range(1, 500, ErrorMessage =
            "Monthly investment amount must be between 1 and 500.")]
        public decimal? MonthlyInvestment { get; set; }

        [Required(ErrorMessage = "Please enter a yearly interest rate.")]
        [Range(0.1, 10.0, ErrorMessage =
            "Yearly interest rate must be between 0.1 and 10.0.")]
        public decimal? YearlyInterestRate { get; set; }

        [Required(ErrorMessage = "Please enter a number of years.")]
        [Range(1, 50, ErrorMessage =
            "Number of years must be between 1 and 50.")]
        public int? Years { get; set; }

        public decimal? CalculateFutureValue()
        {
            int? months = Years * 12;
            decimal? monthlyInterestRate = YearlyInterestRate / 12 / 100;
            decimal? futureValue = 0;
            for (int i = 0; i < months; i++)
            {
                futureValue = (futureValue + MonthlyInvestment) *
                    (1 + monthlyInterestRate);
            }
            return futureValue;
        }
    }
}
```

---

Figure 2-18 The model class with data validation

## How to check the data validation

---

The first example in figure 2-19 shows the Index() action for a POST request in the Home controller. Here, the code has been modified so it uses the ModelState property that's available from the controller class to check whether the data in the model is valid. If so, this code calculates the future value and sets the FV property of the ViewBag to the result of the calculation. Otherwise, this code sets the FV property of the ViewBag to 0. That way, the view can display the result of the calculation or the error messages, depending on the state of the model. Either way, it passes the model object to the view so the values entered by the user are redisplayed.

## How to display validation error messages

---

The second example shows the <form> element of the Index view. Here, the view includes code that displays a summary of all data validation errors in the model. In particular, within the <form> element, the first <div> element includes a tag helper named asp-validation-summary that specifies a value of "All". As a result, if the user enters valid data, the MVC framework hides this <div> element. However, if the user doesn't enter valid data, the MVC framework displays this <div> element and fills it with a list of all validation error messages that apply to the current model.

## The Future Value app after validating data

---

The third example shows that the Future Value app displays validation error messages above the form when a user enters invalid data. Here, the messages indicate that the monthly investment is required, the yearly interest rate is out of range, and the number of years is out of range.

For now, that's all you need to know about validating data in your web apps. Later, in chapter 11, you'll learn more about data validation.

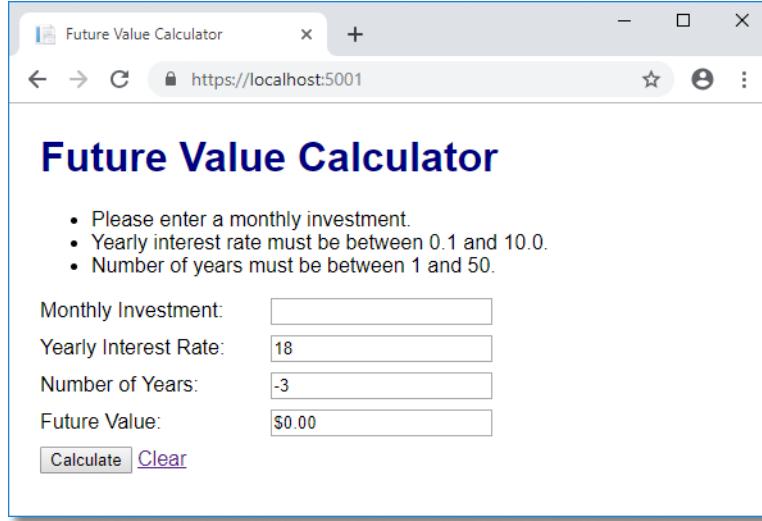
## An action method that checks for invalid data

```
[HttpPost]
public IActionResult Index(FutureValueModel model)
{
    if (ModelState.IsValid)
    {
        ViewBag.FV = model.CalculateFutureValue();
    }
    else
    {
        ViewBag.FV = 0;
    }
    return View(model);
}
```

## A view that displays a summary of validation messages

```
<form asp-action="Index" method="post">
    <div asp-validation-summary="All"></div>
    <div>
        <label asp-for="MonthlyInvestment">Monthly Investment:</label>
        <input asp-for="MonthlyInvestment" />
    </div>
    <!-- rest of input form -->
</form>
```

## The Future Value app with invalid data



## Description

- A controller can use the ModelState property that's available from the controller class to check whether the data in the model is valid.
- A view can use the tag helper named asp-validation-summary to display a summary of all data validation errors in the model.

Figure 2-19 How to check data validation and display error messages

## Perspective

---

The purpose of this chapter has been to teach you the basic skills for creating a one-page ASP.NET Core MVC app with Visual Studio. If you've already used Visual Studio and C# to develop other apps, such as Windows Forms apps, and you have basic HTML and CSS skills, you shouldn't have any trouble mastering these skills.

In the next chapter, you'll learn the basics of using Bootstrap. This open-source library provides CSS and JavaScript classes that make it easy to give your pages a professional appearance. In addition, Bootstrap makes it possible to display your web pages on devices of varying sizes.

## Terms

---

Visual Studio template	model
controller	Razor view imports page
action method	strongly-typed view
action	tag helper
Razor view engine	CSS style sheet
Razor view	Razor layout
Razor code block	Razor view start
Razor expression	data validation
syntax error	validation attributes
exception	validation rules

## Summary

---

- You create a web app from a *Visual Studio template* that determines the folders and files for the project.
- A method of a *controller* class that runs in response to HTTP action verbs such as GET or POST is known as an *action method*, or an *action*.
- In ASP.NET Core MVC, the *Razor view engine* uses server-side code to embed C# code within HTML elements.
- A *Razor view* contains both C# and HTML code. That's why its file extension is .cshtml. A Razor view typically stores elements that are unique to a web page.
- To execute one or more C# statements, you can declare a *Razor code block* by coding the @ sign followed by a pair of curly braces ({}). Within the curly braces, you can code one or more C# statements.
- To evaluate a C# expression and display its result, you can code a *Razor expression* by coding the @ sign and then coding the expression.

- When you attempt to build and run an app, Visual Studio may display *syntax errors* that have to be corrected before the app can be compiled.
- If a compiled statement can't be executed when you run a web app, an *exception* occurs. Then, you can use the information that's displayed in the browser to attempt to fix the exception.
- A *model* is a regular C# class that models the data for the app. The class for a model is typically stored in the Models folder.
- A *Razor view imports page* makes it easier to work with models and tag helpers. As a result, most web apps include a Razor view imports page.
- You use the @model directive to *bind* a model to a view. This kind of view is called a *strongly-typed view*.
- You can use the @Model property to access the properties and methods of the model object that's specified by the @model directive.
- *Tag helpers* automatically generate attributes for some HTML elements. They can also *bind* HTML elements to the properties of the object that's the model for the view.
- A *CSS style sheet* provides a way to store the formatting for multiple web pages in a single external file.
- A *Razor layout* provides a way to store elements that are common to multiple web pages in a single file.
- A *Razor view start* lets you specify the default Razor layout for the Razor views of a web app.
- The process of checking data to make sure it's valid is known as *data validation*.
- You can use the *validation attributes* to add *validation rules* to your model.

## Before you do the exercises for this book...

If you haven't already done so, you should install the software that's required for this book, and you should download the source code for this book. Appendixes A (Windows) and B (macOS) show how to do that.

### Exercise 2-1 Build the Future Value app using the MVC template

This exercise guides you through the development of the Future Value app that's presented in this chapter. This gives you some hands-on experience using Visual Studio to build a web app.

#### Create and set up a web app using the MVC template

1. Start a web app that's based on the MVC template as shown in figures 2-1 and 2-2. Use a project name of FutureValue and a solution name of Ch02Ex1FutureValue and store it in this directory:  
`/aspnet_core_mvc/ex_starts`
2. Delete all the files inside the Controllers, Models, and Views folders (including the files inside the Views/Home and Views/Shared folders), but don't delete the Home and Shared folders themselves.
3. Add a controller named HomeController to the Controllers folder and modify it so it contains the code from figure 2-4.
4. Add a new empty Razor view named Index to the Views/Home folder and modify it so it contains the code from figure 2-5.
5. Edit the Startup.cs file so it contains the code from figure 2-6.
6. Select the Kestrel server by selecting the name of the project (FutureValue) from the Start button drop-down list.
7. Press Ctrl+F5 to run the app. This should start the default web browser and display the Home/Index view, including the data that the HomeController stored in the ViewBag.

#### Add the model, Razor view imports page, and a strongly-typed view

8. Add a class named FutureValueModel to the Models folder and modify it so it contains the code from figure 2-9.
9. Add the Razor view imports page to the Views folder and modify it so it contains the code shown in figure 2-10.
10. Modify the code of the Home/Index view so it contains the code from figure 2-11. Make sure to include all the CSS style rules from figure 2-14 within the `<style>` element.
11. Modify the HomeController class to handle both GET and POST requests as shown in figure 2-12.
12. Run the app. If you enter valid data, it should calculate and display a future value. However, if you enter invalid data, you may get unexpected results.

### Add the Razor layout and view start, and modify the Razor view

13. Add a custom.css file to the wwwroot/css folder. If necessary, create this folder first. Then, modify it so it contains the CSS style rules shown in figure 2-14. To do that, you can cut the CSS style rules from the Home/Index file and paste them into the custom.css file.
14. Add a Razor layout named \_Layout.cshtml to the Views/Shared folder and modify it so it contains the code shown in figure 2-16. Make sure to include a <link> element that points to the custom.css file.
15. Add a Razor view start named \_ViewStart to the Views folder (not the Views/Shared folder) and modify it so it contains the code shown in figure 2-16.
16. Modify the code in Home/Index view so it contains the code shown in figure 2-16. To do that, you can cut all elements that are already specified by the Razor layout.
17. Run the app. It should work the same as it did before.

### Add data validation to the Future Value app

18. Modify the FutureValueModel class so it specifies the Required and Range attributes as shown in figure 2-18. To do that, you must use nullable types for the properties and the method.
19. Modify the HomeController class so it checks for invalid data as shown in figure 2-19.
20. Modify the Home/Index view so it displays a summary of validation messages as shown in figure 2-19.
21. Run the app. It should work correctly if you enter valid data, and it should display appropriate messages if you enter invalid data.

## Exercise 2-2

### Build the Future Value app using the Empty template

This exercise guides you through the development of the Future Value app that's presented in this chapter if you start from the Empty template instead of the MVC template.

#### Create and set up a web app using the Empty template

1. Start a web app that's based on the Empty template as shown in figures 2-1 and 2-2. Use a project name of FutureValue and a solution name of Ch02Ex2FutureValue and store it in this directory:

`/aspnet_core_mvc/ex_starts`

2. Add the Controllers, Models, and Views folders to the project.
3. Add a Home folder and Shared folder within the Views folder that you just created.
4. Follow exercise 2-1 starting at step 3.



# 3

## How to make a web app responsive with Bootstrap

In chapter 2, you learned how to build the Future Value app using HTML and CSS to format the view for the app so it looks good on devices such as computers that have large screens. However, since so many users browse websites with devices such as phones that have small screens, it's important to make sure your web apps look good on devices of every size. To do that, you can use CSS and JavaScript provided by a client-side library such as Bootstrap to make your web apps responsive as described in this chapter. This can improve the appearance of your apps and make them more user friendly.

<b>An introduction to responsive web design .....</b>	<b>82</b>
A responsive user interface.....	82
How to add client-side libraries such as Bootstrap and jQuery .....	84
How to manage client-side libraries with LibMan.....	86
How to enable client-side libraries .....	88
<b>How to get started with Bootstrap.....</b>	<b>90</b>
The classes of the Bootstrap grid system .....	90
How the Bootstrap grid system works.....	92
How to work with forms .....	94
How to work with buttons, images, and jumbotron.....	96
How to work with margins and padding .....	98
The code for the view of the Future Value app .....	100
<b>More skills for Bootstrap CSS classes .....</b>	<b>102</b>
How to format HTML tables.....	102
How to align and capitalize text.....	104
How to provide context.....	106
<b>More skills for Bootstrap components .....</b>	<b>108</b>
How to work with button groups .....	108
How to work with icons and badges.....	110
How to work with button dropdowns .....	112
How to work with list groups .....	114
How to work with alerts and breadcrumbs.....	116
<b>How to work with navigation bars .....</b>	<b>118</b>
How to create navs.....	118
How to create navbars .....	120
How to position navbars .....	122
<b>Perspective .....</b>	<b>124</b>

## An introduction to responsive web design

---

Due to the popularity of phones and tablets, it's important to make sure your web apps look good on devices of every size. This is called *responsive web design*. An app that uses responsive web design doesn't just look good on phones and tablets, it's easier to use too. That's because you don't have to scroll and resize on a small screen. In fact, this is so important that apps that are mobile friendly perform better in Google's search results.

One way to make your web apps responsive is to use CSS3 *media queries*. Media queries let you adjust the layout of a page based on conditions, such as the width of the screen. Because it can be time consuming to develop media queries, though, frameworks have been developed to automate this process.

A *framework* contains general code for common situations and can be customized to meet the needs of individual projects. *Bootstrap* is a popular framework for responsive web design that was originally developed by Twitter. Bootstrap uses CSS and JavaScript to make a web page automatically adjust for different screen sizes.

## A responsive user interface

---

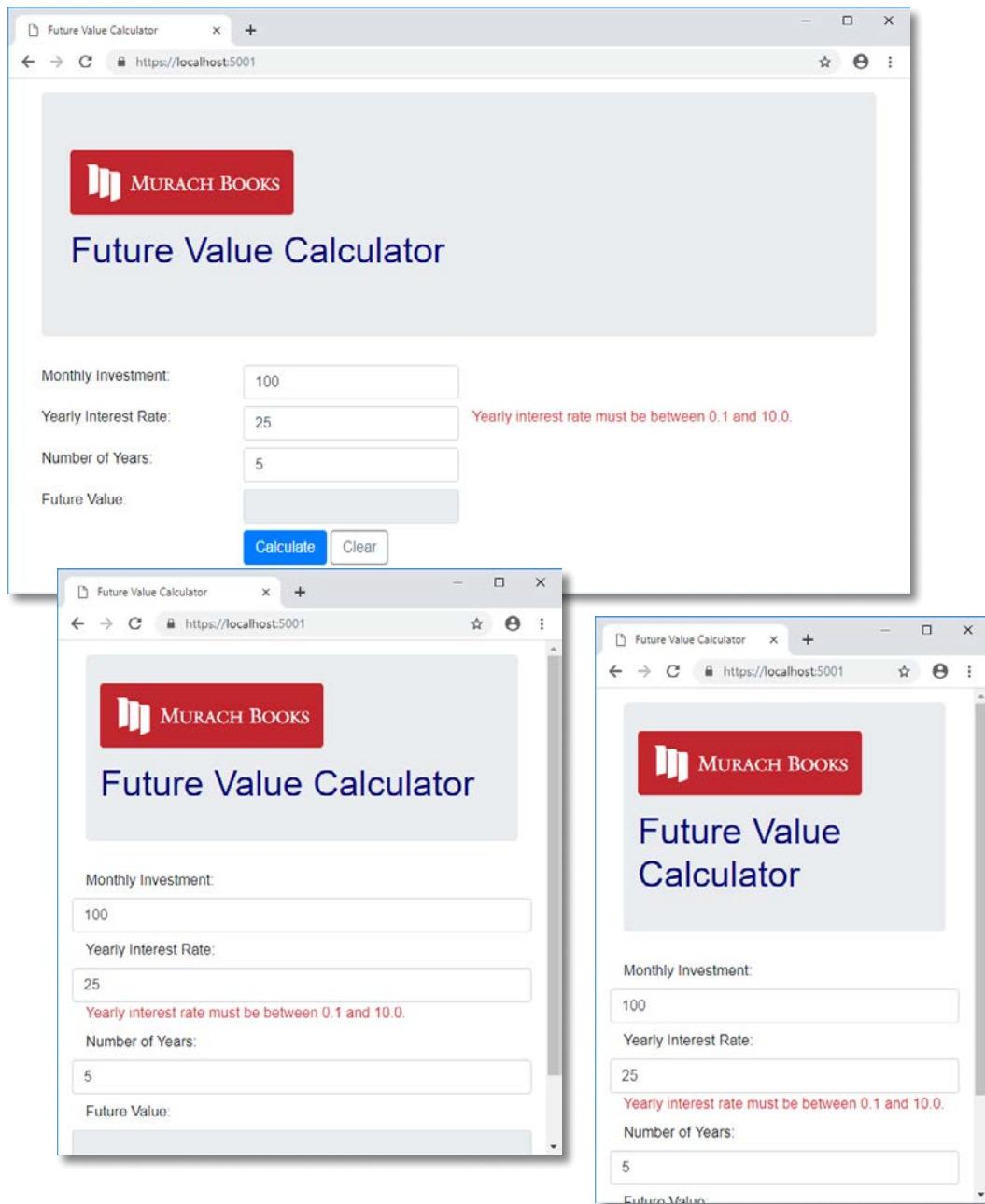
Figure 3-1 shows a version of the Future Value app that uses Bootstrap to make it responsive. Here, the first screen corresponds to the width of a desktop browser. This page displays a large banner section followed by rows of labels, controls, and validation messages.

The second screen corresponds to the width of a tablet browser. In this screen, the page still has a large banner, but the labels, controls, and validation messages are stacked on top of each other. This narrows the page to match the narrowed screen, and makes it so you don't need to scroll from side to side to see the whole page. The controls also span the width of the screen to make them easier to use on touch screens.

The third screen corresponds to the width of a smart phone browser. In this screen, the banner and its contents have adjusted their positions to accommodate the narrower screen. The labels, controls, and validation messages are still stacked, and they still span the screen so they're easy to use on touch screens. But they've also narrowed to fit in the screen.

When you use Bootstrap, these changes take place automatically when you narrow or widen the browser window or when you display the page on a phone, tablet, or desktop computer. All you have to do is add Bootstrap to your app and then use the classes it provides.

## The responsive Future Value app at different screen widths



### Description

- A web app should adapt to every screen size. This is called *responsive web design*.
- You can use the open-source *Bootstrap* library to implement a responsive web design with your ASP.NET Core MVC apps.

Figure 3-1 A responsive user interface

## How to add client-side libraries such as Bootstrap and jQuery

---

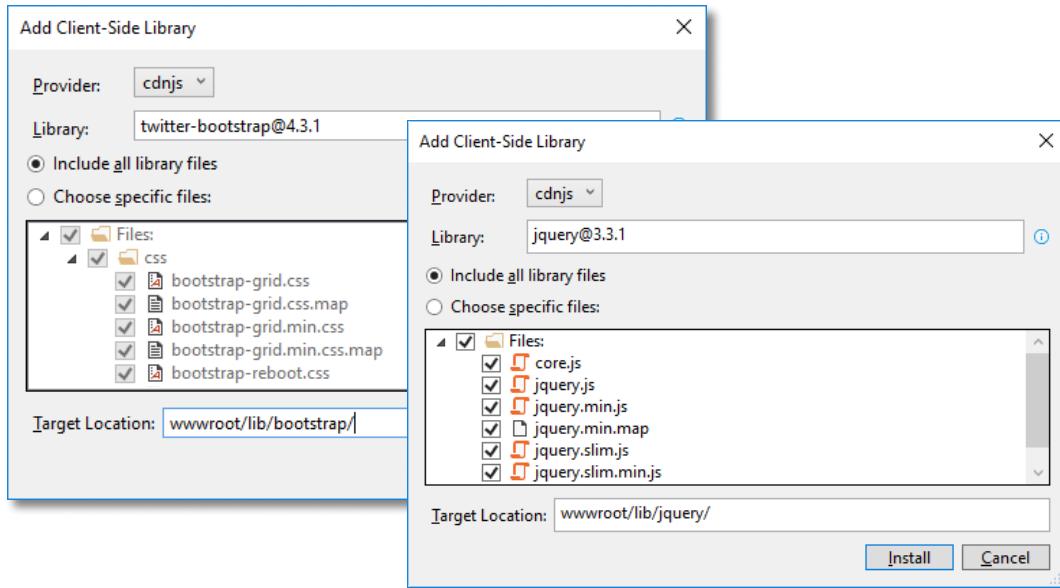
The MVC template that you learned about in the previous chapter includes a version of Bootstrap by default. However, this version of Bootstrap might be different than the version presented in this chapter. All of the apps presented in this book have been tested against the versions of Bootstrap, jQuery, and Popper.js specified in this figure. As a result, if you want these libraries to work as described in this book, you should use these versions for your apps too. If you prefer to use the newest versions of these libraries, you may need to modify the code presented in this book to get it to work correctly with the newer libraries.

Figure 3-2 shows how to add Bootstrap and other useful client-side libraries to your app using the *Library Manager*, also known as *LibMan*. To start, the procedure shows how to add the jQuery JavaScript library that Bootstrap depends on. Then, it shows how to add Bootstrap. Next, it shows how to add Popper.js, which is another JavaScript library that Bootstrap depends on.

In addition, this procedure shows how to add two other jQuery libraries that are necessary to enable client-side data validation. This allows data validation like the validation described in the previous chapter to run on the client without needing to make an HTTP request to the server. Bootstrap doesn't depend on these libraries, but it's common to add them to your app at the same time that you add the other client-side libraries that are needed by Bootstrap.

By convention, an ASP.NET Core MVC app stores these libraries in the wwwroot/lib folder as shown in this figure. In addition, when you use the procedure in this figure to add client-side libraries, Visual Studio creates a file named libman.json. If you want, you can edit this file to easily manage the versions or locations of your client-side libraries as described in the next figure.

## The dialog boxes for adding client-side libraries



## How to add the Bootstrap and jQuery libraries to a web app

1. Start Visual Studio and open a project. In the Solution Explorer, expand the wwwroot/lib folder and delete any old Bootstrap or jQuery libraries.
2. In the Solution Explorer, right-click on the project name and select the Add→Client-Side Library item.
3. In the dialog box that appears, type “jquery@”, select “3.3.1” from the list that appears, and click the Install button.
4. Repeat steps 2 and 3 for the library named “twitter-bootstrap@4.3.1”, but change the target location to “www/lib/bootstrap”.
5. Repeat steps 2 and 3 for the library named “popper.js@1.14.7”.
6. Repeat steps 2 and 3 for the library named “jquery-validate@1.19.0”.
7. Repeat steps 2 and 3 for the library named “jquery-validation-unobtrusive@3.2.11”.
8. In the Solution Explorer, expand the wwwroot/lib folder and view the libraries that have been installed. If you didn’t already do it in step 4, rename the twitter-bootstrap folder to bootstrap.

### Description

- You can use the *Library Manager*, also known as *LibMan*, to add client-side libraries such as Bootstrap and jQuery to a project.
- The MVC template that you learned about in the previous chapter includes a version of Bootstrap by default. However, this version of Bootstrap might be different than the version presented in this chapter.

Figure 3-2 How to add client-side libraries such as Bootstrap and jQuery

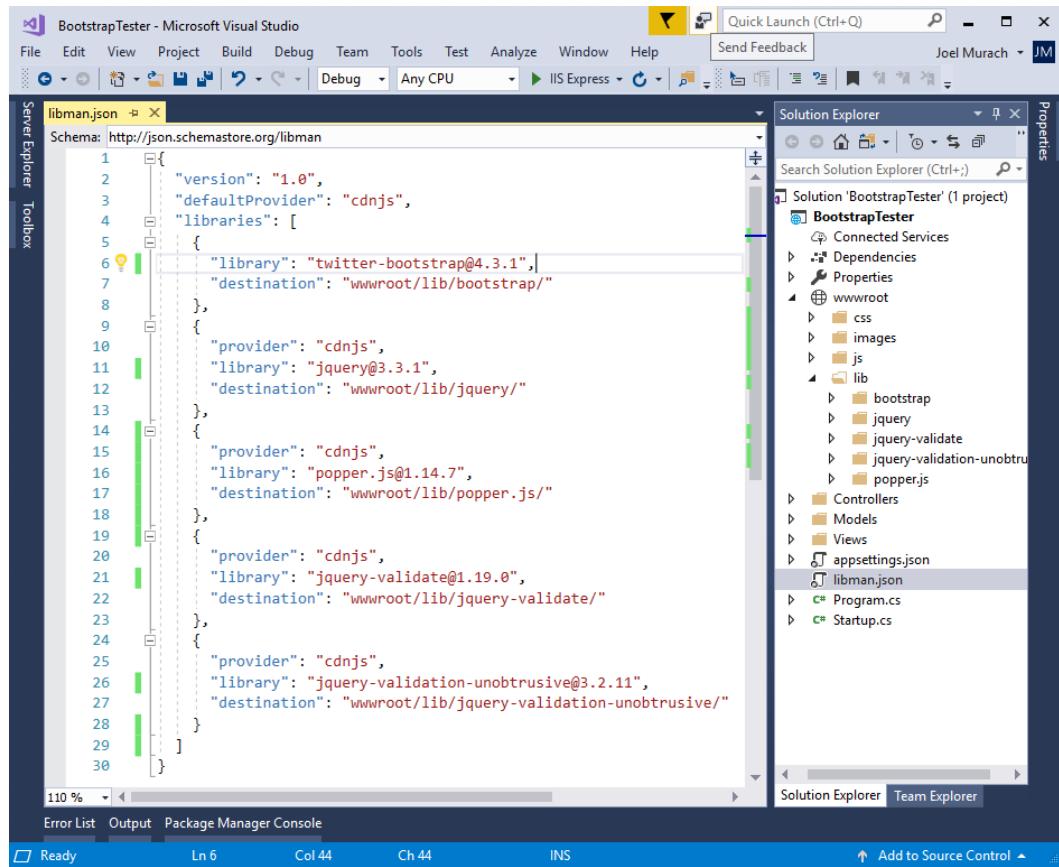
## How to manage client-side libraries with LibMan

---

By convention, an ASP.NET Core MVC app stores its client-side libraries in the wwwroot/lib folder shown in figure 3-3. In addition, when you use the procedure in the previous figure to add client-side libraries, Visual Studio creates a libman.json file like the one shown in this figure. If you want, you can easily manage the versions or locations of your client-side libraries by editing the contents of this file. Then, you can right-click on this file and select the Restore Client-Side Libraries item. This automatically updates the client-side libraries on your system so they match the contents of the libman.json file.

If your app contains client-side libraries that aren't in the libman.json file, you can remove them by right-clicking on the libman.json file and selecting the Clean Client-Side Libraries item. Then, you can restore the client-side libraries specified by the libman.json file by right-clicking the libman.json file and selecting the Restore Client-Side Libraries item.

## The libman.json file that's created when you add a client-side library



### Description

- To manage client-side libraries, you can open the libman.json file and edit it to specify the correct versions and locations for your client-side libraries.
- To update all client-side libraries so they match the libman.json file, you can right-click on the libman.json file and select the Restore Client-Side Libraries item.
- To remove all client-side libraries, you can right-click on the libman.json file and select the Clean Client-Side Libraries item.

Figure 3-3 How to manage client-side libraries with LibMan

## How to enable client-side libraries

---

Once you've added Bootstrap and the JavaScript libraries that it depends on to your app, you can enable those libraries by adding some HTML elements to the `<head>` element of a view. Fortunately, most views use a Razor layout. As a result, you typically just need to add these elements to the view's layout. For example, figure 3-4 shows how to add the necessary elements to the layout for the Future Value app presented in the previous chapter.

The first element you need to add is a *meta tag*. A meta tag provides information about your web app to browsers but isn't displayed to the user. For Bootstrap, you need to add a meta tag whose name attribute is set to "viewport".

The *viewport* is the part of the web page that's visible to a viewer, and its size varies by device. When you use Bootstrap, you should set the viewport meta tag as shown in this figure. This meta tag makes sure that the width of the page corresponds to the width of the device.

The two `<link>` elements attach the external style sheets for the app. For these elements to work properly, you must code them in the sequence shown in this figure. That way, the custom style sheet can override existing styles or add new styles to the Bootstrap style sheet.

The first three `<script>` elements attach the JavaScript libraries for jQuery, Popper.js, and Bootstrap. Again, you must code these elements in the sequence shown in this figure. That's because the Bootstrap JavaScript library depends on the jQuery and Popper.js libraries.

The last two `<script>` elements attach the JavaScript libraries necessary for client-side data validation. As mentioned earlier, these libraries aren't necessary for Bootstrap to work properly. However, you typically want to add these libraries to any views that perform data validation.

The folders for these client-side libraries contain many more files than those that are specified in this figure. For example, all of these libraries include a *minified* version and a regular version. In most cases, you use the minified files as shown in this figure because they load faster. However, if you want to examine or debug the code that Bootstrap provides, you can use the regular files because they're easier for humans to read and understand.

Once you've made the elements for Bootstrap available to a view, Visual Studio provides IntelliSense support for the various CSS classes provided by Bootstrap. That makes it easy to discover and enter the names of the Bootstrap classes as you add them to a view.

In this figure, the `<script>` tags are coded at the end of the `<head>` element. However, many programmers prefer to code the `<script>` elements at the end of the `<body>` element. That way, the browser loads most of the HTML before it loads the JavaScript libraries. This allows the HTML to load faster, but it can cause the HTML elements to jump around in the browser after the JavaScript loads, which can be annoying to users.

A project that's based on the MVC template may include an extra `/dist` folder in the paths to the Bootstrap, jQuery, and jQuery validation libraries. As a result, if you use LibMan to update these client libraries, you may need to modify the `<link>` and `<script>` elements so they point to the correct folders.

## How to use a Razor layout to enable client-side libraries

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>@ ViewData["Title"]</title>
    <link rel="stylesheet" href="~/lib/bootstrap/css/bootstrap.min.css" />
    <link rel="stylesheet" href="~/css/custom.css" />

    <script src="~/lib/jquery/jquery.min.js"></script>
    <script src="~/lib/popper.js/popper.min.js"></script>
    <script src="~/lib/bootstrap/js/bootstrap.min.js"></script>

    <script src="~/lib/jquery-validate/jquery.validate.min.js"></script>
    <script src="~/lib/jquery-validation-unobtrusive/
        jquery.validate.unobtrusive.min.js"></script>
</head>
<body>
    @RenderBody()
</body>
</html>
```

### An extra directory that may be included by the MVC template

```
<link rel="stylesheet" href="~/lib/bootstrap/dist/css/bootstrap.min.css" />
```

#### Description

- To make client-side CSS libraries such as Bootstrap available to the views in your web app, you can add the `<link>` elements for the CSS files to the `<head>` element in the Razor layout for the view.
- The `<link>` element for the Bootstrap CSS file should be coded before the `<link>` element for your own custom CSS files. That way, your CSS styles override the Bootstrap styles. Creating a separate style sheet is the preferred way of making changes to Bootstrap.
- To make client-side JavaScript libraries such as jQuery available to the views in your web app, you can add the `<script>` elements for those libraries to the `<head>` element in the Razor layout for the view.
- The `<script>` element for the jQuery library should come first, then the `<script>` element for the Popper.js library, then the `<script>` element for the Bootstrap JavaScript file.
- Most of these libraries include a *minified* version of the library that has removed unnecessary characters such as spaces and indentation. This decreases the size of the file and improves load time but makes it more difficult for humans to read the library. Minified libraries are typically identified with a suffix of `.min.css` or `.min.js`.
- After you've added the necessary `<link>` and `<script>` elements to your web form, Visual Studio provides IntelliSense for the Bootstrap CSS classes.
- The *viewport* is the part of the page that is visible to viewers. The *viewport meta tag* controls the width of the viewport.
- The MVC template may include an extra `/dist` folder for the Bootstrap, jQuery, and jQuery validation libraries.

---

Figure 3-4 How to enable client-side libraries

## How to get started with Bootstrap

So far, you have learned how to install and enable the client-side libraries necessary to support Bootstrap. Now, you'll learn how to use some of the CSS classes available from Bootstrap to style the view for the Future Value app.

### The classes of the Bootstrap grid system

To create responsive web apps, Bootstrap uses a grid system that's based on containers, rows, and columns. A container contains one or more rows, and a row can contain up to 12 columns. Bootstrap provides predefined CSS classes to work with its grid system. The most important of these classes are presented in the first table in figure 3-5.

To start, you use the container and container-fluid classes to identify the main content of the page. The difference between them is that an element that uses the container class is centered in the screen and has a specific width in pixels based on the viewport's width. This is sometimes called a *boxed layout*. An element that uses the container-fluid class, by contrast, is always the same width as the viewport. This is sometimes called a *full width layout*.

The main content of a page should be divided into rows using the row class. Then, within each row, you can code elements with one or more of the column classes that control how the content in the row is displayed. If you use the col class, the width of the column is automatically sized. In the first example, for instance, this class is used on two columns so the columns are sized the same.

If that's not what you want, you can specify a class with one of the screen sizes listed in the second table in this figure, along with the number of columns that the element should span. To understand how this works, the second example shows a <div> element that uses the col-md-4 class. This means that when the element is displayed on a desktop (md) whose width is greater than or equal to 992 pixels, it occupies four columns.

The third example is similar, except that it specifies a second class for a large desktop (lg) screen whose width is greater than or equal to 1200 pixels. Then, the element occupies only three columns. However, it still occupies four columns on a desktop (md) screen that's greater than or equal to 992 pixels but less than 1200 pixels.

Neither of these examples includes classes for smaller devices like tablets (sm) and phones (none). Because of that, the element occupies all twelve columns on those devices, since that's the default.

You can also use CSS classes to move an element to the right a specified number of columns. To do that, you use a class that indicates the screen size and the number of columns that the element should be offset. This is illustrated in the fourth example in this figure. Here, the element occupies four columns on a desktop (md) screen just as in the first two examples. However, the element is moved one column to the right.

If you have a hard time visualizing this, don't worry. The next figure presents an example that demonstrates how the grid system works.

## The URL for the Bootstrap documentation

<https://getbootstrap.com/docs/>

### Valid class values

Class	Description
<code>container</code>	Contains rows or other content. Centered in the <body> element, with a specific width based on the viewport size.
<code>container-fluid</code>	Contains rows or other content. Set to 100% of the width of the viewport.
<code>row</code>	Contains columns inside a container.
<code>col</code>	A column inside a row that will be automatically sized.
<code>col-size-count</code>	The number of columns an element should span on the specified screen size. The number of columns in a row should not exceed 12.
<code>offset-size-count</code>	The number of columns an element should be moved to the right on the specified screen size.

### Valid size values

Size	Description
<code>lg</code>	A large screen with a width greater than or equal to 1200 pixels (e.g. large desktops).
<code>md</code>	A medium screen with a width greater than or equal to 992 pixels (e.g. desktops).
<code>sm</code>	A small screen with a width greater than or equal to 768 pixels (e.g. tablets).
(none)	An extra small screen with a width less than 768 pixels (e.g. phones).

#### Two columns that are automatically sized

```
<div class="col">Column 1</div><div class="col">Column 2</div>
```

#### An element that spans four columns on medium and large screens

```
<div class="col-md-4">This element spans four columns</div>
```

#### An element that spans 4 columns on medium screens and 3 on large

```
<div class="col-md-4 col-lg-3">This element spans three or four columns</div>
```

#### An element that is moved one column to the right on medium screens

```
<div class="col-md-4 offset-md-1">This element is offset by one column</div>
```

### Description

- Bootstrap uses a grid system based on containers, rows, and columns. All rows should be inside a container, and each row must contain no more than 12 columns.
- You can assign a different column class to an element for each screen size to specify the number of columns the element should span at those sizes.
- If you don't assign a column class for a screen size, the class for the next smallest screen size will be used.

---

Figure 3-5 The classes of the Bootstrap grid system

## How the Bootstrap grid system works

To help you understand how the Bootstrap grid system works, figure 3-6 shows an example of a simple grid that contains three rows. Here, the grid is shown as it would appear on a desktop, a tablet, and a phone. In the HTML, each row is defined by a `<div>` element that's assigned the class named “row”. In addition, the `<div>` elements that define the rows are coded within a `<main>` element that's assigned the class named “container”. Each row in the grid contains two `<div>` elements that are used to define the columns.

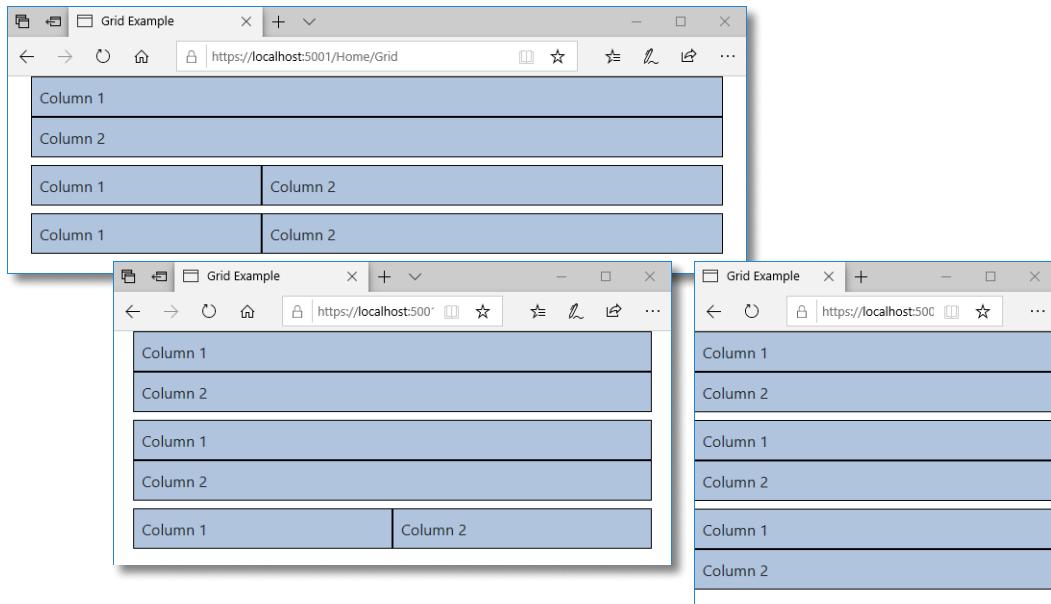
In the first row, each `<div>` element specifies a width of twelve columns for the extra small phone size (none). As a result, each column spans all twelve columns of the row, or the entire width of the container, regardless of the screen size. That causes the columns to be stacked vertically as shown here.

In the second row, each `<div>` element specifies a class for the desktop viewport (md). Here, the first `<div>` element specifies a width of four columns and the second element specifies a width of eight columns. This causes the `<div>` elements to be displayed side by side at the desktop width. It also causes these elements to be displayed side by side at the large desktop width, since no classes are specified for that width (lg). In that case, Bootstrap uses the width of the next smallest screen size (md). By contrast, Bootstrap stacks the `<div>` elements at the tablet (sm) and phone (none) sizes. That's because this code doesn't specify column classes for these viewport sizes. In that case, each `<div>` element spans all twelve columns.

In the third row, the `<div>` elements specify the same column classes for the desktop viewport as the `<div>` elements in the second row. In addition, they're assigned column classes for the tablet (sm) viewport. These classes specify a width of six columns for each `<div>` element. As a result, desktop and large desktop devices display these `<div>` elements side by side at the same widths as the `<div>` elements in the second row. However, tablets display these `<div>` elements side by side but at equal widths as shown by the second screen. Finally, phones stack these `<div>` elements vertically as shown by the third screen.

This figure also presents some custom CSS classes that override the Bootstrap CSS classes. This custom CSS makes it easy to see the rows and columns in the grid by providing custom margins, padding, borders, and a background color. As you review this CSS, you may notice that the margin-bottom and padding attributes use a rem unit to specify size instead of using the more traditional em unit. *Rem units* work similarly to em units. The main difference is that the size of an em is relative to the immediate container element, and the size of a rem, which stands for “root em”, is relative to the root element of the document. In short, rems are used by Bootstrap 4 instead of ems. As a result, when you override Bootstrap classes, it usually makes sense to use rems, not ems.

## A Bootstrap grid on medium, small, and extra small screens



### The HTML for the grid example

```
<main class="container">
  <div class="row">
    <div class="col-12">Column 1</div>
    <div class="col-12">Column 2</div>
  </div>
  <div class="row">
    <div class="col-md-4">Column 1</div>
    <div class="col-md-8">Column 2</div>
  </div>
  <div class="row">
    <div class="col-md-4 col-sm-6">Column 1</div>
    <div class="col-md-8 col-sm-6">Column 2</div>
  </div>
</main>
```

### Custom CSS classes that override the Bootstrap CSS classes

```
.row {
  margin-bottom: 0.5rem;
}
.row div {
  border: 1px solid black;
  padding: 0.5rem;
  background-color: lightsteelblue;
}
```

#### Note

- Bootstrap CSS classes uses *rem units* for sizing. A rem (root em) unit works similarly to the more traditional em unit, but its size is relative to the root element rather than the current element.

---

Figure 3-6 How the Bootstrap grid system works

## How to work with forms

---

Bootstrap also provides some predefined CSS classes for working with the labels and controls of a form. The table in figure 3-7 presents some of the most important of these classes.

To start, you can use the form-vertical or form-horizontal class to determine the layout of the labels and controls on the form. With a vertical layout, the labels and controls stack on top of each other and the controls span the width of the viewport, as shown in the first example. With a horizontal layout, you use the grid system that you learned about in the previous figure to align labels and controls horizontally, as shown in the second example. In the HTML for the form with the vertical layout, the `<form>` element doesn't include a class attribute that specifies the form-vertical class. That's because Bootstrap uses vertical layout for forms by default.

You use the form-group class with vertical layouts. This class groups labels and controls and determines the spacing for those controls. In the first example, `<div>` elements that use the form-group class identify the labels and controls that go together.

You use the form-control class to apply styling to `<input>`, `<textarea>`, and `<select>` elements. This class improves the appearance of these elements, and it makes them easier to use on mobile devices. For example, it improves the appearance of these elements by making them taller and by rounding the corners of these elements. It also makes them span the width of the viewport in a vertical layout or the width of the column span in a horizontal layout, which makes them easier to use on touch screens.

To align the controls in the horizontal layout, you use the grid system. In the HTML for the horizontal layout, for example, the `<div>` elements use the row class to identify the rows in the form. Within these `<div>` elements, the `<label>` elements use the col-sm-2 class, and the `<input>` elements that define the text boxes use the col-sm-10 class. That way, the labels each span two columns on tablets, desktops, and large desktops, and the controls each span ten columns. Because a column class isn't assigned for phones, however, Bootstrap stacks the labels and controls vertically when it displays them on a phone. In other words, since the layout uses the grid system, it is responsive.

You use the last class, control-label, to apply styling to a label that's grouped with a control. Both of the examples use this class to style the labels.

## Some Bootstrap CSS classes for working with forms

Class	Description
<code>form-vertical</code>	The form labels and controls stack vertically. This is the default for forms.
<code>form-horizontal</code>	Used with the grid system to align labels and controls in a horizontal layout.
<code>form-group</code>	Applies spacing to labels and controls that go together.
<code>form-control</code>	Applies styling to input, textarea, or select controls in a form.
<code>control-label</code>	Applies styling to a control's label.

### A form with two text boxes in vertical layout

Email:

Password:

#### The HTML for the form

```
<form asp-action="Login" method="post">
    <div class="form-group">
        <label for="email" class="control-label">Email:</label>
        <input type="email" id="email" class="form-control" />
    </div>
    <div class="form-group">
        <label for="pwd" class="control-label">Password:</label>
        <input id="pwd" type="password" class="form-control" />
    </div>
</form>
```

### A form with two text boxes in horizontal layout

Email:

Password:

#### The HTML for the form

```
<form asp-action="Login" method="post" class="form-horizontal">
    <div class="row">
        <label for="email" class="control-label col-sm-2">Email:</label>
        <input type="email" id="email" class="form-control col-sm-10" />
    </div>
    <div class="row">
        <label for="pwd" class="control-label col-sm-2">Password:</label>
        <input type="password" id="pwd" class="form-control col-sm-10" />
    </div>
</form>
```

---

Figure 3-7 How to work with forms

## How to work with buttons, images, and jumbotrons

---

The grid and form classes that you have learned about so far allow you to set the basic structure of a web page and control the layout for different viewport sizes. Bootstrap also provides many CSS classes for styling individual elements and components that you'll learn about later in this chapter. But first, figure 3-8 presents some of the most common classes, like those for styling buttons and images.

In addition, this figure shows how to create a Bootstrap component known as a *jumbotron*. This component uses a large grey box with rounded corners to highlight content. In this figure, the example uses the jumbotron class to display an image. However, it can also be used to display textual messages.

In this example, the HTML starts with a `<div>` element that uses the `container` class. This centers the content and adjusts its size based on the size of the viewport. Then, the `<header>` element uses the `jumbotron` class to display a large grey box with rounded corners.

Within the `<header>` element, the HTML uses two additional classes to style an image. First, the `img-fluid` class causes the size of the image to be adjusted as the size of the viewport changes. That way, the entire image is always visible in the viewport. Then, the `rounded` class causes the image to be displayed with rounded corners, even though the original image doesn't have rounded corners. However, this won't work in older versions of Internet Explorer.

Within the `<main>` element, the HTML uses Bootstrap classes to provide appropriate spacing and styling for the elements in the form. Here, the elements in the form aren't full screen and vertically stacked like they were in the vertical form example of the previous figure. That's because the form controls in this example don't use the `form-control` class. This shows that you can use just the CSS classes you need to get the look you want.

Within the `<form>` element, the example in the figure uses the `mr-2` class in the `<span>` element to increase the right margin as described in the next figure. Then, it uses some of the button classes to style the Yes and No buttons. The `btn` class provides the basic size and rounded corners. The `btn-primary` class adds emphasis to the Yes button by making it blue with white text. And the `btn-outline-secondary` class makes the No button white with grey text and a grey border.

## Some of the Bootstrap CSS classes for working with buttons

Class	Description
<code>btn</code>	Produces a simple button with rounded corners.
<code>btn-primary</code>	Sets the background color to blue and the text to white.
<code>btn-secondary</code>	Sets the background color to grey and the text to white.
<code>btn-outline-primary</code>	Sets the background color to white and the border and text to blue.
<code>btn-outline-secondary</code>	Sets the background color to white and the border and text to grey.

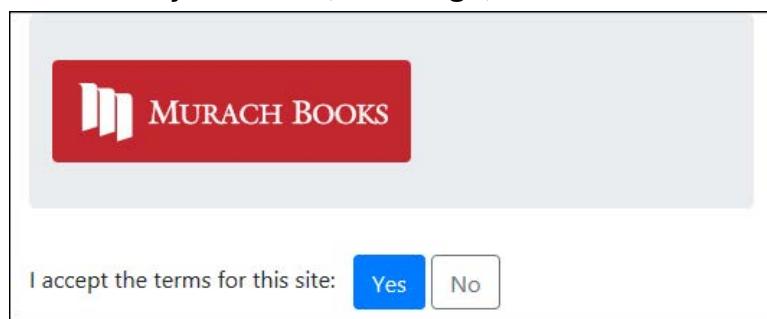
## Some of the Bootstrap CSS classes for working with images

Class	Description
<code>img-fluid</code>	Makes the image automatically adjust to fit the size of the viewport.
<code>rounded</code>	Rounds the corners of the image.

## A Bootstrap CSS class for creating a jumbotron

Class	Description
<code>jumbotron</code>	A large grey box with rounded corners and a large font.

## A form with a jumbotron, an image, and two buttons



### The HTML

```

<div class="container">
    <header class="jumbotron">
        
    </header>
    <main>
        <form asp-action="Index" method="post">
            <span class="mr-2">I accept the terms for this site:</span>
            <button type="submit" class="btn btn-primary">Yes</button>
            <button id="btnNo" class="btn btn-outline-secondary">No
            </button>
        </form>
    </main>
</div>

```

Figure 3-8 How to work with buttons, images, and jumbotrons

## How to work with margins and padding

---

Figure 3-9 shows how to use the Bootstrap CSS classes for setting the margins and padding of an element. Here, the classes use m or p to specify margin or padding. Then, they use t, r, b, or l to specify the top, right, bottom, and left side of an element. Or, if the class doesn't specify a side, the class applies to all four sides of the element.

The example in this figure begins by setting the mt (margin top) class for the `<header>` element to a size of 2. This adds a small margin of .5 rem between the `<header>` element and the top of the page.

This example continues by setting the mr (margin right) class for the `<span>` element to a size of 2. This adds a small margin of .5 rem to the right of the text in the `<span>` element and before the first `<button>` element. Similarly, this example adds a small margin to the right of the first `<button>` element and before the second `<button>` element.

This example finishes by setting the p (padding) class for both `<button>` elements to a size of 3. Since it doesn't specify a side, this adds padding of 1 rem to all four sides of these buttons. That's why these buttons appear larger than the buttons shown in the previous figure.

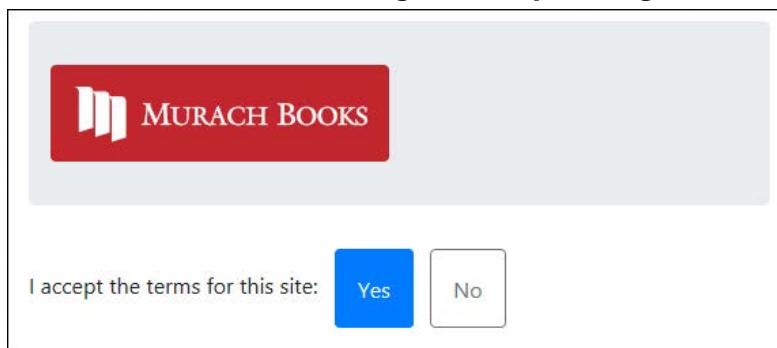
## Some Bootstrap CSS classes for working with margins

Class	Description
<code>mt-size</code>	Sets the margin for the top to a specified size from 0 to 5. By default, these sizes correspond to rem unit values of 0, .25, .5, 1, 1.5, and 3.
<code>mr-size</code>	Sets the margin for the right side to the specified size.
<code>mb-size</code>	Sets the margin for the bottom to the specified size.
<code>ml-size</code>	Sets the margin for the left side to the specified size.
<code>m-size</code>	Sets all four margins to the specified size.

## Some Bootstrap CSS classes for working with padding

Class	Description
<code>pt-size</code>	Sets the padding for the top to the specified size from 0 to 5. By default, these sizes correspond to rem unit values of 0, .25, .5, 1, 1.5, and 3.
<code>pr-size</code>	Sets the padding for the right side to the specified size.
<code>pb-size</code>	Sets the padding for the bottom to the specified size.
<code>pl-size</code>	Sets the padding for the left side to the specified size.
<code>p-size</code>	Sets the padding for all four sides to the specified size.

## Some elements that use margins and padding



### The HTML

```
<div class="container">
    <header class="jumbotron mt-2">
        
    </header>
    <main>
        <form asp-action="Index" method="post">
            <span class="mr-2">I accept the terms for this site:</span>
            <button type="submit"
                class="btn btn-primary p-3 mr-2">Yes</button>
            <button id="btnNo"
                class="btn btn-outline-secondary p-3">No</button>
        </form>
    </main>
</div>
```

Figure 3-9 How to work with margins and padding

## The code for the view of the Future Value app

Now that you understand how the basic Bootstrap classes work, you should be able to understand the code for the view of the responsive Future Value app. Figure 3-10 shows this code. This results in a Future Value app that works on different screen sizes as shown in figure 3-1. Of course, this assumes that the view uses a Razor layout like the one shown in figure 3-4.

To start, the view begins with a `<div>` element that uses the container class to center the page and adjust its width to the viewport. This `<div>` element contains a `<header>` element and a `<main>` element. The `<header>` element uses the jumbotron class, which styles it as a large grey box with rounded corners.

Within the `<header>` element, an `<img>` element uses classes that give it rounded corners and cause its size to automatically adjust to the size of the viewport. Then, an `<h1>` element displays the title of the app. Because the `<h1>` element is coded within a jumbotron component, it has text with a font that's an appropriate size for a jumbotron heading. In addition, the `<h1>` element uses the `mt-size` class to increase the size of the top margin for this heading.

The `<main>` element contains a `<form>` element that uses the `form-horizontal` class. As a result, this view displays labels and controls in a side-by-side layout. This `<form>` element contains five `<div>` elements that use the `row` class to identify the five rows of the form.

The first row contains the monthly investment label, a text box, and a validation message. Here, the label uses the `control-label` class for styling and the `col-sm-3` class so it spans three columns on tablet, desktop, and large desktop devices. Next, the text box uses the `form-control` class and the `col-sm-3` class so that it also spans three columns. Finally, the validation message uses the `col` class so that it spans the remaining six columns. To display this message, a `<span>` element uses the `asp-validation-for` tag helper described in chapter 11. In addition, this `<span>` element uses the `text-danger` class that's presented a little later in this chapter to display the error message in red.

The second and third rows display the text boxes for the interest rate and number of years. The code for these two rows works much like the first row. As a result, you shouldn't have much trouble understanding how they work.

The fourth row also works similarly. However, it doesn't display an error message. As a result, it only specifies six out of twelve possible columns. This shows that you don't have to specify all columns when you work with a grid.

The fifth row contains the Calculate and Clear buttons. The `<div>` element that contains these buttons uses the `col` and `offset-sm-3` classes. The `offset` class moves the buttons three columns to the right, which aligns them with the controls in the previous rows. However, to get the alignment just right, this `<div>` element also uses the `ml-0` class to set the left margin for the element to 0. Then, the buttons use the `btn`, `btn-primary`, and `btn-outline-secondary` classes for styling.

Note that no column classes are included for extra small devices. As a result, on those devices, Bootstrap stacks the form elements vertically and makes them as wide as the screen as shown by the third screen in figure 3-1.

## The code for the Index view

```
@model FutureValueModel
@{
    ViewBag.Title = "Future Value Calculator";
}
<div class="container">
    <header class="jumbotron">
        
        <h1 class="mt-3">@ViewBag.Title</h1>
    </header>
    <main>
        <form asp-action="Index" method="post" class="form-horizontal">
            <div class="row">
                <label asp-for="MonthlyInvestment"
                    class="control-label col-sm-3">
                    Monthly Investment:</label>
                <input asp-for="MonthlyInvestment"
                    class="form-control col-sm-3" />
                <span asp-validation-for="MonthlyInvestment"
                    class="text-danger col"></span>
            </div>
            <div class="row">
                <label asp-for="YearlyInterestRate"
                    class="control-label col-sm-3">
                    Yearly Interest Rate:</label>
                <input asp-for="YearlyInterestRate"
                    class="form-control col-sm-3" />
                <span asp-validation-for="YearlyInterestRate"
                    class="text-danger col"></span>
            </div>
            <div class="row">
                <label asp-for="Years"
                    class="control-label col-sm-3">
                    Number of Years:</label>
                <input asp-for="Years"
                    class="form-control col-sm-3" />
                <span asp-validation-for="Years"
                    class="text-danger col"></span>
            </div>
            <div class="row">
                <label class="control-label col-sm-3">Future Value:</label>
                <input value="@ViewBag.FutureValue" readonly
                    class="form-control col-sm-3" />
            </div>
            <div class="row">
                <div class="col offset-sm-3 pl-0">
                    <button type="submit"
                        class="btn btn-primary">Calculate</button>
                    <a asp-action="Index"
                        class="btn btn-outline-secondary">Clear</a>
                </div>
            </div>
        </form>
    </main>
</div>
```

---

Figure 3-10 The code for the view of the Future Value app

## More skills for Bootstrap CSS classes

So far, this chapter has presented the skills you need to use Bootstrap to create the responsive version of the Future Value app. These skills include using Bootstrap CSS classes to style buttons and text boxes. Now, you're ready to learn how to use some other Bootstrap CSS classes to make other apps responsive. For example, you're ready to learn the skills you need to make the Movie List app presented in the next chapter responsive.

### How to format HTML tables

The table in figure 3-11 shows some of the Bootstrap CSS classes for working with HTML tables. Then, the first example shows an HTML table that uses the default Bootstrap styling. For these Bootstrap table classes to work correctly, an HTML table must have `<thead>` and `<tbody>` elements as shown in this figure.

The second example shows the same table with additional table classes applied. Here, the `table-striped` class applies alternating colors to the rows of the table. The `table-bordered` class adds a border around the table and between cells. And the `table-hover` class makes the color of a row change when you hover the mouse pointer over it.

The `table-responsive` class works differently than the other CSS classes for working with tables. Instead of applying this class directly to a `<table>` element, you apply it to a `<div>` element that contains a `<table>` element that has one or more of the Bootstrap table classes applied to it. Then, if the viewport narrows so the data in each cell of the table can't be displayed, horizontal scrolling is added to the table.

When working with tables, you may sometimes need to specify the width of a column. To do that, you can use the `w-size` class to specify the size of a column in rem. For example, you can specify that a column for a table heading should be 25 rem wide by adding a class like this one:

```
<th class="w-25">Department</th>
```

Then, the table uses that width for the column in the heading and the body of the table.

## Common CSS classes for working with HTML tables

Class	Description
<code>table</code>	Provides default styling for an HTML <table> element.
<code>table-bordered</code>	Adds a border around the table and between cells.
<code>table-striped</code>	Adds alternating colors to the table rows.
<code>table-hover</code>	Makes the color of a row change when you hover over it.
<code>table-responsive</code>	Adds horizontal scrolling to the table when the viewport narrows. Applied to a <div> element that contains a <table> element with the table class.
<code>w-size</code>	The width for a column in rem.

### A table with default styling

Department	Phone Number	Extension
General	555-555-5555	1
Customer Service	555-555-5556	2
Billing and Accounts	555-555-5557	3

```
<table class="table">
  <thead>
    <tr><th>Department</th><th>Phone Number</th><th>Extension</th></tr>
  </thead>
  <tbody>
    <tr><td>General</td><td>555-555-5555</td><td>1</td></tr>
    <tr><td>Customer Service</td><td>555-555-5556</td><td>2</td></tr>
    <tr><td>Billing and Accounts</td><td>555-555-5557</td><td>3</td></tr>
  </tbody>
</table>
```

### A table with alternating stripes and borders

Department	Phone Number	Extension
General	555-555-5555	1
Customer Service	555-555-5556	2
Billing and Accounts	555-555-5557	3

```
<table class="table table-striped table-bordered table-hover">...
```

### Description

- You must include the <thead> and <tbody> elements in your table for the Bootstrap table classes to work properly.

Figure 3-11 How to format HTML tables

## How to align and capitalize text

---

The table in figure 3-12 shows some of the Bootstrap CSS classes for working with text. The classes shown here consist of alignment classes and transformation classes.

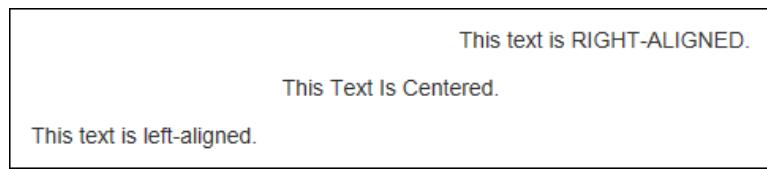
The alignment classes control where the text of an element displays on the page relative to the element that contains it. For instance, assume the `<p>` elements in the example below the table are coded within a `<div>` element that spans six Bootstrap columns. Then, the first `<p>` element is aligned at the right side of the `<div>` element, the second `<p>` element is centered, and the third `<p>` element is aligned at the left side.

The transformation classes control how the text of an element is capitalized. For instance, the text in the first `<span>` element displays in all uppercase letters, even though the text in the HTML is in lowercase letters. Similarly, the text in the second `<span>` element displays with the first letter of each word capitalized, and the text in the third `<span>` element displays in all lowercase letters.

## Common CSS classes for text

Class	Description
<code>text-left</code>	Aligns text to the left within the parent element.
<code>text-right</code>	Aligns text to the right within the parent element.
<code>text-center</code>	Aligns text in the center of the parent element.
<code>text-lowercase</code>	Makes all text in the element lower case.
<code>text-uppercase</code>	Makes all text in the element upper case.
<code>text-capitalize</code>	Capitalizes the first letter of every word in the element.

## Some examples of the text CSS classes



```
<p class="text-right">  
    This text is <span class="text-uppercase">right-aligned</span>.  
</p>  
<p class="text-center">  
    <span class="text-capitalize">This text is centered.</span>  
</p>  
<p class="text-left">  
    This text is <span class="text-lowercase">LEFT-ALIGNED</span>.  
</p>
```

## Description

- The Bootstrap classes for text control the alignment and capitalization for the text.
- The alignment classes, `text-left`, `text-right`, and `text-center`, control where the text of an element is displayed on the page relative to the element that contains it.
- The transformation classes, `text-lowercase`, `text-uppercase`, and `text-capitalize`, control the capitalization for the text of an element.

---

Figure 3-12 How to align and capitalize text

## How to provide context

---

Figure 3-13 presents the context classes that are available with Bootstrap. These classes apply a color to an element depending on its context.

The table in this figure shows the eight main CSS classes for providing context. These classes are available to most elements.

The examples in this figure show how to use some of these context classes. The first example shows how six buttons appear with six different classes applied. Here, each context class is prefixed with *btn-* to indicate that the context is being applied to a button. You can use Visual Studio's IntelliSense feature to find out what prefixes you can use with the context classes.

The second example shows how the success class is applied to the text in a `<p>` element. Here, the text-success class indicates that the color for the success class, which is usually green, should be applied to the text within the `<p>` element.

By contrast, the third example shows how the warning class is applied to the background of a `<p>` element. Here, the bg-warning class indicates that the color for the warning class, which is usually orange, should be applied to the background of the `<p>` element. In addition, this element uses the p-2 class to add .5 rems of padding, and it uses the rounded class to round its corners.

## The context classes available to most elements

Class	Description	Default color
<code>primary</code>	Specifies that the element is a primary element.	Dark blue
<code>secondary</code>	Specifies that the element is a secondary element.	Gray
<code>success</code>	Indicates a successful or positive outcome or action.	Green
<code>info</code>	Indicates neutral information.	Light blue
<code>warning</code>	Indicates something that might need attention.	Orange
<code>danger</code>	Indicates a dangerous or negative outcome or action.	Red
<code>light</code>	Uses a light background.	White
<code>dark</code>	Uses a dark background.	Gray

## Some examples of the context classes applied to buttons

Primary Secondary Success Info Warning Danger

```
<button class="btn btn-primary">Primary</button>
<button class="btn btn-secondary">Secondary</button>
<button class="btn btn-success">Success</button>
<button class="btn btn-info">Info</button>
<button class="btn btn-warning">Warning</button>
<button class="btn btn-danger">Danger</button>
```

## The success class applied to the text of an element

Congratulations! You are now registered.

```
<p class="text-success">Congratulations! You are now registered.</p>
```

## The warning class applied to the background of an element

Warning! Some required fields are empty.

```
<p class="bg-warning p-2 rounded">
    Warning! Some required fields are empty.
</p>
```

## Description

- The context classes are typically combined with a prefix that indicates the element or component being styled. For example, `btn-` specifies button, `text-` specifies text, `bg-` specifies background, and so on. However, the context classes can be applied without a prefix to some elements.

Figure 3-13 How to provide context

## More skills for Bootstrap components

---

Up until now, this chapter has focused on showing how to apply Bootstrap CSS classes to standard HTML elements. However, Bootstrap also provides its own *components*. For example, the jumbotron class presented earlier in this chapter defines a component. This component uses predefined HTML elements and CSS classes to create a user interface element. Now, you'll learn how to use other Bootstrap components such as button groups, icons, badges, breadcrumbs, and alerts.

### How to work with button groups

---

The table in figure 3-14 shows some of the Bootstrap CSS classes that create *button groups*, and the two examples show some of the ways to use these classes. In the first example, four links styled as buttons are grouped together within a `<div>` element that has the `btn-group` class applied to it. This class provides the formatting that makes the buttons in the group look like a menu bar.

In the second example, two `<div>` elements each with two links styled as buttons are coded as button groups. Then, these two `<div>` elements are coded within another `<div>` element that has the `btn-toolbar` class applied to it. This class combines the two button groups. In addition, the first button group uses the `mr-2` class to set its right margin to .5 rems. This adds a little space between these button groups to make it easy to identify each button group.

For assistive technologies such as screen readers to work correctly with button groups, you need to set the role attribute for button groups and toolbars correctly. In this figure, for instance, the first example sets the role attribute to button group, and the second example sets the role attribute to toolbar for the outer `<div>` element and to group for the inner `<div>` elements. In addition, you should specify an `aria-label` attribute that accurately describes each element as shown in this figure.

## Common CSS classes for creating button groups

Class	Description
<code>btn-group</code>	Groups two or more buttons with no padding between them.
<code>btn-toolbar</code>	Combines button groups with appropriate padding between groups.
<code>btn-group-size</code>	Applies sizing to all buttons in a group. Example: <code>btn-group-lg</code> .
<code>btn-group-vertical</code>	Stacks buttons in a group vertically rather than horizontally.

### A basic button group



```
<div class="btn-group" role="group" aria-label="Button group">
  <a href="/" class="btn btn-outline-primary">Home</a>
  <a href="/cart" class="btn btn-outline-primary">Cart</a>
  <a href="/products" class="btn btn-outline-primary">Products</a>
  <a href="/contact-us" class="btn btn-outline-primary">Contact Us</a>
</div>
```

### A toolbar with two button groups



```
<div class="btn-toolbar" role="toolbar" aria-label="Toolbar with groups">
  <div class="btn-group mr-2" role="group" aria-label="First group">
    <a href="/" class="btn btn-outline-primary">Home</a>
    <a href="/Cart" class="btn btn-outline-primary">Cart</a>
  </div>
  <div class="btn-group" role="group" aria-label="Second group">
    <a href="/products" class="btn btn-outline-primary">Products</a>
    <a href="/contact-us" class="btn btn-outline-primary">Contact Us</a>
  </div>
</div>
```

### Description

- A *button group* lets you display a group of buttons.
- For assistive technologies such as screen readers to work correctly with button groups, you need to set the role attribute for button groups and toolbars. In addition, you should specify the aria-label attribute.

---

Figure 3-14 How to work with button groups

## How to work with icons and badges

---

An *icon* is a symbol that decorates or adds meaning to an element. For example, you can add an icon to a button or link to indicate its purpose. Bootstrap 3 included a set of icons called Glyphicons. Bootstrap 4 doesn't include this set of icons. However, you can use the free icons available from Font Awesome with the solid style.

The easiest way to enable the icons that are available from Font Awesome is to include a `<link>` element that links to the CSS file from the Font Awesome website as shown in the first example. Typically, you can place this `<link>` element just below the element that links to the Bootstrap CSS file. The hash code for the integrity attribute is too long to display, but you can copy it from the Font Awesome website. This attribute helps verify that the Font Awesome website has delivered the resource securely.

To use an icon, you include the base `fas` (Font Awesome Solid) class along with the class for the individual icon you want to use. In figure 3-15, the examples use the `fa-home` class for the Home icon and the `fa-shopping-cart` class for the Cart icon.

It's important to note that you can't code the Font Awesome classes directly on other components. For instance, in the first example, you might think that you could code the `fas` and `fa-home` classes on an `<a>` element. However, you must code these classes on an element such as a `<span>` element that's coded within another element such as an `<a>` element. In addition, the `<span>` element that specifies the icon classes can't include any other content such as text.

If you use icons, you should be aware that they don't have any padding by default. As a result, if you're going to use an icon with text as shown in the second example, you need to add space between the icon and the text. To do that, you can add a nonbreaking space entity (`&nbsp;`) to the HTML as shown in this figure. Or, if you prefer, you can use Bootstrap classes or CSS to add space.

When you use icons, you should make them accessible. This is particularly important if an icon doesn't decorate accompanying text. In that case, make sure to include content that reflects the meaning of the icon and that can be read by screen readers. On the other hand, if icons decorate accompanying text, they should be hidden so they don't confuse screen readers.

A *badge* component lets you highlight text within an element. This is illustrated in the second example in this figure. Here, the `<a>` element contains text of "Cart" and a badge with the number 2. Here, the content of the badge is coded in a `<span>` element that uses the `badge` and `badge-primary` classes. As a result, the badge is blue with white text. In addition, the badge collapses so it's hidden if it doesn't contain any text. This is common when you use code to set the content of a badge when an app is running.

The third example shows that you can combine icons and badges within a single element, such as a link that's styled as a button. Here, the first `<span>` element displays the Cart icon, and the second `<span>` element displays the badge that indicates how many items are in the cart. Between these `<span>` elements, the HTML specifies text of "Cart" with a nonbreaking space before and after.

## The URL for the Font Awesome website

<https://fontawesome.com/>

## A typical <link> element that enables Font Awesome icons

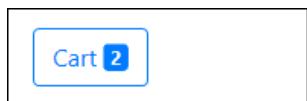
```
<link rel="stylesheet"
      href="https://use.fontawesome.com/releases/v5.8.1/css/all.css"
      integrity="sha-long-hash_code" crossorigin="anonymous">
```

## A button group that includes icons for both of its buttons



```
<div class="btn-group" role="group" aria-label="Button group">
    <a href="/" class="btn btn-outline-primary">
        <span class="fas fa-home"></span> Home
    </a>
    <a href="/cart" class="btn btn-outline-primary">
        <span class="fas fa-shopping-cart"></span> Cart
    </a>
</div>
```

## A button with a badge



```
<a href="/cart" class="btn btn-outline-primary">
    Cart <span class="badge badge-primary">2</span>
</a>
```

## A button with an icon and a badge



```
<a href="/cart" class="btn btn-outline-primary">
    <span class="fas fa-shopping-cart"></span> Cart
    <span class="badge badge-primary">2</span>
</a>
```

## Description

- An *icon* is a symbol that you use to decorate or add meaning to an element.
- A *badge* provides for highlighting text within a component.
- The classes for icons and badges are typically coded in a <span> element that's coded within another element.
- Bootstrap 3 included a set of icons called Glyphicons. Bootstrap 4 doesn't include these icons. However, you can use the free icons available from the Font Awesome solid (fas) style. For more details, please visit the Font Awesome website.

Figure 3-15 How to work with icons and badges

## How to work with button dropdowns

---

Figure 3-16 shows how to create a *button dropdown*. When the user clicks a button dropdown, it displays a menu and lets the user select an item from that menu. To create a button dropdown, you use the CSS classes and the HTML5 data attribute summarized by the table in this figure.

Under this table, the example presents a button dropdown. To start, a `<div>` element uses the `dropdown` class to specify the start and end of the button dropdown. Within this `<div>` element, the code specifies a `<button>` element for the button and another `<div>` element that contains a dropdown menu.

The `<button>` element uses the `dropdown-toggle` class to style the button as a dropdown. This displays a caret to the right of the text on the button. This caret indicates that the button is a dropdown, not a standard button. Then, the `<button>` element sets its `data-toggle` attribute to “dropdown” so the button behaves like a dropdown menu.

The second `<div>` element defines the items in the menu. To start, it uses the `dropdown-menu` class to style this element as a dropdown menu. Then, each `<a>` element within the list uses the `dropdown-item` class to style each link as a menu item. Each of these links specifies the URL to be requested when the user clicks the item.

A button dropdown needs the Popper.js library to work. That’s because the Popper.js library contains the JavaScript that displays the menu when the user clicks on the button. As a result, the page must include the Popper.js library as described in figure 3-4. Alternately, the page can include the `bootstrap.bundle` library since that library includes the Popper.js library.

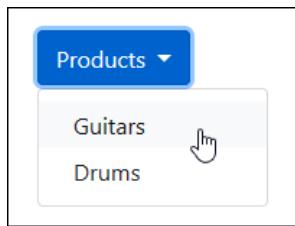
To keep this example simple, the menu uses `<a>` elements for all of the dropdown menu items since that’s the most common scenario. However, it’s possible to include most types of components in a dropdown menu. For example, you can include text boxes that allow the user to enter search terms. Or, you can include forms such as login forms. More commonly, though, you might want to include a checkbox menu item, a radio button group, or a submenu.

To make the dropdown menu work with assistive technologies, you can include the `aria-` attributes shown in this figure. For this to work, the dropdown menu can only contain menu items, checkbox menu items, radio button menu items, radio button groups, and submenus.

## CSS classes and an HTML5 data attribute for creating button dropdowns

Class	Description
<code>dropdown</code>	Marks the start and end of a dropdown list of items.
<code>dropdown-toggle</code>	Applies styling to a button that will function as a dropdown.
<code>dropdown-menu</code>	Applies styling to a dropdown menu.
<code>dropdown-item</code>	Applies styling to the items in a menu.
<code>dropup</code>	Works like the dropdown class but makes the list items drop up.
Attribute	Description
<code>data-toggle</code>	If set to “dropdown”, makes a button dropdown.

### A button dropdown



```
<div class="dropdown">
    <button type="button" class="btn btn-primary dropdown-toggle"
        id="productsDropdown" data-toggle="dropdown"
        aria-haspopup="true" aria-expanded="false">
        Products
    </button>
    <div class="dropdown-menu" aria-labelledby="productsDropdown">
        <a class="dropdown-item" href="/product/list/guitars">Guitars</a>
        <a class="dropdown-item" href="/product/list/drums">Drums</a>
    </div>
</div>
```

### Description

- You can use the classes and attribute shown above to create a *button dropdown*. When you click on a button dropdown, it displays a dropdown menu.
- To create a button dropdown, you must code a button for the button and a dropdown menu that includes the items for the menu.
- A button dropdown needs the Popper.js library to work.
- To make the dropdown menu work with assistive technologies, you can include the `aria-` attributes shown in this figure. For this to work, the dropdown menu can only contain menu items, checkbox menu items, radio button menu items, radio button groups, and submenus.

Figure 3-16 How to work with button dropdowns

## How to work with list groups

---

The table in figure 3-17 shows some of the Bootstrap CSS classes that you can use to create *list groups*. As its name implies, a list group is simply a group of items displayed in a list.

In the first example, the items in an unordered list are styled as a list group. Here, the list-group class is applied to the `<ul>` element, and the list-group-item class is applied to each `<li>` element in the list. It's common to use a list like this when you just want to display information.

If you want to display links or buttons in a list, you should code them within a `<div>` element that uses the list-group class as shown by second example.

Here, the list consists of three `<a>` elements, and the list-group-item class is applied to each of these elements. In addition, the active class is applied to the first link. This highlights the link to show that it's the active link.

It's common to write code that sets the active class when the app is running. That way, you can change the active link depending on the user's actions.

Similarly, you can use the disabled class to disable a link depending on the user's actions.

## Common CSS classes for creating list groups

Class	Description
<code>list-group</code>	Groups two or more items in a list or <div> element.
<code>list-group-item</code>	Styles the individual items in a list group.
<code>active</code>	Highlights the list group item.
<code>disabled</code>	Grays out the list group item.

### A basic list group



```
<ul class="list-group">
  <li class="list-group-item">Guitars</li>
  <li class="list-group-item">Basses</li>
  <li class="list-group-item">Drums</li>
</ul>
```

### Another basic list group with an active item



```
<div class="list-group">
  <a href="/guitars" class="list-group-item active">Guitars</a>
  <a href="/basses" class="list-group-item">Basses</a>
  <a href="/drums" class="list-group-item">Drums</a>
</div>
```

### Description

- The CSS classes for *list groups* let you display a list of items such as links, buttons, and list items. You can also nest a list group within another list group.

Figure 3-17 How to work with list groups

## How to work with alerts and breadcrumbs

The first table in figure 3-18 shows some of the Bootstrap CSS classes and an HTML5 data attribute that you can use to work with *alerts*. In its simplest form, an alert can consist of an element such as a <div> that contains text and has the alert class applied to it. In most cases, though, an alert contains additional components such as a close button and a link like the alert shown in this figure.

There are four things to notice about this alert. First, the alert class is applied to the <div> element that contains the alert. In addition, an alert-context class that provides the appropriate color for the alert is also applied to this <div>, as well as the alert-dismissible class that provides for closing the alert.

Second, within the <div> element, the close class is applied to the HTML button. This class provides for closing the alert. Also, the data-dismiss data attribute of this button tells Bootstrap to dismiss the alert when the button is clicked. Even though this button is coded first in the HTML, it displays on the far right side of the alert.

Third, the text of the close button uses the &times; character entity instead of the letter “X”. This is the HTML5 character entity for the multiplication sign, and it’s often recommended that you use it for close buttons because it lays out better. However, screen readers for the visually impaired may read this character entity as the word “multiplication”. Because of that, you should include an aria attribute such as

```
aria-label="Close Success dialog box"
```

when you use this character entity.

Fourth, the alert-link class is applied to the link in the alert so the link matches the styling of the alert itself. In this case, since the alert uses the success context class, the alert-link class styles the link to match that context.

Figure 3-18 also shows how to create *breadcrumbs*. In this figure, the breadcrumbs provide navigation links that are relative to the user’s current location in a website. Here, the breadcrumbs are coded as an ordered list. In addition, the active class has been applied to the last item in the list. This class indicates the current page, and it displays the text of the item in a different color than the other items. Also, the last item doesn’t contain a link.

It’s common to write code that creates breadcrumbs when the app is running. That way, you can update the breadcrumbs as the user navigates through the website.

Because the breadcrumbs are a navigation component, it’s a good practice to code them within a <nav> element as shown in this figure. In addition, to make breadcrumbs work with assistive technologies, you can include an aria-label attribute on this element as shown in this figure.

## Common CSS classes and an HTML5 data attribute for creating alerts

Class	Description
<code>alert</code>	Wraps text and HTML in a context message area.
<code>alert-context</code>	Applies a context class to an alert. Example: <code>alert-warning</code> .
<code>alert-dismissible</code>	Makes an alert dismissible. The div for the alert should include a button that uses the <code>close</code> class and the <code>data-dismiss</code> attribute.
<code>alert-link</code>	Styles links to match the styling of the alert that contains the link.
<code>close</code>	Provides for closing an alert.
Attribute	Description
<code>data-dismiss</code>	Tells Bootstrap to dismiss the alert.

### A dismissible alert with a link



```
<div class="alert alert-success alert-dismissible">
  <button class="close" data-dismiss="alert">&times;</button>
  Success! <a href="#" class="alert-link">Learn more</a>
</div>
```

## Common CSS classes for creating breadcrumbs

Class	Description
<code>breadcrumb</code>	Makes an ordered list element display inline with separators between items.
<code>breadcrumb-item</code>	Identifies a list item as a breadcrumb.
<code>active</code>	Indicates the current page.

### A breadcrumb with three segments



```
<nav aria-label="breadcrumb">
  <ol class="breadcrumb">
    <li class="breadcrumb-item"><a href="/">Home</a></li>
    <li class="breadcrumb-item"><a href="/Products">Products</a></li>
    <li class="breadcrumb-item active" aria-current="page">Guitars</li>
  </ol>
</nav>
```

## Description

- *Alerts* let you provide context feedback in your app.
- *Breadcrumbs* display navigation links that are relative to the user's current location.

Figure 3-18 How to work with alerts and breadcrumbs

## How to work with navigation bars

Bootstrap provides an easy way to create a responsive menu bar known as a navigation bar that collapses to a dropdown menu on narrower viewports. To do that, you often code one or more nav components within a navbar component.

### How to create navs

The table in figure 3-19 shows some of the Bootstrap CSS classes that you can use to create *nav* components. Nav components provide a way to create a simple menu bar. However, they're more commonly used as part of a navbar component as shown in the next figure.

In this figure, the first example shows how to create a nav component that contains navigation links styled as tabs, and the second example shows how to create the same navigation component with the links styled as pills. These examples show that the only difference between a tab and a pill is in the appearance of the active item.

For both examples, the navigation links are coded within an HTML5 `<nav>` element. This provides accessibility for the items. If for some reason you need to put a nav component within an element other than a `<nav>` element, you should include the `role="navigation"` attribute on that element.

To create a nav component, it's common to use an HTML `<ul>` element that contains `<li>` elements that contain `<a>` elements as shown in the third example. However, when you use Bootstrap, you don't need to include the `<ul>` or `<li>` elements. Instead, you can just apply the `nav-item` and `nav-link` classes directly to the `<a>` element as shown by the first two examples in this figure.

You can also add button dropdowns to a nav component. To do that, you just nest the code for the button dropdown within the nav component. Then, you can change the styling for the button so it looks good in the navigation bar. For example, you may want to use an outline style for the button when you place it in a navigation bar.

## Common CSS classes for creating navs

Class	Description
<code>nav</code>	Groups two or more nav items.
<code>nav-tabs</code>	Styles the nav items in a single line with the active item displayed as a tab.
<code>nav-pills</code>	Styles the nav items in a single line with the active item displayed as a pill.
<code>nav-item</code>	Identifies a nav item such as an <code>&lt;li&gt;</code> element that contains a nav link. However, if you want, you can code nav links outside of an <code>&lt;li&gt;</code> element. In that case, you may need to use this class to identify the link as a nav item.
<code>nav-link</code>	Specifies that the nav item is a link.
<code>active</code>	Styles the active nav item or link differently than the other nav items.

### Nav links styled as tabs



```
<nav class="nav nav-tabs">
  <a class="nav-item nav-link active" href="/">Home</a>
  <a class="nav-item nav-link" href="/products">Products</a>
  <a class="nav-item nav-link" href="/cart">Cart</a>
</nav>
```

### The same nav links styled as pills



```
<nav class="nav nav-pills">
  <a class="nav-item nav-link active" href="/">Home</a>
  <a class="nav-item nav-link" href="/products">Products</a>
  <a class="nav-item nav-link" href="/cart">Cart</a>
</nav>
```

### A more verbose way of coding the same nav links

```
<ul class="nav nav-pills">
  <li class="nav-item">
    <a class="nav-link active" href="/">Home</a>
  </li>
  <li class="nav-item">
    <a class="nav-link" href="/products">Products</a>
  </li>
  <li class="nav-item">
    <a class="nav-link" href="/cart">Cart</a>
  </li>
</ul>
```

### Description

- You can use the CSS classes for `navs` to create tabs and pills. The difference between a tab and a pill is in the appearance of the active item.

Figure 3-19 How to create navs

## How to create navbars

---

The Bootstrap *navbar* component creates a responsive menu bar that collapses to a dropdown menu on narrower viewports. The table in figure 3-20 presents some of the Bootstrap CSS classes and HTML5 data attributes that you can use to create a navbar. To get a navbar to work correctly, you don't need to understand exactly what each class and attribute does. Instead, you can often cut and paste code for an existing navbar and modify it as necessary. That's why this figure focuses on the most important Bootstrap classes for creating navbars.

This figure begins by showing the same navbar at two different viewport widths. Here, a wide screen displays the brand and all the links in the menu bar across the top of the page. By contrast, a small screen only displays the brand and a toggle button that you can use to display and hide the menu bar.

After showing this navbar, this figure shows the HTML that creates it. To start, the navbar is coded within a `<nav>` element for accessibility. Then, four classes are applied to the `<nav>` element. The `navbar` class identifies it as a navigation bar, the `navbar-expand-md` class expands the navbar on medium and large screens, the `navbar-dark` class sets the color scheme for a dark background, and the `bg-primary` sets the background color to the primary color, which is usually dark blue.

Within the `<nav>` element, the first `<a>` element uses the `navbar-brand` class to style the text for the brand with a larger font than the other navigation links. The brand is typically text or an image that displays the home page when clicked. By default, it's displayed at the left side of the navbar.

After the brand link, a `<button>` element defines the toggle button that's displayed when the navbar is collapsed. By default, this button is displayed at the right side of the navbar. This button uses the `navbar-toggler` class and the `data-toggle` and `data-target` attributes. This class and these attributes provide for collapsing the navbar and displaying and hiding the navigation links on small screens. In addition, this button uses the `aria-` classes that provide for accessibility.

After the toggle button, the nested `<nav>` element contains the links for the navbar. This element uses the `collapse` and `navbar-collapse` classes so the navbar collapses properly on small screen sizes. It also includes an `id` attribute whose value is identified by the `data-target` attribute of the element for the toggle button.

The links for the navbar shown here are coded within two `<div>` elements. The `navbar-nav` class is applied to both of these elements. In addition, the `navbar-right` class is applied to the second `<div>` element so the link it contains is aligned at the right side of the navbar. For this to work, the first `<div>` element must use the `mr-auto` class to allow the right margin to be set automatically.

## Common CSS classes and HTML5 data attributes for creating navbars

Class	Description
<code>navbar</code>	Creates a responsive navigation bar that collapses in smaller viewports.
<code>navbar-expand-size</code>	Sets the minimum size for the navbar to be expanded.
<code>navbar-light-or-dark</code>	Sets the color scheme for a light or dark background color.
<code>navbar-brand</code>	Identifies the brand for your navbar.
<code>navbar-toggler</code>	Identifies and styles the toggler button.
<code>navbar-collapse</code>	Identifies and styles the parts of the navbar that collapse.
<code>collapse</code>	Collapses the navbar until the user clicks on the toggler button.
<code>navbar-nav</code>	Identifies and styles part of a navbar.
<code>navbar-alignment</code>	Aligns the nav items to the right or the left. Example: <code>navbar-right</code> .
Attribute	Description
<code>data-toggle</code>	If set to "collapse", makes a navbar collapsible.
<code>data-target</code>	Identifies the HTML element that will be changed.

### A navbar expanded on a wide screen and collapsed on a small screen



### The code

```
<nav class="navbar navbar-expand-md navbar-dark bg-primary">
    <a class="navbar-brand" href="/">My Guitar Shop</a>
    <button class="navbar-toggler" type="button"
        data-toggle="collapse" data-target="#navbarSupportedContent"
        aria-controls="navbarSupportedContent" aria-expanded="false"
        aria-label="Toggle navigation">
        <span class="navbar-toggler-icon"></span>
    </button>
    <nav class="collapse navbar-collapse" id="navbarSupportedContent">
        <div class="navbar-nav mr-auto">
            <a class="nav-item nav-link active" href="/">Home</a>
            <a class="nav-item nav-link" href="/products">Products</a>
            <a class="nav-item nav-link" href="/about">About</a>
        </div>
        <div class="navbar-nav navbar-right">
            <a class="nav-item nav-link" href="/cart">
                <span class="fas fa-shopping-cart"></span>&ampnbspCart&ampnbsp
                <span class="badge badge-primary">2</span>
            </a>
        </div>
    </nav>
</nav>
```

Figure 3-20 How to create navbars

## How to position navbars

---

When you code a navbar, you can code it as the first element in the `<body>` element. In that case, the navbar stretches across the entire viewport. However, you can also nest the navbar within a container element such as a `<div>` element. This causes the navbar to span the width of the container element.

In the previous figure, the navbar is coded as the first element of the `<body>` element. As a result, it's positioned at the top of the screen and stretches across the entire viewport. However, when the user scrolls down, the navbar will scroll off the screen. In many cases, that's how you want your navbars to work. In other cases, you may want your navbar to always be displayed at the top of the screen, even when the user scrolls down. Or, you may want to display a navbar at the bottom of the screen. To do that, you can use the skills shown in figure 3-21.

The table in this figure shows some of the Bootstrap CSS classes that you can use to create fixed navbars. Then, this figure shows a navbar that's fixed at the top of the screen and the code that's used to create it. Here, the `<nav>` element that defines the navbar component is coded as the first element of the `<body>` element. As a result, it spans the entire width of the viewport.

In this figure, the `<nav>` element includes the `fixed-top` class. Otherwise, the rest of the HTML is the same as for the `<nav>` element shown in the previous figure.

When fixing a navbar to the top of the screen, you typically need to adjust the top margin of the `<body>` element. If you don't, the navbar covers the content at the top of the page and makes it unreadable. In this figure, for example, the custom CSS adds 70 pixels to the top margin. Of these pixels, 50 accommodate the height of the navbar and an additional 20 pixels provide space between the page's content and the navbar.

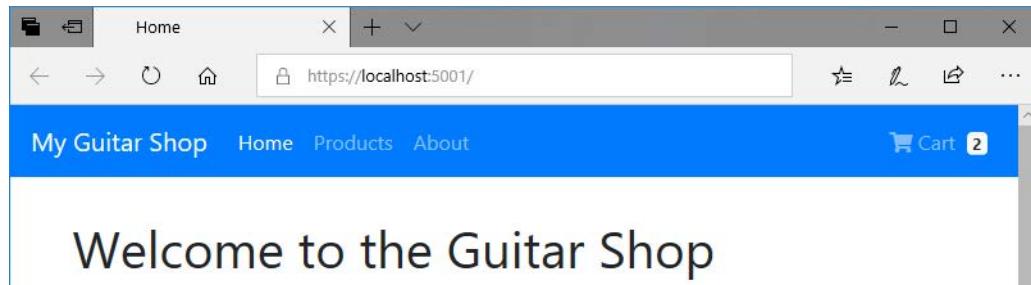
If you want, you can easily modify the HTML and CSS code shown in this figure so it fixes the navbar to the bottom of the screen. First, edit the `<nav>` element so it uses the `fixed-bottom` class instead of the `fixed-top` class. Then, edit the custom CSS class so it sets the bottom margin to 70px. This should position the navbar at the bottom of the page as shown by the second screen in this figure.

Although the examples in this figure and the previous figure should get you off to a good start, there's a lot more you can do with navbars. For example, you can add brand images, buttons, search forms, and more. For more information, please see the Bootstrap documentation.

## More CSS classes for positioning navbars

Class	Description
<code>fixed-top</code>	Makes the navbar stay at the top of the screen even when the user scrolls. Will overlay other content unless you add enough margin to the top of the body.
<code>fixed-bottom</code>	Makes the navbar stay at the bottom of the screen even when the user scrolls. Will overlay other content unless you add margin to the bottom of the body.

### A navbar that's fixed at the top of the screen



#### The HTML that displays the navbar

```
<body>
    <nav class="navbar navbar-expand-md navbar-dark bg-primary fixed-top">
        <!-- navbar items go here -->
    </nav>
    <div class="container">
        <!-- container items go here -->
    </div>
</body>
```

#### The CSS that sets the top margin

```
body {
    margin-top: 70px;
}
```

### A navbar that's fixed at the bottom of the screen

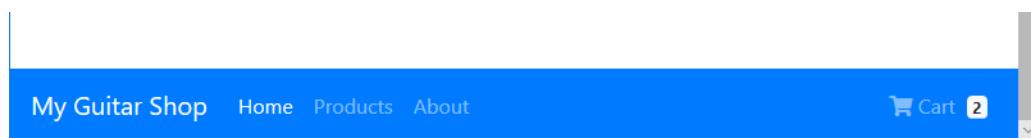


Figure 3-21 How to position navbars

## Perspective

---

Now that you've completed this chapter, you know the right way to use Bootstrap in an ASP.NET Core MVC web app. That means using HTML for the content and structure of a page, using Bootstrap for responsive web design, and using a custom CSS style sheet whenever necessary. That separates the content and structure of each page from its formatting. And that makes it easier to create and maintain a mobile-friendly web app.

Of course, there's a lot more to Bootstrap than what's presented in this chapter. That includes more classes and components, as well as best practices for making Bootstrap accessible. Fortunately, the documentation for Bootstrap is excellent. As a result, if you need to learn more about Bootstrap, you can start by checking out the official Bootstrap documentation.

## Terms

---

responsive web design	rem unit
framework	jumbotron
Bootstrap	button group
Library Manager	icon
LibMan	badge
minified	button dropdown
viewport	list group
meta tag	alert
boxed layout	breadcrumbs
full width layout	nav
Bootstrap CSS classes	navbar
Bootstrap components	

## Summary

---

- *Responsive web design* helps you create web apps that look good and are easy to use on all screen sizes.
- One way to create a responsive web design is to use a *framework* like *Bootstrap*, which uses CSS and JavaScript to make your web pages automatically adjust to different screen sizes.
- With Visual Studio, you can use the *Library Manager* tool, also known as *LibMan*, to add client-side libraries such as Bootstrap and jQuery to a project.
- A *minified* version of a CSS or JavaScript library has removed unnecessary characters such as spaces and indentation. This improves load time but makes it more difficult for humans to read.

- The *viewport* is the part of the page that is visible to viewers. The *viewport meta tag* controls the width of the viewport.
- Bootstrap uses a grid system based on containers, rows, and columns.
- In a *boxed layout*, a container is centered in the screen and has a specific width in pixels based on the viewport's width. In a *full width layout*, a container is always the same width as the viewport.
- *Bootstrap CSS classes* let you style HTML elements such as buttons, images, and tables.
- *Bootstrap components* let you create user interface elements such as jumbotron, button groups, and navbars.
- Bootstrap CSS classes often use *rem units* for sizing. A rem unit is similar to the more traditional em unit, which is commonly used in CSS.
- A *jumbotron* displays a large grey box with rounded corners to highlight content.
- A *button group* lets you display a group of buttons.
- An *icon* is a symbol that you can use to decorate or add meaning to an element.
- A *badge* provides for highlighting text within a component.
- A *button dropdown* is a button that displays a dropdown menu when it's clicked.
- A *list group* displays a list of items such as list items, links, and buttons.
- *Alerts* let you provide contextual feedback in your app.
- *Breadcrumbs* display navigation links for the user's current location.
- A *nav* component creates a simple menu bar.
- A *navbar* component can be used to create a responsive menu bar that collapses to a dropdown menu on narrower viewports.

## Exercise 3-1 Work with the Bootstrap classes in the Future Value app

In this exercise, you'll review the Bootstrap files and classes that are used by the responsive version of the Future Value app. Then, you'll modify some of the classes to see how those changes affect the app.

### Open the Future Value app and set up the client-side libraries

1. Open the Ch03Ex1FutureValue web app that's in the ex\_starts folder.
2. In the Solution Explorer, expand the wwwroot folder and the lib folder to view the files for the client-side libraries. Note that these files include the Bootstrap CSS library.
3. Use the procedure in figure 3-3 to display the LibMan file and note the version numbers and file paths for the Bootstrap CSS library and the other client-side libraries.
4. Open the \_Layout.cshtml file and note that the first <link> element specifies a path that leads to the Bootstrap CSS library. Also, note that the <script> elements specify paths that lead to client-side JavaScript libraries.

### Review and modify the Bootstrap classes for the grid system

5. Open the Index.cshtml file and review the Bootstrap classes that are applied to various elements in this page.
6. Run the app. With the text fields empty, click the Calculate button so the validation messages are displayed.
7. Narrow and widen the browser window to see how the layout responds. Pay attention to when the labels and controls change from being displayed side-by-side to being stacked. When you're done, close the browser.
8. Locate the Bootstrap column classes and change the size from small (col-sm-size) to extra small (col-size). Then, repeat steps 6 and 7 to see how the changes you've made impact the page.
9. If you're feeling adventurous, experiment with using the column classes so the elements display the way you want them on all screen sizes.

### Experiment with some other the Bootstrap classes

10. Modify the HTML for the Clear button so it uses the Bootstrap btn-secondary class. This should display the Clear button with a gray background.
11. Remove the rounded class from the image. This should display the image with its original square corners.
12. Before the form element, add a dismissible alert like the one shown in figure 3-18. This alert should say, "This site uses cookies". In addition, it should have a link that says, "View cookie policy". However, the link doesn't need to do anything.
13. Run the app and test your changes. When your changes are working correctly, close the app.

## Exercise 3-2 Work with the Bootstrap classes in the Movie List app

In this exercise, you'll run the Movie List app that's presented in the next chapter. In addition, you'll modify some of the Bootstrap classes that it uses.

### Open the Movie List app, review its Bootstrap files, and run it

1. Open the Ch03Ex2MovieList web app that's in the ex\_starts folder.
2. In the Solution Explorer, expand the wwwroot folder and the lib folder to view the files for the client-side libraries including Bootstrap.
3. Open the \_Layout.cshtml file and note that it includes a <link> element that includes a path to the Bootstrap CSS library. However, it doesn't link to any other client-side libraries. That shows that this app only needs the Bootstrap CSS library.
4. Press Ctrl+F5 to run this app. If you get an error message that says the app cannot open the Movies database, you need to create the database as described in appendix A (Windows) or B (macOS).
5. When the app starts, it should display a list of movies in a table. Note how the rows in the table alternate from white to a light gray. Also, note the border around the table and its cells.

### Review and modify the Bootstrap classes

6. In the Views/Home folder, open the Index.cshtml file and review the Bootstrap classes that are applied to various components on this page.
7. Locate the Bootstrap classes for the table. Remove the classes that make it bordered and striped.
8. Run the app. The table should be displayed using Bootstrap's default settings for a table.
9. Add the table-hover class to the table.
10. Run the app again. If you hover the mouse pointer over a row, it should change the color of the row.
11. Modify the Edit and Delete links so they use the btn and btn-primary classes.
12. Run the app. The Edit and Delete links should now appear as buttons.
13. In the Views/Movie folder, open the Edit.cshtml file and edit the code for the Cancel link so it uses the btn and btn-outline-secondary classes.
14. Run the app and click on an Edit button. The Cancel button on the Edit Movie page should now appear white with a gray outline.
15. Repeat steps 13 and 14 for the Delete.cshtml file. This should change the appearance of the Cancel button on the Delete Movie page.
16. Close the app.



# 4

## How to develop a data-driven MVC web app

In this chapter, you'll learn how to build a multi-page, data-driven MVC web app. This app displays data that's stored in a database and allows users to add, update, and delete that data. To do that, this app uses Microsoft's Entity Framework (EF) to work with two related tables in a SQL Server database.

<b>An introduction to the Movie List app.....</b>	<b>130</b>
The pages of the app .....	130
The folders and files of the app .....	132
<b>How to use EF Core.....</b>	<b>134</b>
How to add EF Core to your project.....	134
How to create a DbContext class.....	136
How to seed initial data .....	138
How to add a connection string .....	140
How to enable dependency injection .....	140
How to use migrations to create the database .....	142
<b>How to work with data.....</b>	<b>144</b>
How to select data.....	144
How to insert, update, and delete data .....	146
How to view the generated SQL statements.....	146
<b>The Movie List app .....</b>	<b>148</b>
The Home controller.....	148
The Home/Index view .....	148
The Movie controller .....	150
The Movie/Edit view .....	152
The Movie/Delete view .....	154
<b>How to work with related data.....</b>	<b>156</b>
How to relate one entity to another .....	156
How to update the DbContext class and the seed data.....	158
How to use migrations to update the database .....	160
How to select related data and display it on the Movie List page .....	162
How to display related data on the Add and Edit Movie pages .....	164
<b>How to make user-friendly URLs .....</b>	<b>166</b>
How to make URLs lowercase with a trailing slash .....	166
How to add a slug.....	168
<b>Perspective .....</b>	<b>170</b>

## An introduction to the Movie List app

---

This chapter shows how to create web apps that work with data that's stored in a database. To illustrate, it presents a simple Movie List app that stores data about movies in a SQL Server LocalDB database.

### The pages of the app

---

Figure 4-1 shows the four pages of the Movie List app. To start, the Movie List page displays the movie data in a table. Above the table, the page displays a link that users can click when they want to add a new movie. Within the table, each row displays two links users can click when they want to edit or delete an existing movie.

If the user clicks the Add New Movie link, the app displays the Add Movie page. This page displays a form with text boxes that let the user enter the name, year, and rating for a movie. In addition, it displays a drop-down list that allows users to select a genre for the movie. The bottom of this form displays two buttons. As you might expect, the Add button adds the new movie to the database, and the Cancel button cancels the operation and displays the Movie List page.

If the user clicks one of the Edit links in the table on the Movie List page, the app displays the Edit Movie page. This page also contains a form with the same fields as the Add Movie page. However, it uses an Edit button to save changes to the database, not an Add button.

In fact, since the Add and Edit pages require the same fields, they both use the same view. But, they have different URLs and button names, and the Edit Movie page has data while the Add Movie page is blank. Later in this chapter, you'll see how this works.

If the user clicks one of the Delete links in the table on the Movie List page, the app displays the Delete Movie page. This page simply asks users to confirm that they really want to delete the selected movie. If they do, they click the Delete button to delete the movie from the database. If they don't, they click the Cancel button to return the user to the Movie List page.

Each of these pages communicates with a database that stores movie data. The Movie List page selects data, the Add Movie page inserts data, the Edit Movie page selects and then updates data, and the Delete Movie page selects and then deletes data. You'll learn how this works as you read this chapter.

## The pages of the Movie List app

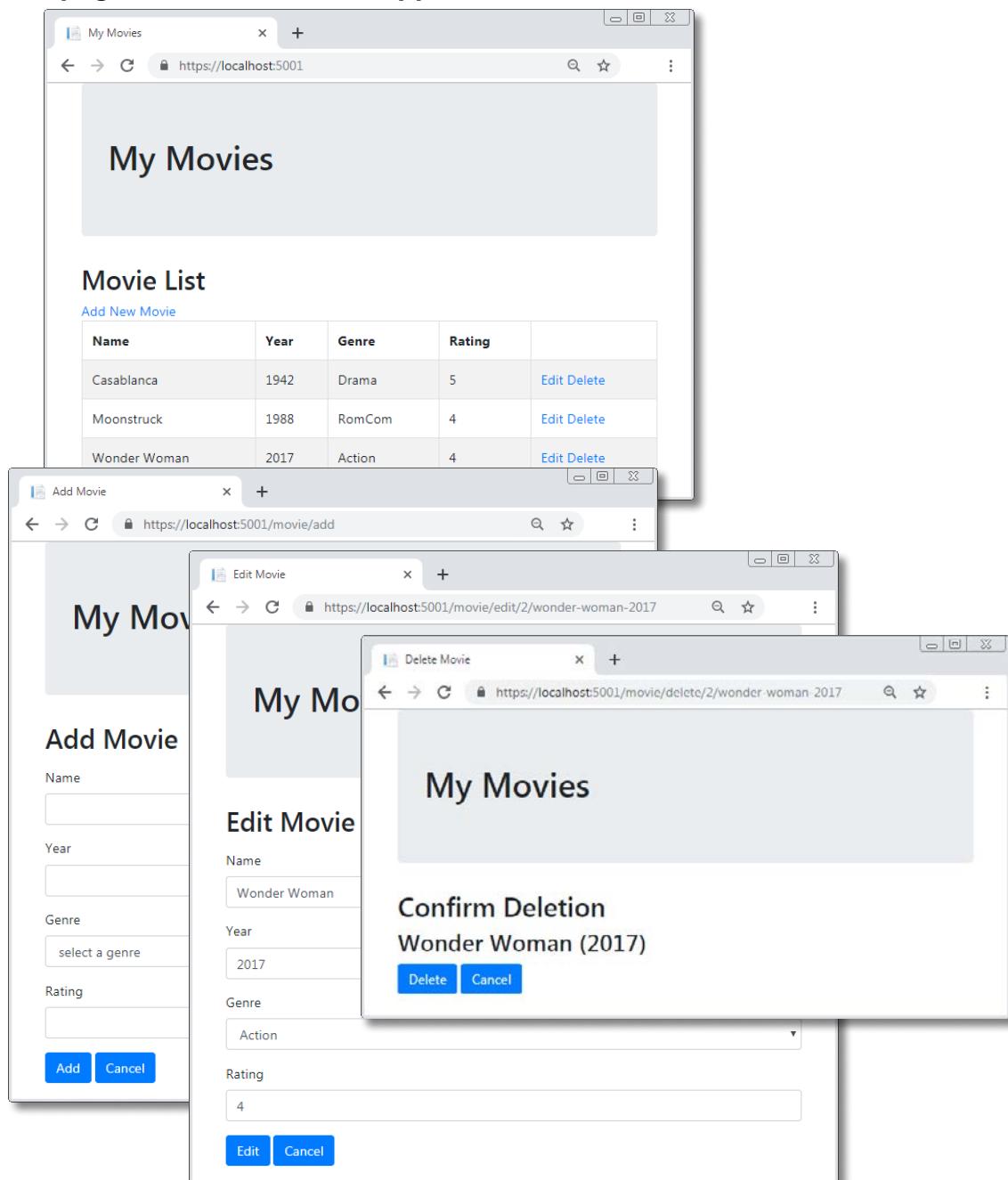


Figure 4-1 The pages of the Movie List app

## The folders and files of the app

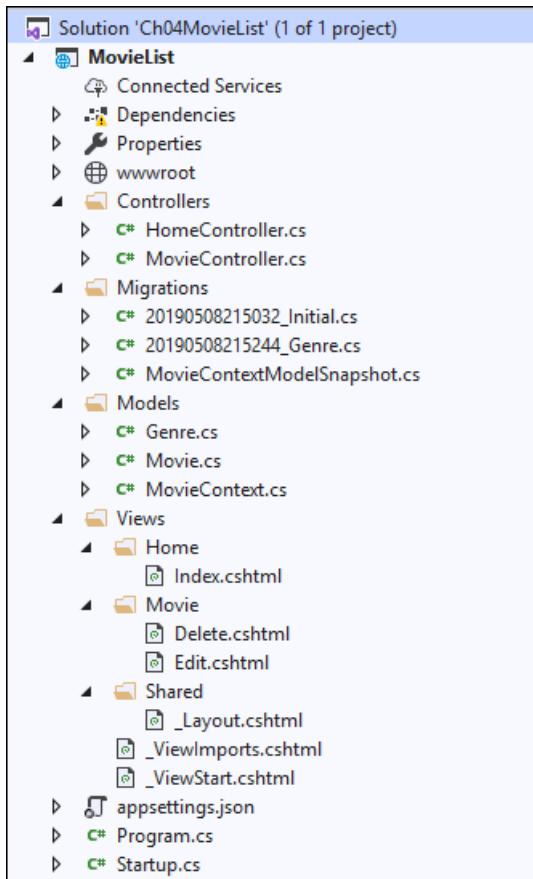
---

Before you learn the details of working with a database, you should learn a little more about the structure of the app presented in this chapter. Figure 4-2 presents the Solution Explorer window for the Movie List app. This window displays the folders and files found in the app's root directory.

The Solution Explorer contains the Models, Views, and Controllers folders that you would expect in an MVC app. In addition, it contains a folder named Migrations that you haven't seen before. This folder contains files that are used to create and update the underlying database. You'll learn how to work with the files in the Migrations folder as you progress through this chapter.

The last table in this figure describes two of the files in the Solution Explorer. As you'll see in this chapter, both of these files are important for apps that access a database.

## The Solution Explorer for the Movie List app



## Folders in the Movie List app

Folder	Contains
Models	The files that define the Movie and Genre classes. Also, a MovieContext class that handles communication with the database.
Views	The view files associated with action methods in the Home and Movie controllers. Also, the Shared folder contains a layout that's used by the views.
Controllers	The files that define the Home controller that gets a list of movies, and the Movie controller that handles adding, editing, and deleting movies.
Migrations	Files that create a SQL Server LocalDB database and seed it with initial data.

## Files in the Movie List app

File	Description
appsettings.json	A configuration file that stores static configuration settings such as database connection strings.
Startup.cs	Code that runs when the app starts and configures the middleware for the app, including the objects that controllers need to communicate with the database.

Figure 4-2 The folders and files of the Movie List app

## How to use EF Core

---

*Entity Framework (EF) Core* is an *object-relational mapping (ORM)* framework that allows you to work with the objects of a database in code. In the next few figures, you'll learn how to code classes that define the structure of a database. Then, you'll learn how to create a database from this code.

### How to add EF Core to your project

---

Prior to .NET Core 3.0, EF Core and the tools to work with it were included with ASP.NET Core by default. However, with .NET Core 3.0 and later, you must manually add EF Core and the EF Core Tools to your project. To do that with Visual Studio, you need to find and install the correct NuGet packages as described in figure 4-3.

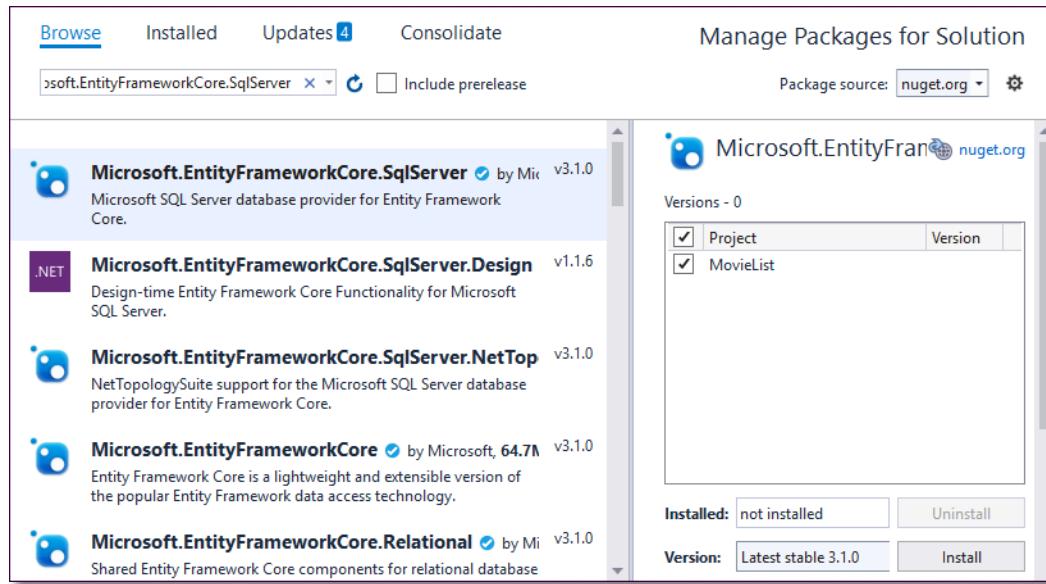
If you try to install a version of EF Core or EF Core Tools that's newer than the version of .NET Core you installed, you might get errors. That's why step 4 says to select the version that matches your version of .NET Core.

In addition, there are many different versions of EF. This means you need to make sure to select the right NuGet package. For example, when you first click on the Browse link in the NuGet Package Manager, you might see a package named Entity Framework in the left-hand panel. However, this is most likely an older version known as EF6. If you install EF6, almost none of the EF code in this chapter will work correctly. So, be sure to carefully follow the instructions presented here, especially the searches described in steps 2 and 7. That way, you can be sure to find the correct EF Core NuGet packages for .NET Core.

## How to open the NuGet Package Manager

- Select Tools → Nuget Package Manager → Manage NuGet Packages for Solution.

## The NuGet Package Manager



## How to install the EF Core and EF Core Tools NuGet packages

1. Click the Browse link in the upper left of the window.
2. Type “Microsoft.EntityFrameworkCore.SqlServer” in the search box.
3. Click on the appropriate package from the list that appears in the left-hand panel.
4. In the right-hand panel, check the project name, select the version that matches the version of .NET Core you’re running, and click Install.
5. Review the Preview Changes dialog that comes up and click OK.
6. Review the License Acceptance dialog that comes up and click I Accept.
7. Type “Microsoft.EntityFrameworkCore.Tools” in the search box.
8. Repeat steps 3 through 6.

## Description

- With .NET Core 3.0 and later, you must manually add EF Core and EF Core Tools to your project.

Figure 4-3 How to add EF Core to your project

## How to create a DbContext class

---

The most common way to work with EF Core is to code your model classes first. Then, you can use EF Core to generate a database from those classes. This approach is known as *EF Code First*.

Figure 4-4 begins by presenting a table that describes three classes provided by EF Core. The `DbContext` class is the primary class for communicating with a database. The `DbContextOptions` class provides configuration information to the `DbContext` class. And the `DbSet` class represents a collection of model classes, also known as *entity classes*, or *domain model classes*, that map to a database table.

The first code example in this figure shows a `DbContext` class named `MovieContext`. This class inherits the `DbContext` base class, which is in the `Microsoft.EntityFrameworkCore` namespace.

The `MovieContext` class has a constructor that accepts a `DbContextOptions` object and passes it to the constructor of the base class, which is the `DbContext` class. In a moment, you'll learn how the `Startup.cs` file passes the context options to the constructor.

To enable your `DbContext` class to work with collections of your entity classes, you need to add properties of the `DbSet<Entity>` type. For instance, the `MovieContext` class has a `Movies` property of the `DbSet<Movie>` type. EF Core uses `DbSet` properties like this one to generate database tables. In addition, you can use LINQ to query a `DbSet` property.

The second code example shows the `Movie` class. This class defines four public properties and supplies data validation attributes for three of those properties. When you use the `Required` attribute, the property must have a nullable data type. To make the `int` types for the `Year` and `Rating` properties nullable, you must code a question mark (?) after them. You must also code a question mark after the `string` type for the `Name` property if you enable the nullable compiler option. Otherwise, `string` types are nullable by default.

When you use EF Core to create the database, it uses the `Movie` class to create a database table. As you review this class, there are two things you need to understand about the database creation process.

First, EF Core automatically treats a property with a name of `Id` (or `ID`) or the entity name followed by `Id` (or `ID`) as a primary key. A *primary key* value uniquely identifies an entity. This value is often used to select a specific entity from the database. If the `Id` property is also of the `int` type, EF Core configures the corresponding column in the database as an *identity column*. This means that the database automatically generates a value for new entities, so you don't need to do that in your code. In most cases, that's what you want. If it isn't, you can configure your code so the database doesn't automatically generate this value. You'll learn how to do that in chapter 12.

Second, EF Core uses some of the data validation attributes to configure the database table it creates. For instance, `string` properties typically create columns in the table that can accept nulls. But since the `Name` property shown here is decorated with a `Required` attribute, EF Core configures the `Name` field in the table to not accept nulls. Again, you'll learn more about how this works in chapter 12.

## Three classes provided by EF Core

Class	Description
<code>DbContext</code>	The primary class for communicating with a database.
<code>DbContextOptions</code>	Stores configuration options for the <code>DbContext</code> object.
<code>DbSet&lt;Entity&gt;</code>	A collection of objects created from the specified entity.

## A MovieContext class that inherits the DbContext class

```
using Microsoft.EntityFrameworkCore;

namespace MovieList.Models
{
    public class MovieContext : DbContext
    {
        public MovieContext(DbContextOptions<MovieContext> options)
            : base(options)
        { }

        public DbSet<Movie> Movies { get; set; }
    }
}
```

## A Movie class with a property whose value is generated by the database

```
using System.ComponentModel.DataAnnotations;

namespace MovieList.Models
{
    public class Movie
    {
        // EF Core will configure the database to generate this value
        public int MovieId { get; set; }

        [Required(ErrorMessage = "Please enter a name.")]
        public string Name { get; set; }

        [Required(ErrorMessage = "Please enter a year.")]
        [Range(1889, 2999, ErrorMessage = "Year must be after 1889.")]
        public int? Year { get; set; }

        [Required(ErrorMessage = "Please enter a rating.")]
        [Range(1, 5, ErrorMessage = "Rating must be between 1 and 5.")]
        public int? Rating { get; set; }
    }
}
```

## Description

- Within the `DbContext` class, you can use `DbSet<Entity>` properties to work with the model classes that map to database tables. These classes are also known as *entity classes*, or *domain model classes*.
- Any property in your entity with a name of Id (or ID) or the entity name followed by Id (or ID) is a *primary key*. If this property is also of the `int` type, the corresponding column is an *identity column* whose value is automatically generated.

Figure 4-4 How to create a `DbContext` class

## How to seed initial data

---

Sometimes you want to include, or *seed*, some initial data in the database tables that EF Core creates. For example, you might want to include a few initial records for test purposes. Or, you might want to include *lookup data* such as a list of states or movie genres that an app can use to look up data that it needs.

The `DbContext` class has an `OnModelCreating()` method that you can override to configure your context. You'll learn more about this in chapter 12. For now, all you need to know is that one of the things you can do when you override this method is seed initial data in your database.

The example in figure 4-5 shows the `MovieContext` class updated to seed initial Movie data. Here, the `OnModelCreating()` method overrides the method in the base class. Within this method, the code passes an array of new `Movie` objects to the `HasData()` method of the `Entity<Movie>()` method of the `ModelBuilder` object. This code runs when the database is created, and it inserts this data into the table that's created from the `Movie` class.

Since the `MovieId` attribute is a primary key and an identity column, EF Core automatically generates a value for the `MovieId` property when new movies are inserted. However, this doesn't apply to seeding initial data in the `DbContext` class. Instead, you need to specifically provide a value for the `MovieId` property of each `Movie` object as shown in this example.

The process of seeding initial data in Entity Framework has gone through many changes. As a result, it's easy to be confused by older examples. To keep this from happening, be sure to search for version 3.0 or higher when you're looking for help or examples online.

As you review this example, note that the `OnModelCreating()` method accepts a `ModelBuilder` object. This `ModelBuilder` object provides an `Entity<T>()` method that returns an `EntityTypeBuilder<T>` object. In this figure, for instance, the `Entity<Movie>()` method returns an `EntityTypeBuilder<Movie>` object. This object, in turn, provides a `HasData()` method that accepts an array of entity objects.

## One method of the DbContext class

Method	Description
<code>OnModelCreating(mb)</code>	Called by the framework when the context is created. You can override it to configure your context.

## The MovieContext class updated to seed initial Movie data

```
namespace MovieList.Models
{
    public class MovieContext : DbContext
    {
        public MovieContext(DbContextOptions<MovieContext> options)
            : base(options)
        { }

        public DbSet<Movie> Movies { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            modelBuilder.Entity<Movie>().HasData(
                new Movie {
                    MovieId = 1,
                    Name = "Casablanca",
                    Year = 1942,
                    Rating = 5
                },
                new Movie {
                    MovieId = 2,
                    Name = "Wonder Woman",
                    Year = 2017,
                    Rating = 3
                },
                new Movie {
                    MovieId = 3,
                    Name = "Moonstruck",
                    Year = 1988,
                    Rating = 4
                }
            );
        }
    }
}
```

### Description

- The DbContext class has an `OnModelCreating()` method that you can override to configure the context.
- The `OnModelCreating()` method accepts a `ModelBuilder` object as an argument. You can use the `Entity().HasData()` method of this object to seed initial data in the database.
- When you use the `HasData()` method to seed data, you need to provide values for the `Id` properties, even the ones that will be configured as identity columns.

---

Figure 4-5 How to seed initial data in the DbContext class

## How to add a connection string

---

A *connection string* is a string that specifies information that an app needs to connect to a database or other data source. Although you can hard-code connection strings in the code for your app, it's generally considered a best practice to store them in a configuration file. That way, if you move the database to another server or make some other change that affects the connection string, you won't have to change the code for the app. Instead, you can simply change the connection string in the configuration file.

For Core MVC apps, you typically use the configuration file named `appsettings.json`. This file is a text file that uses JSON to provide static configuration information for your app.

The first example in figure 4-6 presents an `appsettings.json` file with a connection string named `MovieContext` for a SQL Server LocalDB database named `Movies`. To fit this connection string on the page, it's shown here on two lines. For this string to work correctly, though, it must be coded on one line.

## How to enable dependency injection

---

*Dependency injection* is a design pattern in which the services an object needs are passed to it rather than being hard coded as part of the object. In other words, the dependencies of an object are injected into it. This *decouples* the object from its dependencies and makes it easier to test.

Core MVC uses dependency injection to pass `DbContext` objects to the controllers that need them. To enable this, you can add code to the `Startup.cs` file like the code shown in the second example in figure 4-6.

The `Startup` class shown here begins with a constructor. Within this constructor, the code sets the `Configuration` property to the configuration object that's passed to the constructor. Then, the `Startup` class provides a read-only `Configuration` property that provides a way to get this configuration object. Once you do that, you can use this object to read configuration data such as connection strings.

To enable injection of `DbContext` objects, you need to call the `AddDbContext()` method from within the `ConfigureServices()` method. Typically, you pass this method a lambda expression that creates a `DbContextOptions` object. In turn, this object gets passed to the constructor of the `DbContext` class so it can provide information about the database server. In addition, this lambda expression uses the `Configuration` property of the `Startup` class to get the connection string from the `appsettings.json` file and pass it to the `DbContext` object.

Once you've configured dependency injection in the `Startup.cs` file like this, MVC automatically creates and passes a `DbContext` object to any controller whose constructor has a `DbContext` parameter. In the next few figures, you'll see how this works.

## A connection string in the appsettings.json file

```
{  
    "Logging": {  
        "LogLevel": {  
            "Default": "Warning"  
        }  
    },  
    "AllowedHosts": "*"  
    "ConnectionStrings": {  
        "MovieContext": "Server=(localdb)\\mssqllocaldb;Database=Movies;  
                        Trusted_Connection=True;MultipleActiveResultSets=true"  
    }  
}
```

## Code that enables dependency injection for DbContext objects

```
using Microsoft.Extensions.Hosting;  
using Microsoft.EntityFrameworkCore;  
using MovieList.Models;  
...  
public class Startup  
{  
    public Startup(IConfiguration configuration)  
    {  
        Configuration = configuration;  
    }  
  
    public IConfiguration Configuration { get; }  
  
    public void ConfigureServices(IServiceCollection services)  
    {  
        ...  
        services.AddDbContext<MovieContext>(options =>  
            options.UseSqlServer(  
                Configuration.GetConnectionString("MovieContext")));  
    }  
    ...  
}
```

## Description

- A *connection string* contains information that an app needs to connect to a database or other data source.
- The appsettings.json file stores configuration settings for your app.
- *Dependency injection* is a design pattern in which the services an object needs are passed to it rather than being hard coded as part of the object. Core MVC uses dependency injection to pass DbContext objects to the controllers that need them.
- To inject DbContext objects into a controller, call the AddDbContext() method of the services object that's passed to the ConfigureServices() method of the Startup class.
- The lambda expression passed to AddDbContext() creates a DbContextOptions object with information about what database server and connection string to use.
- Once you've configured dependency injection, MVC automatically creates and passes a DbContext object to any controller whose constructor has a DbContext parameter.

---

Figure 4-6 How to add a connection string and enable dependency injection

## How to use migrations to create the database

Once you've coded your DbContext class and your entity classes, you need to tell EF Core to translate them into a database. With Windows, you can use Visual Studio to do that by executing PowerShell commands in the Package Manager Console (PMC) window.

If you run an app before you create its database, you'll get an error message that indicates that app can't open the database. For example, if you try to run the Movie List app presented in this chapter, and you haven't created its database, you'll get an error message that says:

```
Cannot open database "Movies" requested by the login.
```

Figure 4-7 explains how to use the PMC window to create a database from your code. Before you can do that, you need to add a connection string and enable dependency injection as described in the previous figure.

The first PowerShell command you execute, Add-Migration, creates a migration file in a folder named Migrations. If the Migrations folder doesn't already exist in your project, it's created the first time you run the Add-Migration command.

A *migration* is a file that contains C# code for creating, modifying, or deleting database objects. More specifically, it has an Up() method with C# code that implements a migration, and it has a Down() method with C# code that rolls back a migration. Both of these methods accept a MigrationBuilder object to do their work.

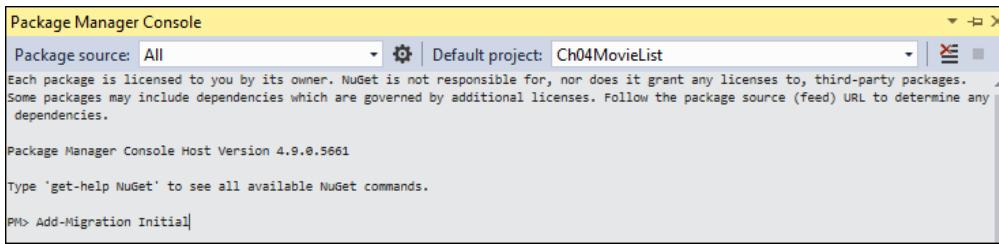
This figure shows the code in the Up() method of the Initial migration file that's created by the Add-Migration command. This method creates a table based on the Movie entity and seeds it with initial data. In general, it's a good idea to review every migration file you generate. That way, you can make sure your DbContext and entity class code has been translated the way you want.

The second PowerShell command you run, Update-Database, applies the migration file. That is, it executes the Up() method of the migration file and creates, modifies, or deletes database objects. But first, if the database with the name specified in the connection string doesn't yet exist, the Up() method creates it. If you're using SQL Server LocalDB, you can make sure the database was created correctly by using the procedure described at the bottom of this figure to view the database.

If you don't understand the details of how this works, don't worry! You'll learn more about migrations in chapter 12.

If you're using Visual Studio on macOS, or if you're using Visual Studio Code instead of Visual Studio, the procedure shown in this figure won't work for you. Instead, you'll need to use the CLI to execute the migration commands as described in chapter 17.

## The Package Manager Console (PMC) window in Visual Studio



### How to open the Package Manager Console window

- Select the Tools→NuGet Package Manager→Package Manager Console command.

### How to create the Movies database based on your code files

- Make sure the connection string and dependency injection are set up.
- Type “Add-Migration Initial” in the PMC at the command prompt and press Enter.
- Type “Update-Database” at the command prompt and press Enter.

### The code in the Up() method of the Initial migration file

```
protected override void Up(MigrationBuilder migrationBuilder)
{
    migrationBuilder.CreateTable(
        name: "Movies",
        columns: table => new {
            MovieId = table.Column<int>(nullable: false)
                .Annotation("SqlServer:ValueGenerationStrategy", "SqlServerValueGenerationStrategy.IdentityColumn"),
            Name = table.Column<string>(nullable: false),
            Year = table.Column<int>(nullable: false),
            Rating = table.Column<int>(nullable: false)
        },
        constraints: table => {
            table.PrimaryKey("PK_Movies", x => x.MovieId);
        });
}

migrationBuilder.InsertData(
    table: "Movies",
    columns: new[] { "MovieId", "Name", "Rating", "Year" },
    values: new object[] { 1, "Casablanca", 5, 1942 });

// code that inserts the other two movies
}
```

### How to view the database once it's created

- Choose the View→SQL Server Object Explorer command in Visual Studio.
- Expand the (localdb)\MSSQLLocalDB node, then expand the Databases node.
- Expand the Movies node, then expand the Tables node.
- To view the table columns, expand a table node and then its Columns node.
- To view the table data, right-click a table and select the ViewData command.

---

Figure 4-7 How to use migrations to create the database

## How to work with data

---

Now that you know how to create a database and seed it with data, you're ready to learn how to work with that data. In the next two figures, you'll learn how to use *Language-Integrated Query (LINQ)* to select data, and you'll learn how to use methods of the DbSet and DbContext classes to insert, update, and delete data.

In these figures, the examples use a private property named context to get a MovieContext object. This property is created by a controller class as shown in figure 4-10.

### How to select data

---

There are two steps to using LINQ and EF Core to select data from a database. First, you build a *query expression*. Then, you execute that query expression at the database.

The first table in figure 4-8 presents four methods for working with query expressions. You use the first two to build query expressions. These methods accept a lambda expression and have a return type of IQueryble<T>. The Where() method allows you to filter data according to the logic of the lambda expression, and the OrderBy() method allows you to sort data.

The third and fourth methods in this table execute query expressions and retrieve data. The FirstOrDefault() method returns either the first entity found or null if nothing is found. The ToList() method returns a List<T> object that stores zero or more entities of the specified type.

The second table in this figure presents a method of the DbSet<Entity> class that you can use to retrieve an entity. Like the FirstOrDefault() method, the Find() method returns either the first entity found or null if nothing is found. However, with Find(), you can only search by primary key.

The code examples below the tables show how to build and execute query expressions. The first example shows how to import the LINQ namespace. The second example uses the Movies property and the OrderBy() method to build a query and store it in an IQueryble object. And the third example uses the ToList() method to execute that query and return a List of Movie objects that contains the data in the result set.

The fourth example shows how to combine the second and third examples into a single statement. To do that, this example chains the two method calls together. In addition, it shows how to use the dynamic var type. As a result, the compiler determines the data type for the variable at runtime. This results in code that's shorter and more flexible. The only downside is that the data type for the movies variable isn't as clear.

The fifth and sixth examples show how to build a more complex query expression. Here, the fifth example chains methods calls, and the sixth example uses multiple lines of code.

## A DbContext property that's used in the following examples

```
private MovieContext context { get; set; }
```

## LINQ methods that build or execute a query expression

Method	Description
<code>Where(lambda)</code>	Filters the entities according to the logic of the lambda expression.
<code>OrderBy(lambda)</code>	Orders the entities according to the logic of the lambda expression.
<code>FirstOrDefault(lambda)</code>	Returns the first instance of the entity identified by the lambda expression parameter, or null if nothing is found.
<code>ToList()</code>	Returns a List<T> object with one or more entities.

## A method of the DbSet<Entity> class that gets an entity by its id

Method	Description
<code>Find(id)</code>	Returns the first instance of the entity identified by the id value for its primary key, or null if nothing is found.

## A using directive that imports the LINQ namespace

```
using System.Linq;
```

## Code that builds a query expression

```
IQueryable<Movie> query = context.Movies.OrderBy(m => m.Name);
```

## Code that executes a query expression

```
List<Movie> movies = query.ToList();
```

## Code that builds and executes a query expression

```
var movies = context.Movies.OrderBy(m => m.Name).ToList();
```

## Code that builds a query expression by chaining LINQ methods

```
var query = context.Movies.Where(m => m.Rating > 3).OrderBy(m => m.Name);
```

## Code that builds a query expression on multiple lines

```
IQueryable<Movie> query = context.Movies;
query = query.Where(m => m.Year > 1970);
query = query.Where(m => m.Rating > 3);
query = query.OrderBy(m => m.Name);
```

## Three ways to select a movie by its id

```
int id = 1;
var movie = context.Movies.Where(m => m.MovieId == id).FirstOrDefault();
var movie = context.Movies.FirstOrDefault(m => m.MovieId == id);
var movie = context.Movies.Find(id);
```

## Description

- *Language-Integrated Query (LINQ)* is a .NET component that allows you to query data in code. You can use LINQ to query the data in a DbSet property.

---

Figure 4-8 How to select data

The last example shows three C# statements that translate to the same SQL statement. In other words, each of these statements is functionally the same as the others. Here, using `Find()` leads to the shortest and cleanest code. However, if you aren't searching on a primary key, you will need to use one of the other techniques.

When you combine the building and executing of a query expression, make sure the method that executes the query comes last. That way, the filtering and sorting can be performed by the database server. Otherwise, the database server returns an unfiltered and unsorted collection to the web app, and the web app performs the filtering and sorting. In general, it's better to have the database server perform these filter and sort operations, especially if you're dealing with a large number of rows.

## How to insert, update, and delete data

---

The first table in figure 4-9 presents three methods of the `DbSet` class that mark the specified entity for insertion, update, or deletion in the database. The `Add()` method also adds the specified entity to the `DbSet` collection. However, these methods don't execute code against the database. Instead, they mark the entities that require database action.

The second table presents the `SaveChanges()` method of the `DbContext` class. This method executes the operations at the database. For example, if the `Update()` method has marked a row for update, the `SaveChanges()` method executes that update operation at the database.

The first code example shows how to import the EF Core namespace. Then, the second example shows how to use EF Core to add a new movie to the database. Here, the first statement creates a new `Movie` entity. Then, the second statement passes the `Movie` entity to the `Add()` method. This adds the movie to the `DbSet` property named `Movies` and marks it as `Added`. Next, the third statement calls the `SaveChanges()` method to execute this insertion at the database. The code for editing and deleting movies looks similar as shown a little later in figure 4-11.

## How to view the generated SQL statements

---

When you use EF Core, it generates SQL statements that it executes at the database. Sometimes it's helpful to view these SQL statements. To do that, you can add a logging setting to the `appsettings.json` file like the one shown in the third example of figure 4-9. Then, the generated SQL statements are displayed in the Output window of Visual Studio.

The fourth example in this figure shows a code statement that builds and executes a simple select query. Then, the fifth example shows the SQL statement this code executes at the database.

When you no longer want to see the SQL that EF Core generates, you can delete this setting. Or, to make it easy to enable EF Core logging again in the future, you can change its value to "Warning".

### Three methods of the DbSet class

Method	Description
<code>Add(entity)</code>	Adds an entity to the DbSet collection and marks it as Added.
<code>Update(entity)</code>	Marks the entity as Modified.
<code>Remove(entity)</code>	Marks the entity as Deleted.

### One method of the DbContext class

Method	Description
<code>SaveChanges()</code>	Saves changes to the database.

### A using directive that imports the EF Core namespace

```
using Microsoft.EntityFrameworkCore;
```

### Code that adds a new movie to the database

```
var movie = new Movie { Name = "Taxi Driver", Year = 1976, Rating = 4 };
context.Movies.Add(movie);
context.SaveChanges();
```

### An appsettings.json file that displays the generated SQL statements

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft": "Warning",
      "Microsoft.Hosting.Lifetime": "Information",
      "Microsoft.EntityFrameworkCore.Database.Command": "Debug"
    }
  ...
}
```

### Code that selects movies from the database

```
var movies = context.Movies.OrderBy(m => m.Name).ToList();
```

### The generated SQL statement

```
SELECT [m].[MovieId], [m].[Name], [m].[Rating], [m].[Year]
FROM [Movies] AS [m]
ORDER BY [m].[Name]
```

### Description

- You use the methods of the DbSet and DbContext classes to add, update, or delete entities.
- You can add a logging setting to the appsettings.json file to see the SQL that EF Core executes at the database. This SQL is displayed in the Output window of Visual Studio.

---

Figure 4-9 How to insert, update, and delete data and view the generated SQL

## The Movie List app

---

So far, you have learned how to code an entity class and a DbContext class. Then, you learned how to use them to create a database for storing data about movies. Next, you learned how to select, add, update, and delete data from that database.

Now, you'll learn how to code the controllers and views used by the Movie List app presented at the beginning of this chapter. However, the code in the next few topics displays slightly simpler pages than the ones presented in figure 4-1. That's because this Movie List app won't be complete until the end of the chapter.

### The Home controller

---

Figure 4-10 starts by presenting the code for the Home controller of the Movie List app. This controller starts with a private property named context of the MovieContext type. Then, the constructor accepts a MovieContext object and assigns it to the context property. As a result, the other methods in this class can easily access the MovieContext object.

This constructor works because of the dependency injection code in the Startup.cs file presented in figure 4-6. That's because the constructor specifies that it needs an instance of the MovieContext class. As a result, the MVC framework creates one based on the options specified in the Startup.cs file and passes it to that constructor.

The Index() action method of the Home controller uses the context property to get a collection of Movie objects from the database. But first, it sorts those objects alphabetically by movie name. Then, it passes that collection to the view.

### The Home/Index view

---

This figure also presents the code for the Home/Index view of the Movie List app. This view begins by using the @model directive to specify that the model for this view is a collection of Movie objects because that's what the Index() action method of the Home controller passes to this view.

Most of the HTML in this view displays a table of movie data. Just above the table, a link requests the Add() action method of the Movie controller. This link allows the user to add a new movie.

Within the body of the table, an inline foreach statement loops through the collection of Movie objects and displays each one in a row. Within that loop, the fourth column adds links that request the Edit() and Delete() action methods of the Movie controller for that specific movie. These links also use the asp-route-id tag helper to pass the ID for each movie. You'll learn more about how this tag helper works in chapter 7. For now, all you need to know is that it appends the MovieId value for the selected movie to the end of the URL.

## The Home controller

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

namespace MovieList.Controllers
{
    public class HomeController : Controller
    {
        private MovieContext context { get; set; }

        public HomeController(MovieContext ctx) {
            context = ctx;
        }

        public IActionResult Index() {
            var movies = context.Movies.OrderBy(m => m.Name).ToList();
            return View(movies);
        }
    }
}
```

## The Home/Index view

```
@model List<Movie>
 @{
     ViewBag.Title = "My Movies";
 }

<h2>Movie List</h2>

<a asp-controller="Movie" asp-action="Add">Add New Movie</a>
<table class="table table-bordered table-striped">
    <thead>
        <tr>
            <th>Name</th>
            <th>Year</th>
            <th>Rating</th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var movie in Model) {
            <tr>
                <td>@movie.Name</td>
                <td>@movie.Year</td>
                <td>@movie.Rating</td>
                <td>
                    <a asp-controller="Movie" asp-action="Edit"
                        asp-route-id="@movie.MovieId">Edit</a>
                    <a asp-controller="Movie" asp-action="Delete"
                        asp-route-id="@movie.MovieId">Delete</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

---

Figure 4-10 The Home controller and the Home/Index view

## The Movie controller

---

Figure 4-11 presents the code for the Movie controller of the Movie List app. This controller starts by defining a private MovieContext property named context. Then, it uses its constructor to set that property. This works the same as it does for the Home controller described in the previous figure.

The Movie controller has three action methods: Add(), Edit(), and Delete(). In addition, the Edit() and Delete() actions have overloads that handle both GET and POST requests. The Add() action, by contrast, only handles GET requests. That's because the Add() and Edit() actions both use the Movie/Edit view.

For a GET request, both the Add() and Edit() actions set a ViewBag property named Action and pass a Movie object to the view. However, the Add() action passes an empty Movie object, and the Edit() action passes a Movie object with data for an existing movie. To do that, the Edit() action passes its id parameter to the Find() method to retrieve a movie from the database. This is possible because the Edit link in the Home/Index view uses the `asp-route-id` tag helper to specify a value for the id argument.

To be able to use the same view as the Edit() action, the Add() action passes “Edit” as the first argument to the View() method. As a result, the Add() action displays the same view as the Edit() action.

Because the Edit() and Add() actions both use the Movie/Edit view, the Edit() action method for a POST request must be able to add new movies and update existing ones. To do that, it starts by checking whether the user entered valid data for the model.

If the user entered valid data, the code checks the value of the `MovieId` property of the Movie object it receives from the view. If the value is zero, it's a new movie. In that case, the code passes it to the Add() action of the `Movies` property. Otherwise, it's an existing movie. In that case, the code passes it to the Update() action of the `Movies` property.

After marking the Movie object for insertion or update, the Edit() action calls the `SaveChanges()` method of the `MovieContext` property. This causes the movie data to be inserted or updated in the database. Then, the code redirects the user back to the `Index()` action of the Home controller, which displays the Home/Index view.

However, if the user didn't enter valid data for the model, the code resets the `Action` property of the `ViewBag` to “Add” or “Edit”. Then, it sends the data the user entered back to the view to be redisplayed.

The Delete() action for a GET request uses its `id` parameter to retrieve a Movie object for the specified movie from the database. Then, it passes that Movie object to the view. Again, this is possible because the Delete link in the Home/Index view specifies a value for the `id` argument.

The Delete() action for a POST request passes the Movie object it receives from the view to the `Remove()` method of the `Movies` property. After that, it calls the `SaveChanges()` method of the `MovieContext` property to delete the movie from the database. Finally, it redirects the user back to the `Index` action of the Home controller.

## The Movie controller

```
namespace MovieList.Controllers
{
    public class MovieController : Controller
    {
        private MovieContext context { get; set; }

        public MovieController(MovieContext ctx)
        {
            context = ctx;
        }

        [HttpGet]
        public IActionResult Add()
        {
            ViewBag.Action = "Add";
            return View("Edit", new Movie());
        }

        [HttpGet]
        public IActionResult Edit(int id)
        {
            ViewBag.Action = "Edit";
            var movie = context.Movies.Find(id);
            return View(movie);
        }

        [HttpPost]
        public IActionResult Edit(Movie movie)
        {
            if (ModelState.IsValid)
            {
                if (movie.MovieId == 0)
                    context.Movies.Add(movie);
                else
                    context.Movies.Update(movie);
                context.SaveChanges();
                return RedirectToAction("Index", "Home");
            } else {
                ViewBag.Action = (movie.MovieId == 0) ? "Add": "Edit";
                return View(movie);
            }
        }

        [HttpGet]
        public IActionResult Delete(int id)
        {
            var movie = context.Movies.Find(id);
            return View(movie);
        }

        [HttpPost]
        public IActionResult Delete(Movie movie)
        {
            context.Movies.Remove(movie);
            context.SaveChanges();
            return RedirectToAction("Index", "Home");
        }
    }
}
```

## Description

- The Add() and Edit() action methods both display the Movie/Edit view.

---

Figure 4-11 The Movie controller

## The Movie/Edit view

---

Figure 4-12 presents the code for the Movie/Edit view of the Movie List app. This view begins by using the @model directive to specify that the model for this view is a Movie object because that's what the Add() and Edit() actions for a GET request pass to this view.

In the Razor code block, a string variable named title is used to store the value of the ViewBag.Action property followed by a space and "Movie". This creates a title variable of "Add Movie" for the Add() action and a title of "Edit Movie" for the Edit() action. Then, this code uses the title variable to set the ViewBag.Title property that's used by the layout for the view and as the text of the <h2> element.

The Edit view contains a form that posts to the Edit() action for a POST request. This form uses an asp-action tag helper but not an asp-controller one. As a result, the form posts to the Edit() action of the current controller, which is the Movie controller.

This form begins with a <div> element that displays any data validation messages. Then, it displays labels and text boxes for the movie name, year, and rating. In addition, it uses a hidden field to store the movie ID.

If the Add() action method served this view, the textboxes are blank and the value of the hidden field is zero. That's because that action method passes an empty Movie object to this view. However, if the Edit() action method served this view, these fields contain data for the selected movie. Either way, the values in these fields are posted to the Edit() action that handles POST requests. By the way, this code binds the MovieId value to a hidden field rather than a textbox because it's a primary key value, and you don't usually want to allow users to change a primary key.

The bottom of the form displays a button and a link that's formatted as a button. The first button is a submit button. As a result, it posts the form to the Edit() action method when the user clicks it. The text for this button is the value of the ViewBag.Action property. As a result, this button displays text of "Add" or "Edit", depending on which action method served the view.

The link that's formatted as a button allows the user to cancel the operation. It uses helper tags to request the Index() action of the Home controller. This displays the Home/Index view.

## The Movie/Edit view

```
@model Movie
{
    string title = ViewBag.Action + " Movie";
    ViewBag.Title = title;
}

<h2>@ViewBag.Title</h2>

<form asp-action="Edit" method="post">
    <div asp-validation-summary="All" class="text-danger"></div>

    <div class="form-group">
        <label asp-for="Name">Name</label>
        <input asp-for="Name" class="form-control">
    </div>

    <div class="form-group">
        <label asp-for="Year">Year</label>
        <input asp-for="Year" class="form-control">
    </div>

    <div class="form-group">
        <label asp-for="Rating">Rating</label>
        <input asp-for="Rating" class="form-control">
    </div>

    <input type="hidden" asp-for="MovieId" />

    <button type="submit" class="btn btn-primary">@ViewBag.Action</button>
    <a asp-controller="Home" asp-action="Index"
        class="btn btn-primary">Cancel</a>
</form>
```

### Description

- The Edit view uses a Movie object as its model and binds HTML elements to the properties of the Movie object.
- The Edit view uses the ViewBag.Action property to set the text of the <title>, <h2>, and <button> elements. This makes it possible for this view to work for both the Add() and Edit() actions.
- The MovieId value is bound to a hidden field. As a result, users can't change this value.
- When an empty Movie object is passed to the view by the Add() action method, the text boxes are blank. When a Movie object with data is passed to the view by the Edit() action method, the data in that object is displayed in the text boxes.
- The form uses tag helpers to specify the action but not the controller. As a result, the submit button posts the form to the Edit() action of the current controller, which is the Movie controller.
- The Cancel link uses tag helpers to request the Index() action of the Home controller. This displays the Home/Index view.

---

Figure 4-12 The Movie/Edit view

## The Movie/Delete view

---

Figure 4-13 presents the code for the Movie/Delete view. This view begins with code that uses the @model directive to specify that the model for this view is a Movie object, because that's what the Delete() action for a GET request passes to this view.

The Delete view contains a form that posts to the Delete() action method for a POST request. Like the form in the Edit view, this form uses tag helpers to specify an `asp-action` tag helper but not an `asp-controller`. As a result, the form posts to the Delete() action of the current controller, which is the Movie controller.

Unlike the Edit form, this form doesn't contain any labels or text boxes. That's because it doesn't ask the user to input any data. Instead, it asks the user to confirm the deletion by clicking a button.

This form begins by binding the `MovieId` value to a hidden field. That way, the form posts the primary key of the movie to be deleted to the Delete() action.

The Delete() action method for a POST request expects to receive a Movie object from this view. However, if you look at the code presented here, you can see that this view only posts the `MovieId` value to that action method. This means the Movie object that the Delete() action method receives has null values for all the properties except `MovieId`. That's OK, though, because the `Remove()` method of the `Movies` property only needs the `MovieId` value to mark the selected entity for deletion.

Like the Edit form, the bottom of this form displays a button and a link that's formatted as a button. The first button is a submit button. As a result, it posts the form to the Delete() action when the user clicks it.

Like the Edit form, the link that's formatted as a button allows the user to cancel the operation. To do that, it uses tag helpers to request the Index() action of the Home controller. This displays the Home/Index view.

## The Movie/Delete view

```
@model Movie
{
    ViewBag.Title = "Delete Movie";
}

<h2>Confirm Deletion</h2>
<h3>@Model.Name (@Model.Year)</h3>

<form asp-action="Delete" method="post">
    <input type="hidden" asp-for="MovieId" />

    <button type="submit" class="btn btn-primary">Delete</button>
    <a asp-controller="Home" asp-action="Index"
        class="btn btn-primary">Cancel</a>
</form>
```

### Description

- The Delete view uses a Movie object as its model and displays properties of that Movie object to the user to confirm deletion.
- The MovieId value is bound to a hidden field. That way, it is passed to the Delete() action method when the user clicks the submit button and the form is posted.
- The form uses tag helpers to specify the action but not the controller. As a result, the submit button posts the form to the Delete() action of the current controller, which is the Movie controller.
- The Cancel link uses tag helpers to request the Index() action of the Home controller. This displays the Home/Index view.

---

Figure 4-13 The Movie/Delete view

## How to work with related data

So far, you've learned how to create a data-driven web app that uses a Movie entity class to generate a Movie table in a database. Now, you'll learn how to add another entity class and generate another database table. In addition, you'll learn how to relate this new entity to the existing one. As you learn how to do this, you should know that working with related data like this is a big topic and that you'll learn more about it in chapter 12.

### How to relate one entity to another

You can relate one entity to another by making one a property of another. To do this, you code a property with a data type of the entity you want to relate. Figure 4-14 shows how this works.

The first example presents a Genre class that has two properties: GenreId and Name. Here, the GenreId property is the primary key of this class. Because this property uses the string type instead of the int type, the database doesn't generate values for it.

The second example shows the Movie class used earlier in this chapter after it has been updated to contain a property named Genre of the Genre type. This is a simple way to relate the Genre class to the Movie class. And in cases where you only need to read and display related data, this technique is adequate.

However, you often need to do more than just display related data. For example, you might want to seed related data, update related data, validate related data, or include related data in a query. In cases like these, your entity classes are easier to work with if you also add a *foreign key property* that refers to the primary key property in the related entity.

The third example shows how this works. Here, a GenreId property is added to the Movie class just before the Genre property. This GenreId property has the same data type as the primary key property in the Genre entity and specifically links the two entities on that value.

## The Genre class

```
public class Genre
{
    public string GenreId { get; set; }
    public string Name { get; set; }
}
```

## How to add a Genre property to the Movie class

```
public class Movie
{
    /* MovieId, Name, Year, and Rating properties same as before */

    [Required(ErrorMessage = "Please enter a genre.")]
    public Genre Genre { get; set; }
}
```

## How to specify a foreign key property when adding a Genre property

```
public class Movie
{
    /* MovieId, Name, Year, and Rating properties same as before */

    [Required(ErrorMessage = "Please enter a genre.")]
    public string GenreId { get; set; }
    public Genre Genre { get; set; }
}
```

## Description

- You can relate one entity to another by coding a property with that entity class as its data type.
- A *foreign key property* indicates the property that's the primary key in the related class.
- Using a foreign key property makes it easier to seed, update, validate, or query related data. As a result, it's considered a best practice to use a foreign key property when you need to perform these operations.
- You'll learn more about primary and foreign keys in chapter 12.

---

Figure 4-14 How to relate one entity to another

## How to update the **DbContext** class and the seed data

---

When developing an app, it's common to need to update the database with new entities and seed initial data. Luckily, EF Core and migrations make this easy to do. For example, the Movie List app in this chapter started with just a Movie entity and some initial movie data. But now, a Genre entity has been added to the app. As a result, the MovieContext class needs to be updated to add this new entity with some initial data.

Figure 4-15 shows the MovieContext class after it has been updated to add the Genre entity. To start, this class adds a second DbSet property named Genres. EF Core can use this property to create a Genres table in the database. In addition, EF Core can use it with LINQ queries to select genre data from the database.

Next, the MovieContext class uses the HasData() method to seed initial genre data in the new table. This works similarly to the way that movie data is seeded, so you shouldn't have much trouble understanding how it works.

Finally, the MovieContext class updates the HasData() method for the Movie table. It does this by adding a GenreId field to each movie object, with the ID value of the appropriate genre. This is possible because the Movie entity has a foreign key property.

## The MovieContext class updated to add the Genre model with initial data

```
public class MovieContext : DbContext {
    public MovieContext(DbContextOptions<MovieContext> options)
        : base(options)
    { }

    public DbSet<Movie> Movies { get; set; }
    public DbSet<Genre> Genres { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder) {
        base.OnModelCreating(modelBuilder);

        modelBuilder.Entity<Genre>().HasData(
            new Genre { GenreId = "A", Name = "Action" },
            new Genre { GenreId = "C", Name = "Comedy" },
            new Genre { GenreId = "D", Name = "Drama" },
            new Genre { GenreId = "H", Name = "Horror" },
            new Genre { GenreId = "M", Name = "Musical" },
            new Genre { GenreId = "R", Name = "RomCom" },
            new Genre { GenreId = "S", Name = "SciFi" }
        );

        modelBuilder.Entity<Movie>().HasData(
            new Movie { MovieId = 1, Name = "Casablanca", Year = 1942,
                Rating = 5, GenreId = "D"
            },
            new Movie { MovieId = 2, Name = "Wonder Woman", Year = 2017,
                Rating = 3, GenreId = "A"
            },
            new Movie { MovieId = 3, Name = "Moonstruck", Year = 1988,
                Rating = 4, GenreId = "R"
            }
        );
    }
}
```

### Description

- When you add new entities to your app, you also add them to your DbContext class as DbSet properties.
- You can also seed initial data for the new entities. And, if the new entities are related to existing ones, you can update the seed data for the existing entities.
- EF Core uses the updated DbContext class and seed data to change the database.

---

Figure 4-15 How to update the DbContext class and the seed data

## How to use migrations to update the database

---

After you update the MovieContext class, you need to update the database to add a table and initial data for the new Genre entity. To do that, you can use the procedure presented in figure 4-16.

This procedure is similar to the procedure that you used to create a table for the Movie entity as described earlier in this chapter. To start, you display the Package Manager Console window. Then, you enter the Add-Migration command followed by a name for the migration file.

In this figure, the Add-Migration command specifies a name of “Genre” since that’s the name of the entity being added. However, if you want, you can use a more descriptive name such as “AddGenreField”. The more descriptive the names, the more the Migrations folder becomes self-documenting. In other words, if you use descriptive names, reading the names in the Migrations folder gives you a sense of the history of the database.

This figure presents some of the code in the Up() method of the Genre migration file that’s produced by the Add-Migration command. This file has more work to do than the Up() method presented in figure 4-7. Like that method, this method creates a table based on an entity and seeds it with initial data for the Genre entity. But, it also adds a GenreId column to the Movies table, updates each movie with a value for GenreId, and makes the GenreId column in the Movies table a foreign key. For now, don’t worry if you don’t understand all the foreign key code shown here. You’ll learn more about it in chapter 12.

If the migration file looks good, you can finish the procedure by running the Update-Database command. This executes the Up() method of the most recently created migration file, which in this case is the Genre file. If you’re using SQL Server LocalDB, you can use the procedure described in figure 4-7 to view the database and see the changes you made.

## How to update the database with the new Genre model and seed data

1. Select Tools → NuGet Package Manager → Package Manager Console to open the Package Manager Console window.
2. Type “Add-Migration Genre” at the command prompt and press Enter.
3. Type “Update-Database” at the command prompt and press Enter.

## Some of the code in the Up() method of the Genre migration file

```
protected override void Up(MigrationBuilder migrationBuilder)
{
    migrationBuilder.AddColumn<string>(
        name: "GenreId",
        table: "Movies",
        nullable: false,
        defaultValue: "");

    migrationBuilder.CreateTable(
        name: "Genres",
        columns: table => new {
            GenreId = table.Column<string>(nullable: false),
            Name = table.Column<string>(nullable: true)
        }, constraints: table => {
            table.PrimaryKey("PK_Genres", x => x.GenreId);
        });

    migrationBuilder.InsertData(
        table: "Genres",
        columns: new[] { "GenreId", "Name" },
        values: new object[,] {
            { "A", "Action" },
            { "C", "Comedy" },
            { "D", "Drama" },
            { "H", "Horror" },
            { "M", "Musical" },
            { "R", "RomCom" },
            { "S", "SciFi" }
        });
}

migrationBuilder.UpdateData(
    table: "Movies",
    keyColumn: "MovieId",
    keyValue: 1,
    column: "GenreId",
    value: "D");

// code that updates the other two movies

migrationBuilder.AddForeignKey(
    name: "FK_Movies_Genres_GenreId",
    table: "Movies",
    column: "GenreId",
    principalTable: "Genres",
    principalColumn: "GenreId",
    onDelete: ReferentialAction.Cascade);
}

...
```

Figure 4-16 How to use migrations to update the database

## How to select related data and display it on the Movie List page

---

At this point, you're ready to update the Movie List app so it displays this new Genre data. That's why figure 4-17 shows how to add each movie's genre to the table on the Movie List page. In other words, this figure shows how to make the Movie List page look like the one presented in figure 4-1.

The first example in this figure shows how to update the controller for the Movie List page. To start, this controller adds a using directive for the EF Core namespace. Then, the Index() action uses the Include() method of the EF Core namespace to select the genre data related to each movie.

The Include() method accepts a lambda expression that specifies the related entity. Whenever necessary, you can chain the Include() method as part of a longer LINQ query. Like the OrderBy() method, the Include() method doesn't execute at the database. Instead, it helps build up the query expression that the ToList() method eventually executes.

If you only need the GenreId value, not the data for the entire entity, you don't need to use the Include() method. That's because the Movie entity contains a foreign key property named GenreId. In other words, the GenreId value is automatically included when you select a Movie object. However, in this case, you want to get all the Genre data, not just the GenreId value. As a result, you need to use Include().

Once you've selected the related data, you use regular C# dot notation to work with it. In the second example, for instance, the Home/Index view adds a Genre column to the movie table. Within the foreach loop, it uses dot notation to display the Name property of the Genre property of the Movie object.

## Another LINQ method that builds a query expression

Method	Description
<code>Include(lambda)</code>	Includes a related entity according to the logic of the lambda expression.

## The Index() action method of the Home controller

```
using Microsoft.EntityFrameworkCore;
...
public class HomeController : Controller {
...
    public IActionResult Index() {
        var movies = context.Movies.Include(m => m.Genre)
            .OrderBy(m => m.Name).ToList();
        return View(movies);
    }
}
```

## The <table> element of the Home/Index view

```
<table class="table table-bordered table-striped">
    <thead>
        <tr>
            <th>Name</th>
            <th>Year</th>
            <th>Genre</th>
            <th>Rating</th><th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var movie in Model) {
            <tr>
                <td>@movie.Name</td>
                <td>@movie.Year</td>
                <td>@movie.Genre.Name</td>
                <td>@movie.Rating</td>
                <td><!-- Edit/Delete links same as before --></td>
            </tr>
        }
    </tbody>
</table>
```

Figure 4-17 How to select related data and display it on the Movie List page

## How to display related data on the Add and Edit Movie pages

---

Once you’re done displaying the related data on the Movie List page, you need to modify the Movie/Edit view to add a Genre drop-down list as shown in figure 4-18. This makes the Add and Edit Movie pages look like the ones presented in figure 4-1.

For this to work properly, the Add() and Edit() actions of the Movie controller must get a collection of Genre objects sorted in alphabetical order by name. Then, they must assign that collection to the Genres property of the ViewBag object. In this figure, the first three examples all contain a line of code that does this. That way, the Add() and Edit() actions for GET requests provide the data that the Add and Edit movie pages need to be able to display a drop-down list of genres when they’re first displayed. However, the Edit() action for POST requests only needs to provide the genre data for the drop-down list of genres if the user enters invalid data. That’s because the Edit() action for a POST request redirects to the Home/Index action if the user enters valid data.

In the Edit() action for GET requests, the code selects a Movie object by primary key as before. However, for the Edit Movie page, the Movie object only needs the GenreId value, not the entire Genre entity. As a result, this code doesn’t use the Include() method.

The Movie/Edit view receives a Movie object as a model and a collection of Genre objects in the ViewBag. Remember, it receives these objects from both the Add() and Edit() action methods. Then, it uses them to create a Genre drop-down list. To do that, this code binds the <select> element to the GenreId property of the model object, which is a Movie object. Then, a foreach statement loops through the Genre objects in the ViewBag to create a drop-down item for each Genre.

Since the <select> element is bound to the GenreId property of the model, the Genre drop-down on the Edit Movie page selects the genre of the selected movie. For the Add Movie page, none of the items in the Genre down-down match the GenreId property. As a result, the drop-down displays the first option in the list that says “select a genre”.

## The Add() action method of the Movie controller

```
[HttpGet]
public IActionResult Add()
{
    ViewBag.Action = "Add";
    ViewBag.Genres = context.Genres.OrderBy(g => g.Name).ToList();
    return View("Edit", new Movie());
}
```

## The Edit() action method of the Movie controller for GET requests

```
[HttpGet]
public IActionResult Edit(int id) {
    ViewBag.Action = "Edit";
    ViewBag.Genres = context.Genres.OrderBy(g => g.Name).ToList();
    var movie = context.Movies.Find(id);
    return View(movie);
}
```

## The Edit() action method of the Movie controller for POST requests

```
[HttpPost]
public IActionResult Edit(Movie movie)
{
    if (ModelState.IsValid)
    {
        if (movie.MovieId == 0)
            context.Movies.Add(movie);
        else
            context.Movies.Update(movie);
        context.SaveChanges();
        return RedirectToAction("Index", "Home");
    }
    else
    {
        ViewBag.Action = (movie.MovieId == 0) ? "Add": "Edit";
        ViewBag.Genres = context.Genres.OrderBy(g => g.Name).ToList();
        return View(movie);
    }
}
```

## The form tag of the Movie/Edit view

```
<form asp-action="Edit" method="post">
    ...
    <div class="form-group">
        <label asp-for="GenreId">Genre</label>
        <select asp-for="GenreId" class="form-control">
            <option value="">select a genre</option>
            @foreach (Genre g in ViewBag.Genres)
            {
                <option value="@g.GenreId">@g.Name</option>
            }
        </select>
    </div>
    ...

```

---

Figure 4-18 How to display related data on the Add and Edit Movie pages

## How to make user-friendly URLs

---

If you've made it this far, you've learned how to create a data-driven web app that works with related data. That's a significant accomplishment! However, there are a few more things you can do to make the URLs that your app produces more friendly to your users.

### How to make URLs lowercase with a trailing slash

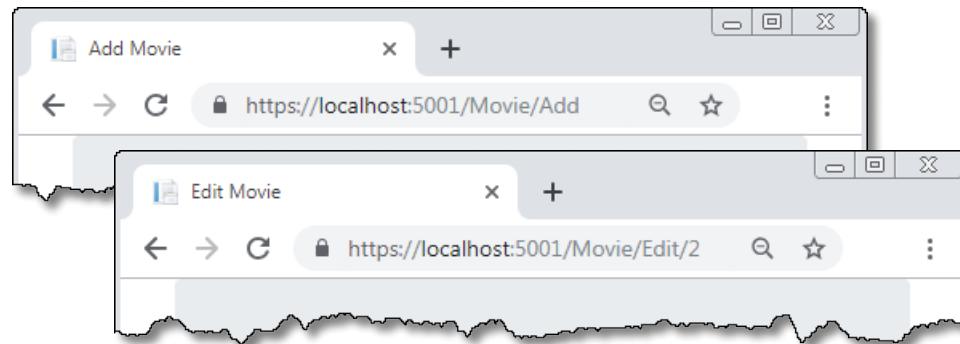
---

By default, MVC uses the names of the controllers and their action methods to create the URLs of the app. By convention, these names begin with an uppercase letter. This produces URLs that use some uppercase letters like those shown at the top of figure 4-19.

However, there's also a convention that URLs should be lowercase. This makes them easier for users to type. In addition, some developers like to include a trailing slash after a URL. This makes it easy for users to add text to the end of a URL.

Fortunately, it's easy to make your MVC app produce URLs that are lowercase and have a trailing slash. To do that, you just need to adjust the configuration in the Startup.cs file as shown in this figure. Once you've done that, your app will produce URLs that look like those shown at the bottom of this figure. This is a good example of how you can customize your app by configuring the middleware pipeline.

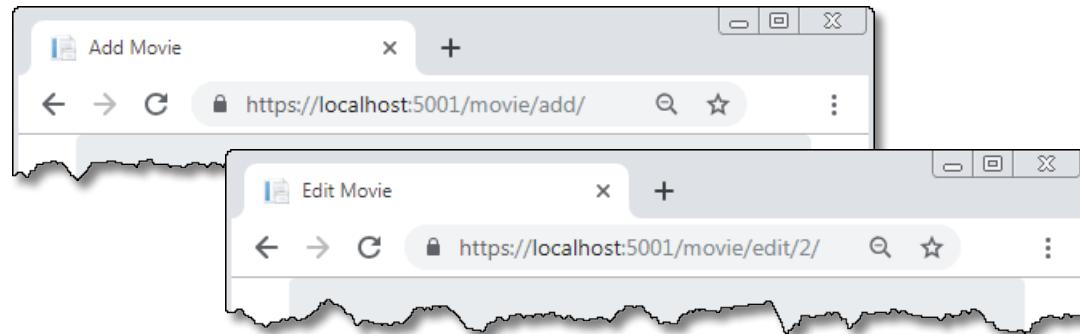
## The default URLs of an MVC app



**The ConfigureServices() method of the Startup.cs file updated to make URLs lowercase and end with a trailing slash**

```
public void ConfigureServices(IServiceCollection services) {
    ...
    services.AddRouting(options => {
        options.LowercaseUrls = true;
        options.AppendTrailingSlash = true;
    });
    ...
}
```

**The same MVC pages after changing the URL configuration**



### Description

- By default, MVC uses the names of the controllers and their action methods to create the URLs of the app. By convention, these names begin with an uppercase letter.
- It's generally considered a good practice to use lowercase letters for URLs.
- Some developers like to include a trailing slash after each URL to make it easy for users to type text at the end of a URL.
- You can modify the Startup.cs file to make URLs lowercase with a trailing slash.

---

Figure 4-19 How to make URLs lowercase with a trailing slash

## How to add a slug

---

In the Movie List app, when you click on the Edit or Delete link for a movie on the Product List page, the MovieId value of the selected movie is appended to the URL. For example, the URLs at the top of figure 4-20 include the ID values for movies.

These URLs work fine. In fact, many websites use URLs like these. However, some developers prefer to make their URLs more descriptive by adding a slug to them.

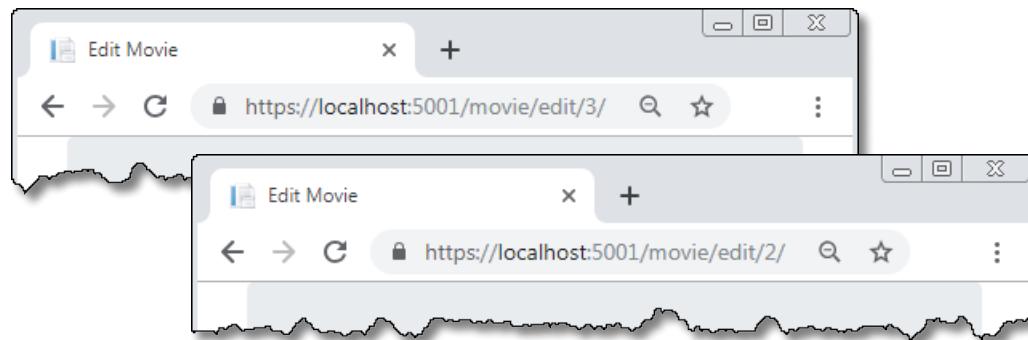
A *slug* is a descriptive section that comes at the end of a URL, often after URL sections that are used to look up data, such as a primary key value or a date. Adding a slug to a URL can make it easier for a user to predict the content of the page. This, in turn, can make a link more attractive for a user to click. This is particularly true when links are included in emails or text messages.

This figure presents one way to add slugs to the URLs in the Movie List app. First, add a second optional parameter named `slug` to the default route in the `Startup.cs` file. To do that, you can add a segment named `slug` followed by a question mark (?) to indicate that it's optional. For now, that's all you need to understand. You'll learn more about routing and the default route in chapter 6.

After you add an optional parameter named `slug` to the routing, you can add a read-only property named `Slug` to the `Movie` class. In this figure, this property just concatenates the `Name` and `Year` values into a single string, connected by a dash. In addition, it replaces any spaces in the `Name` value with dashes. In the real world, such a property would probably also perform other tasks like removing punctuation and returning an empty string for an empty `Movie` object.

After you add the `Slug` property, you can update the `Edit` and `Delete` links to add the `Slug` property of the `Movie` class as the optional `slug` URL parameter. Once you do that, your app should include the slug in its URLs as shown at the bottom of this figure.

## The Edit page with numeric ID values only in the URL



## The default route in the Startup.cs file updated to include a second optional parameter

```
app.UseEndpoints(endpoints =>
{
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}/{slug?}");
});
```

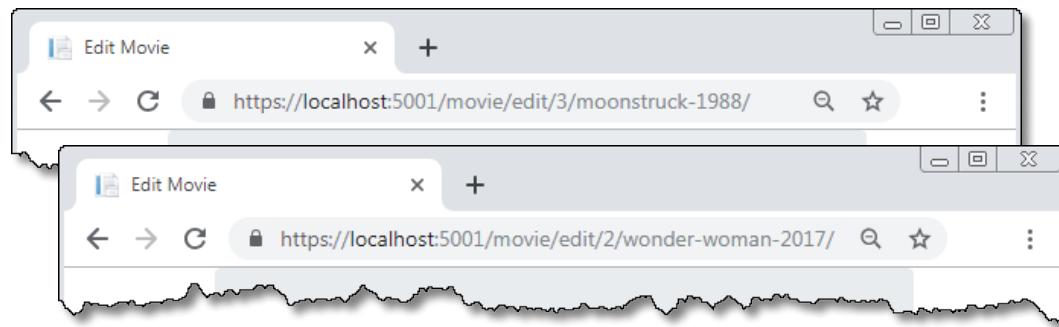
## A read-only property named Slug in the Movie class

```
public string Slug =>
    Name?.Replace(' ', '-').ToLower() + '-' + Year?.ToString();
```

## The Edit/Delete links in the Home/Index view

```
<a asp-controller="Movie" asp-action="Edit"
    asp-route-id="@movie.MovieId"
    asp-route-slug="@movie.Slug">Edit</a>
<a asp-controller="Movie" asp-action="Delete"
    asp-route-id="@movie.MovieId"
    asp-route-slug="@movie.Slug">Delete</a>
```

## The Edit page after updating the code to add a slug to the URL



## Description

- A *slug* is a descriptive section at the end of a URL. You can add a slug by adding an optional route parameter named slug to the Startup.cs file, adding a Slug property to the entity class, and including the Slug property on a link.

Figure 4-20 How to add a slug

## Perspective

---

The purpose of this chapter has been to get you started with learning how to develop multi-page, data-driven MVC web apps. Now, if this chapter has succeeded, you should be able to develop apps of your own that use multiple pages to work with a database. Yes, there's a lot more to learn, but you should be off to a good start.

## Terms

---

Entity Framework (EF) Core  
Object-relational mapping (ORM)  
EF Code First  
entity classes  
domain model classes  
primary key  
identity column  
seed data  
lookup data  
connection string  
dependency injection  
migration  
Language-Integrated Query (LINQ)  
query expression  
foreign key property  
slug

## Summary

---

- *Entity Framework (EF) Core* is an *object-relational mapping (ORM)* framework that allows you to work with the objects of a database in code.
- When you use the *EF Code First* approach, you code your model classes first and then create the database from those classes.
- Model classes that map to a database table are also known as *entity classes*, or *domain model classes*.
- When using EF Core, any property in an entity with a name of Id (or ID) or the entity name followed by Id (or ID) is a *primary key* that uniquely identifies the entity.
- When using EF Core, a primary key property of the int type specifies an *identity column* whose value is automatically generated.

- When using EF Core to create a database, you can include, or *seed*, some initial data in the database tables.
- *Lookup data* is data such as a list of states or movie genres that an app can use to look up related data.
- A *connection string* contains information that an app needs to connect to a database or other data source.
- *Dependency injection* is a design pattern in which the services an object needs are passed to it, or injected, rather than being hard coded as part of the object.
- A *migration* is a file that contains C# code for creating, modifying, or deleting database objects.
- *Language-Integrated Query (LINQ)* is a .NET component that allows you to query data in code. You can use LINQ to query the data in a DbSet property.
- Some LINQ methods build a *query expression*. Other LINQ methods execute those expressions at the database.
- A *foreign key property* indicates the primary key property in the related class.
- A *slug* is a descriptive section that comes at the end of a URL.

## Exercise 4-1 Create the Movie List app

This exercise guides you through the development of the Movie List app that's presented in this chapter. This will give you a chance to generate a database from entity classes.

### Set up the file structure

1. Create a new ASP.NET Core Web Application in the ex\_starts folder with a project name of MovieList and a solution name of Ch04Ex1MovieList. Base this app on the Web Application (Model-View-Controller) template.
2. Using figure 4-2 as a guide, remove all unnecessary files such as Models/ErrorViewModel.cs, Views/Home/Privacy.cshtml, and so on.
3. Using figure 4-3 as a guide, install the EF Core and EF Core Tools NuGet packages.
4. Using figure 3-2 as a guide, use LibMan to install version 4.3.1 of the Bootstrap CSS library. In the Solution Explorer, expand the wwwroot/lib folder and make a note of the path to the bootstrap.min.css file.

## Modify existing files

5. Modify the Views/Shared/\_Layout.cshtml file so it contains this code:

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
    <link rel="stylesheet" type="text/css"
        href="~/lib/bootstrap/css/bootstrap.min.css">
</head>
<body>
    <div class="container">
        <header class="jumbotron">
            <h1>My Movies</h1>
        </header>
        @RenderBody()
    </div>
</body>
</html>
```

Make sure the href attribute specifies the correct path to the bootstrap.min.css file!

6. Remove all action methods from the HomeController except for the Index() method.

## Code the classes for the model and the DB context

7. Add a Movie class to the Models folder and edit it so it contains the code shown in figure 4-4. Don't forget to add the using directive for data annotations.
8. Add a MovieContext class to the Models folder and edit it so it contains the code shown in figure 4-4. Don't forget to add the using directive for the EF Core namespace.
9. Modify the MovieContext class to contain the code for the OnModelCreating() method shown in figure 4-5.
10. Add the connection string to appsettings.json as shown in figure 4-6. However, to avoid database conflicts, specify a name of MoviesExercise for the database by editing the Database parameter like this:

**Database=MoviesExercise**

Make sure to enter the entire connection string on one line and to add a comma to the end of the previous line.

11. Modify the Startup.cs file so it includes the code shown in figure 4-6. This enables dependency injection for DbContext objects. At the top of the file, make sure to include all of the necessary using directives, including the using directive for the Models namespace.
12. Using figures 2-6 and 4-6 as a guide, remove any statements from the Startup.cs file that aren't needed by the Movie List app.

## Create the Movies database

13. Using figure 4-7 as a guide, open the Package Manager Console. At the command prompt, enter the “Add-Migration Initial” command. This should add a Migrations folder and migration files to the Solution Explorer. If you get an error, troubleshoot the problem.
14. At the Package Manager Console command prompt, enter the “Update-Database” command. This should create the database. If you get an error, troubleshoot the problem.
15. View your database. To do that, display the SQL Server Object Explorer as described in figure 4-7. Then, expand the nodes until you can see the MoviesExercise database that you just created.
16. View the seed data. To do that, expand the MoviesExercise node and the Tables node. Then, right-click on the dbo.Movies table and select View Data. This should show the data that’s stored in the Movies table.

## Modify the Home controller and its view

17. Modify the Home controller so it contains the code shown in figure 4-10. At the top of the controller, make sure to include using directives for all necessary namespaces including the Models namespace.
18. Modify the Home/Index view so it contains the code shown in figure 4-10.
19. From the Start drop-down list, select the name of the app, not IISExpress. Then, run the app. It should display the list of movies in the default browser. However, clicking the Add, Edit, or Delete links should cause an error.

## Add the Movie controller and its views

20. Add a new controller named MovieController to the Controllers folder. Then, modify this controller so it contains the code shown in figure 4-11. Again, make sure to include the using directive for the Models namespace.
21. Add a folder named Movie under the Views folder.
22. Add a new view named Edit to the Views/Movie folder. Then, modify this view so it contains the code shown in figure 4-12.
23. Add a new view named Delete to the Views/Movie folder. Then, modify this view so it contains the code shown in figure 4-13.
24. Run the app. It should display the list of movies. In addition, you should be able to add, edit, and delete movies.

## Update the database to store genre data

25. Add a Genre class to the Models folder. Then, modify this class so it contains the code shown in figure 4-14.
26. Modify the code for the Movie class so it includes a Genre property and foreign key as shown in figure 4-14.
27. Modify the MovieContext class to add the Genre model and seed it with initial data as shown in figure 4-15.

28. Open the Package Manager Console and enter the “Add-Migration Genre” command. This should add a migration file to the Solution Explorer.
29. At the Package Manager Console command prompt, enter the “Update-Database” command. This should add a Genre table and data to the database.

### **Update the controllers and views to work with genre data**

30. Modify the Home controller’s Index() method so it contains the code shown in figure 4-17.
31. Modify the Home/Index view so it contains the code shown in figure 4-17.
32. Modify the Movie controller’s Add() and Edit() action methods so they contain the code shown in figure 4-18.
33. Modify the Movie/Edit view so it contains the code shown in figure 4-18.
34. Run the app. It should work with genre data.

### **Make the URLs more user friendly**

35. Edit the Startup.cs file as shown in figure 4-19 to make the URLs for the app lowercase and with a trailing slash.
36. Add a slug to the URLs for editing or deleting a movie as described in figure 4-20.
37. Run the app. Its URLs should now be lowercase and use slugs when editing or deleting a movie.

# 5

## How to manually test and debug an ASP.NET Core web app

Testing and debugging are often the most difficult and time-consuming phase of web development. Fortunately, Visual Studio includes an integrated debugger that can help you locate and correct even the most obscure bugs. In this chapter, you'll learn how to use this debugging tool. In addition, you'll learn how to use your browser's developer tools to analyze the HTML and CSS for a page.

The tools presented in this chapter provide a way for you to manually test an app by running it, clicking on links, entering data, and so on. This approach is adequate when you're learning how to develop a simple web app. However, as you develop more complex web apps, you'll want to automate the testing for your apps as described in chapter 14.

<b>How to test an ASP.NET Core web app .....</b>	<b>176</b>
How to run a web app .....	176
How to use the browser's developer tools.....	178
How to use the Internal Server Error page.....	180
How to use the Exception Helper .....	182
<b>How to use the debugger .....</b>	<b>184</b>
How to use breakpoints .....	184
How to work in break mode .....	186
How to monitor variables and expressions.....	188
How to use tracepoints .....	190
<b>Perspective .....</b>	<b>192</b>

## How to test an ASP.NET Core web app

When you *test* a web app, you try to make it fail. In other words, the goal of testing is to find all of the errors. When you *debug* an app, you determine the cause of any errors that you've found and fix them.

To test an ASP.NET Core web app, you typically start by running it from Visual Studio in the default browser. Then, you test the app with other web browsers to make sure it works right in all of them.

### How to run a web app

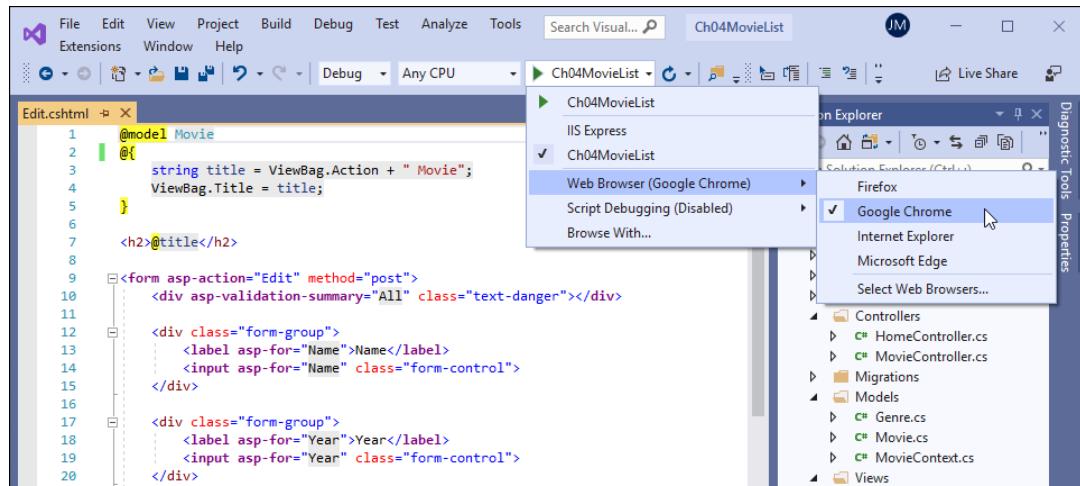
Figure 5-1 begins by showing how to change the default browser. Then, it presents several ways you can run a web app in the default browser. The first two techniques run the app without the debugger. However, the next three techniques show how to run the app with the debugger.

So, when should you run the app with the debugger and when should you run it without the debugger? This is mostly a matter of personal preference. If you don't think you'll need to use the debugger, you can run the app without debugging. Since Visual Studio doesn't need to attach the debugger, this starts the app slightly faster than when you use the debugger. Then, if an error occurs, it displays the Internal Server Error page shown in figure 5-3. In most cases, you can use this page to determine the cause of the error and fix it.

However, if you need to take a closer look at an error, you can run the app with debugging. This allows you to use the debugger to find the cause of an error as described later in this chapter. Since it's common to need to fix errors when you're developing an app, some developers almost always run the app with debugging. This may cause the app to start slightly more slowly, but if the app encounters an error, the debugger can help you find and fix the error more quickly.

Once you've thoroughly tested a web app with your default browser, you typically want to test it for *browser incompatibilities*. To do that, you need to run your app in all of the browsers it supports to make sure it is formatted correctly in all of them. One way to do that is to use the last technique presented in this figure to run the app in multiple browsers at the same time. This runs the app without the debugger. In most cases, that's fine since you don't usually need the debugger to resolve browser incompatibilities. Instead, you typically use the browser's developer tools as shown in the next figure.

## The Start drop-down menu and the Web Browser menu



### How to change the default browser

- Display the Start drop-down menu by clicking on its down arrow. Then, display the Web Browser menu and select a browser from it.

### How to run an app without debugging

- Press Ctrl+F5.
- Select Debug→Start Without Debugging.

### How to run an app with debugging

- Press F5.
- Click the Start button in the Standard toolbar.
- Select Debug→Start Debugging.

### How to stop debugging

- Press Shift+F5.
- Click the Stop Debugging button in the Debug toolbar.
- Select Debug→Stop Debugging.

### How to run an app in multiple browsers

- To run an app in two or more browsers, select Browse With from the Start drop-down menu. Then, in the resulting dialog, hold down the Ctrl key, select the browsers you want to use, and click the Browse button.
- When you run an app in two or more browsers, the app is run without debugging.

Figure 5-1 How to run a web app

## How to use the browser's developer tools

When you test your app and find that the pages aren't formatted correctly in all browsers, you need to change the HTML or CSS so the formatting is correct. In some cases, though, it's hard to figure out what HTML or CSS needs to be changed. To help you with that, most browsers provide *developer tools* that let you view the HTML elements and CSS styles that are rendered by the web server. Figure 5-2 presents the basic skills for working with the developer tools in five popular browsers.

To start, you can open or close the developer tools in Chrome, Firefox, and Edge by pressing F12. Because of that, these tools are sometimes called *F12 tools*. If you're using Opera or Safari, though, you have to use the other techniques described in this figure to open and close the developer tools.

The screen in this figure shows the developer tools in Chrome. To display the HTML and CSS for a page, you use the Elements tab as shown here. This tab displays the HTML in a hierarchical structure. Then, you can expand and collapse elements using the arrowheads to the left of the elements. In this case, a <div> element that uses the Bootstrap container class has been expanded so you can see that it contains a <header> element. In addition, the <header> element has been expanded so you can see that it contains a <h1> element.

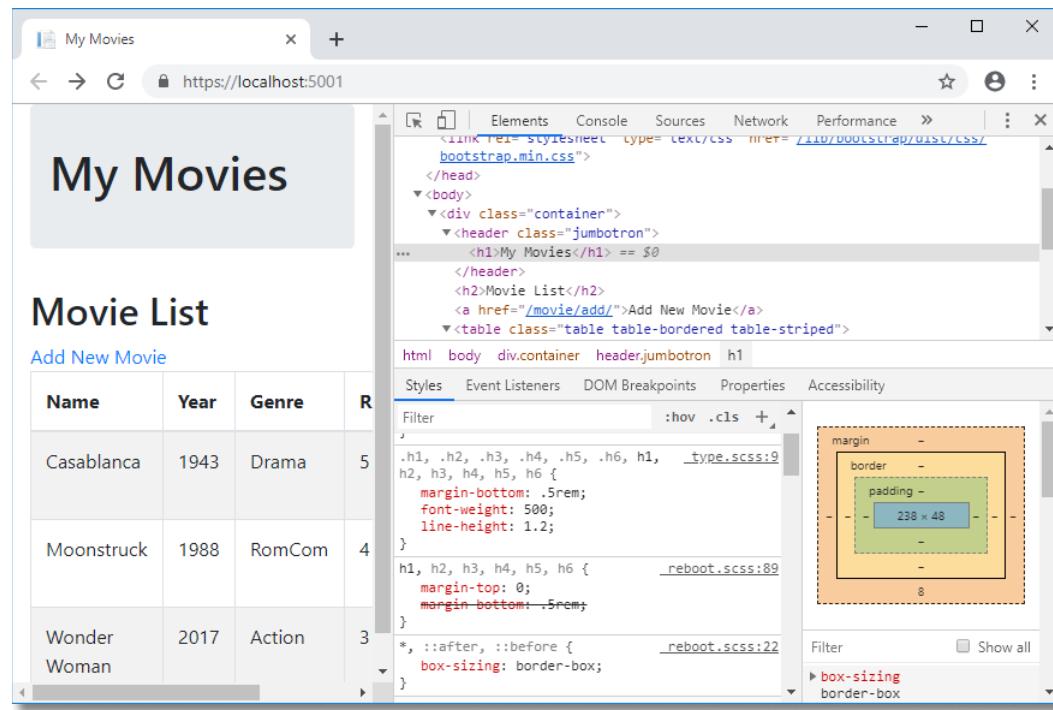
If you want to view the styles that have been applied to any element, you just select that element. Then, the styles are displayed in the Styles tab that's displayed below the Elements tab. In this figure, for example, the Styles tab shows the styles for the <h1> element. This shows the styles for each style rule, and it shows what style sheet contains the style rule. In this example, all of the style sheets are part of the Bootstrap library.

When a line appears through a style, it means that the style has been overridden by another style. Knowing this can be helpful when you're trying to figure out why your styles aren't working the way you expect.

One of the best uses for the developer tools is to determine the cause of formatting issues. Another is to view the HTML that's generated by ASP.NET Core and returned to the browser. Yet another is to view the JavaScript that's generated for the validation controls.

Although this figure shows the developer tools for Chrome, the developer tools for the other browsers work similarly. The best way to learn how to use these tools is to experiment with them. You can also learn about all of the functionality provided by these tools by searching for more information online.

## Chrome with the Elements tab of the developer tools open



## How to open and close the developer tools In Chrome, Firefox, and Edge

- To open, press F12. Or, right-click an element in the page and select Inspect.
- To close, press F12. Or, click the X in the upper right corner of the tools panel.

## How to open and close the developer tools in Opera and Safari

- To open, right-click an element in the page and select Inspect Element.
- To close, click the X in the upper right corner of the tools panel.
- In Safari, you must enable the developer tools before you can use them. To do that, select Preferences, click the Advanced tab, and select the “Show Develop menu” item.

## How to view the rendered HTML and CSS styles

- Open the appropriate panel by clicking on its tab. In Firefox, it's called the Inspector tab. In Chrome, Safari, and Opera, it's called the Elements tab.
- Expand the nodes to navigate to the element you want. Then, click that element.
- The HTML elements for a page are typically shown in the top of the panel, and the CSS styles for the selected element are typically shown below the HTML elements.

## Description

- The *developer tools* of the major browsers provide some excellent debugging features, like viewing the HTML elements rendered by the web server and viewing the styles applied to those HTML elements.

Figure 5-2 How to use the browser's developer tools

## How to use the Internal Server Error page

As you test an ASP.NET Core web app, you may encounter errors known as *exceptions* that prevent the app from executing. Often, you can write code that anticipates these exceptions, catches them, and handles them appropriately. If an exception isn't handled, however, the app can't continue.

If you are running an app without debugging, the app displays an error page. If you have configured your app to use developer exception pages as described in chapter 2, your app should display an Internal Server Error page like the one shown in figure 5-3. For example, this page is displayed if an exception occurs when the Movie List app attempts to add a new movie to the database.

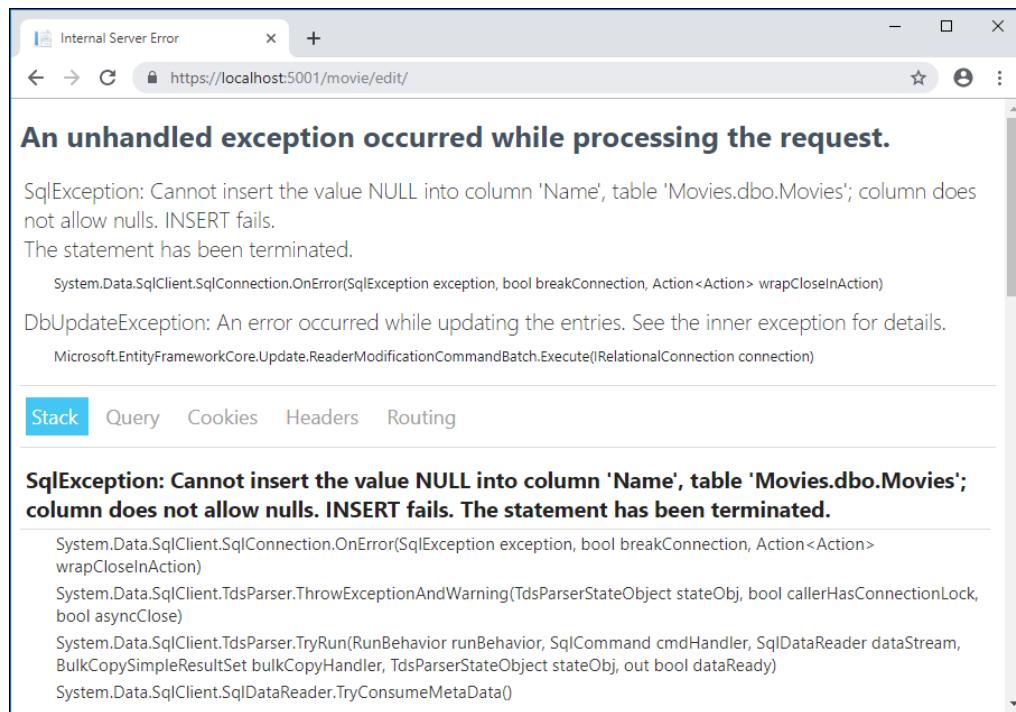
This Internal Server Error page isn't user friendly. In other words, you wouldn't want to display it to the end users of your app because it contains technical details about your app that aren't helpful to end users. However, the Internal Server Error page often provides all the information a developer needs to determine the cause of an exception. As a result, you typically want to use this page when you're developing an app. Later, when you deploy the app, you can replace it with a user-friendly error page that you've designed for the end users of your app.

The Internal Server Error page begins by displaying the name and description of the unhandled exception that occurred. In this figure, for example, a `SqlException` occurred because the app attempted to insert a `NULL` value into a column in the database that doesn't allow `NULL` values. This information alone is often enough to give you a good idea of what's causing the exception.

After displaying the name and description of the exception, the Internal Server Error page displays a row of links that allow you to learn more about the exception. By default, the `Stack` link is selected. This link displays a *stack trace*, which is a list of the methods that were active when the exception occurred. If you scroll down through this list of methods, you can find the line of code in your app that caused the exception. That's important because determining the cause of the exception is the first step to fixing your code to prevent or handle the exception.

In addition, the Internal Server Error page includes links that allow you to display the query strings, cookies, and routing data for the current request. These links can be helpful for determining the cause of some types of errors, and the information that they display should make more sense to you as you progress through this book and learn more about query strings, cookies, and routing.

## The Internal Server Error page



### The Internal Server Error page can display...

- The name and description of the exception.
- A stack trace that you can use to find the line of code that caused the exception.
- Query strings, cookies, headers, and routing data for the current request.

### Description

- An *exception* is an error that may occur when you run an app. If an exception occurs and isn't handled, the app can't continue.
- During development, if you run an app without debugging and an exception occurs, ASP.NET Core typically stops the app and sends an Internal Server Error page to the browser.
- By default, the Internal Server Error page selects the Stack link to display a *stack trace*, which is a list of the methods that were active when the exception occurred. However, you can also display information about the current request by clicking the Query, Cookies, Headers, and Routing links.

---

Figure 5-3 How to use the Internal Server Error page

## How to use the Exception Helper

---

If you are running an app with debugging and you encounter an exception like the one described in the previous figure, the app enters break mode and displays an Exception Helper like the one in figure 5-4. This non-modal dialog box indicates the type of exception that occurred and points to the statement that caused the error.

In many cases, you can use this information to determine what caused the error and what should be done to correct it. For example, the Exception Helper dialog box in this figure indicates that EF can't add a movie to the database if the Name column of the Movie table is NULL. In addition, the Exception Helper shows that this exception was caused by this line of code in the MovieController class:

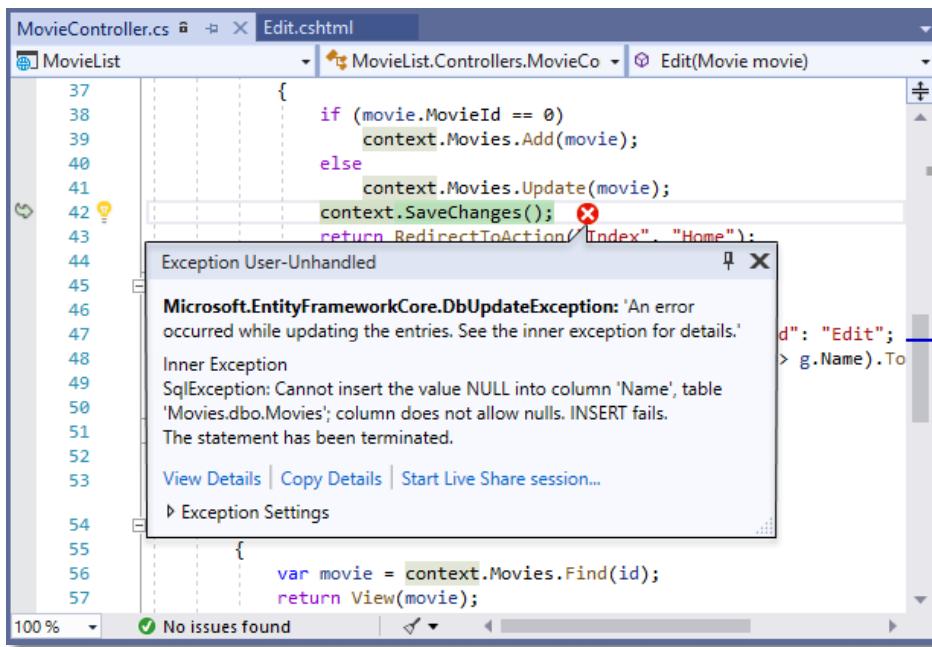
```
context.SaveChanges();
```

Based on that information, you can assume that your code must have set the name of the movie to NULL. This would be possible if the app didn't include the Required attribute before the Name property of the Movie class and the user didn't enter a name for a new movie. This would cause an exception because the database requires this data, but the app isn't supplying it. As a result, the call to the SaveChanges() method causes an exception to occur. To fix this issue, you can start by checking to make sure the Required attribute for the Name property is set properly.

When you are testing and debugging, you will encounter many exceptions that apply to general system operations such as arithmetic operations and the execution of methods. In addition, if your apps use databases, you will encounter EF and SQL exceptions.

In some cases, you won't be able to determine the cause of an exception just by analyzing the information displayed by the Exception Helper. Then, to get more information about the possible cause of the exception, you can use the links at the bottom of the Exception Helper to view the details of the exception, copy the details of the exception, or even start a Live Share session to collaborate with other developers. But first, you typically want to use the debugger to further analyze the problem as described in the next few figures.

## The Exception Helper



### Description

- If you run an app with debugging and an exception occurs, the debugger stops on the line of code that caused the exception and displays the Exception Helper.
- The Exception Helper provides the name and description of the exception and points to the statement that caused the exception. It also includes links to view the details, copy the details, or start a Live Share session.
- The Exception Helper often provides all the information you need to determine the cause of an exception.
- Since the Exception Helper is a non-modal dialog box, you can edit code while it is open.
- If you want to close the Exception Helper, you can click the X in its upper right corner. Then, you can open it again by clicking on the exception icon (the red circle with an X in it). This icon is displayed to the right of the statement that caused the exception.
- If you continue program execution after an exception occurs by pressing F5 or clicking on the Continue button, ASP.NET Core terminates the app and sends an error page to the browser as described in the previous figure.

Figure 5-4 How to use the Exception Helper

## How to use the debugger

---

The topics that follow introduce you to the basic techniques for using the Visual Studio *debugger* to debug an ASP.NET Core app. These techniques are almost identical to the techniques you use to debug any type of .NET app. As a result, if you have experience debugging other types of .NET apps, you should already be familiar with most of these techniques.

### How to use breakpoints

---

Figure 5-5 shows how to use *breakpoints* in an ASP.NET Core app. To start, you can set a breakpoint before you run an app or as an app is executing. However, a web app ends after it generates a page. So, if you switch from the browser to Visual Studio to set a breakpoint, the breakpoint won't be taken until the next time the page is executed. As a result, if you want a breakpoint to be taken the first time a page is executed, you'll need to set the breakpoint before you run the app.

After you set a breakpoint and run the app, the app enters *break mode* before it executes the statement that contains the breakpoint. In this figure, for example, the app will enter break mode before it executes the statement that caused the exception in the last figure to occur. Then, you can use the debugging features to debug the app.

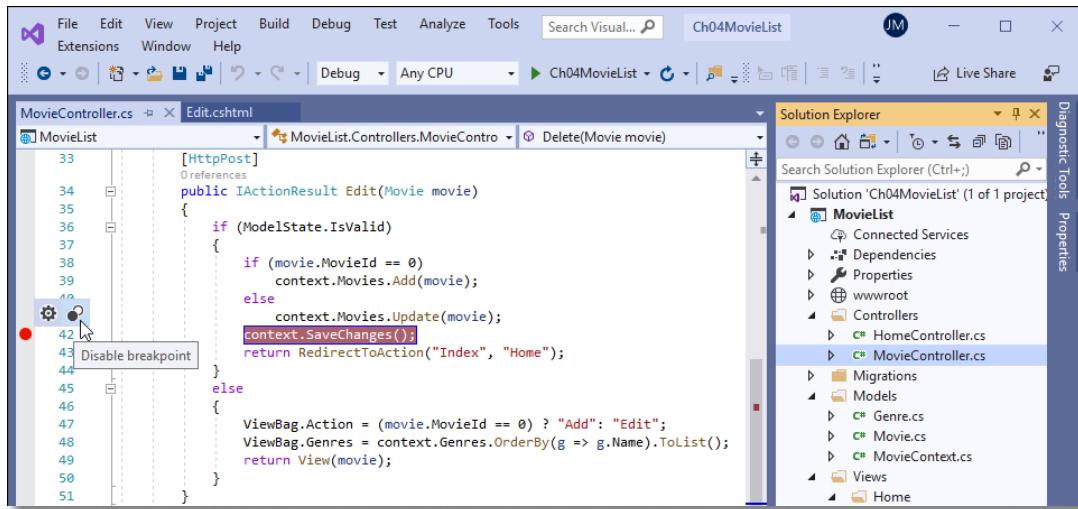
In some cases, you may want to set more than one breakpoint. You can do that either before you begin the execution of the app or while the app is in break mode. Then, when you run the app, it stops at the first breakpoint. And when you continue execution, the app executes up to the next breakpoint.

Once you set a breakpoint, it remains active until you remove it. In fact, it remains active even after you close the project. If you want to remove a breakpoint, you can use one of the techniques presented in this figure.

If you don't want to remove a breakpoint completely, but you don't want to stop on it, you can disable it using one of the techniques shown in this figure. Then, if you later want to stop on that breakpoint, you can enable it.

One easy way to enable and disable breakpoints is to use the Breakpoints window. This window lets you perform more advanced tasks like labeling groups of breakpoints, filtering breakpoints, and setting break conditions and hit counts.

## The Movie controller with a breakpoint



### How to set and remove breakpoints

- To set a breakpoint, click in the margin indicator bar to the left of the line number for a statement. This highlights the statement and adds a breakpoint indicator (a red dot) in the margin.
- To remove a breakpoint, click the breakpoint indicator.
- To remove all breakpoints, select Debug→Delete All Breakpoints.

### How to enable and disable breakpoints

- To enable or disable a breakpoint, point to the breakpoint indicator and select Enable/Disable Breakpoint from the resulting menu.
- To disable all breakpoints, select Debug→Disable All Breakpoints.
- To enable all breakpoints, select Debug→Enable All Breakpoints.
- To display the Breakpoints window, select Debug→Windows→Breakpoints. This window is most useful for enabling and disabling existing breakpoints.

### Description

- When Visual Studio encounters a *breakpoint*, it enters *break mode* before it executes the statement on which the breakpoint is set.
- You can set and remove breakpoints before you run an app or while you're in break mode.
- You can only set a breakpoint on a line that contains an executable statement.

Figure 5-5 How to use breakpoints

## How to work in break mode

Figure 5-6 shows the Movie controller of the Movie List app in break mode. In this mode, the next statement to be executed is highlighted. Then, you can use the debugging information that's available to try to determine the cause of an exception or a logical error.

One easy way to get information about what your code is doing is to use *data tips*. A data tip displays the current value of a variable or property when you hover the mouse pointer over it. You can also view the values of the members of an array, structure, or object by placing the mouse pointer over the arrowhead in a data tip.

In this figure, for example, the screen shows a data tip for a Movie object. This data tip displays its properties, including its Name property, whose value is null. You can view all this information just by pointing the mouse at variables and properties. When you move the mouse, the data tip disappears. However, you can keep the data tip open by clicking on the pin icon to the right of a data tip.

The debugging windows at the bottom of Visual Studio also display the values of variables and properties. In this figure, for example, the Locals window provides access to the same Movie object that's displayed in the data tip. In the next figure, you'll learn more about this window as well as some other debugging windows.

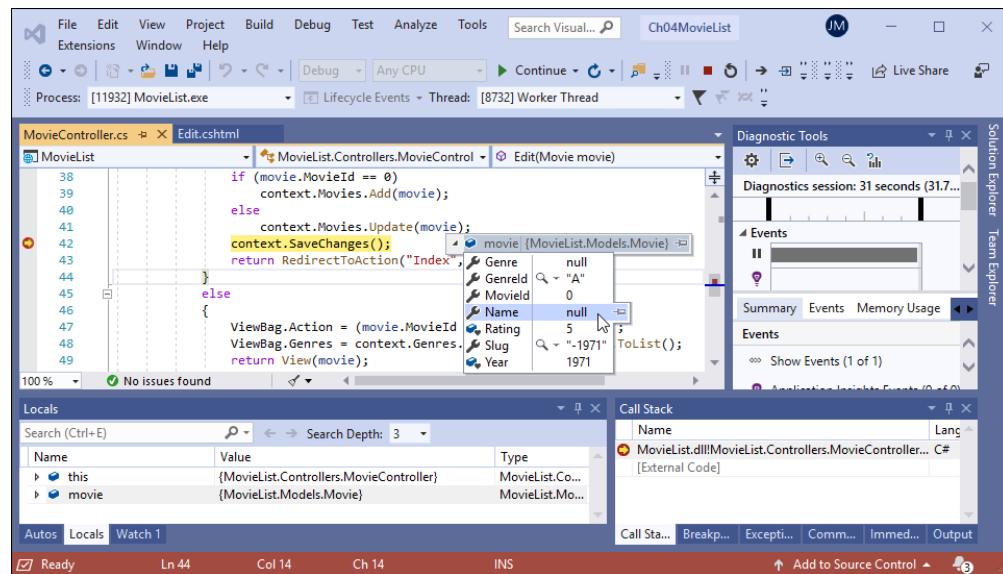
Once you're in break mode, you can use the commands summarized in this figure to control the execution of the app. These commands are available from the Debug menu as well as the Debug toolbar. In addition, you can use shortcut keys to start these commands.

To execute the statements of an app one at a time, you use the Step Into command. Each time you use this command, the app executes the next statement and returns to break mode. That way, you can check the values of properties and variables and perform other debugging functions as you step through code one statement at a time. The Step Over command is similar to the Step Into command, but it executes the statements in called methods without interruption (they are "stepped over").

The Step Out command executes the remaining statements in a method without interruption. When the method finishes, the app enters break mode before the next statement in the calling method is executed.

If your app gets caught in a loop that keeps executing indefinitely without generating a page, you can force it into break mode by choosing the Debug→Break All command. This command lets you enter break mode any time during the execution of an app.

## The Movie List app in break mode



## Commands in the Debug menu and toolbar

Command	Keyboard	Function
Start/Continue	F5	Start or continue execution of the app.
Break All	Ctrl+Alt+Break	Stop execution and enter break mode.
Stop Debugging	Shift+F5	Stop debugging and end execution of the app.
Restart	Ctrl+Shift+F5	Restart the entire app.
Step Into	F11	Execute one statement at a time.
Step Over	F10	Execute one statement at a time except for called methods.
Step Out	Shift+F11	Execute the remaining lines in the current method.

## Description

- When you enter break mode, the debugger highlights the next statement to be executed. Then, you can use the debugging windows and the buttons in the Debug menu and toolbar to control the execution of the program and determine the cause of an exception.
- To display the value of a variable or property in a *data tip*, position the mouse pointer over the variable or property in the Code Editor window. You can also use the pin icon to the right of a data tip to pin the data tip so it remains displayed.
- To display the members of an array, structure, or object in a data tip, position the mouse pointer over it to display its data tip, and then point to the arrow to the left of the data tip.
- You can use the Step Into, Step Over, and Step Out commands to execute one or more statements and return to break mode.
- To stop an app that's caught in a loop, use the Debug→Break All command.

Figure 5-6 How to work in break mode

## How to monitor variables and expressions

---

If you need to view the values of several app variables or expressions, you can do that using the Autos, Locals, or Watch windows. By default, these windows are displayed in the lower left corner of the IDE when an app enters break mode. If they're not displayed, you can display them by selecting the appropriate command from the Debug→Windows menu.

Figure 5-7 begins by displaying the Locals and Watch windows. The Locals window displays information about the variables within the scope of the current method. If the code in a controller is currently executing, this window also includes information about the controller and all of its properties such as its ViewBag property. The Autos window is similar to the Locals window, but it only displays information about the variables used in the current statement and the previous statement.

Unlike the Autos and Locals windows, a Watch window lets you choose the values that are displayed. For example, the Watch window in this figure displays the Name property of the movie object. You can also add properties of the page or of a business class to a Watch window as well as the values of expressions. In fact, an expression doesn't have to exist in the app for you to add it to a Watch window.

To add an item to a Watch window, you can type it directly into the Name column. Alternatively, if the item appears in the Code Editor window, you can highlight it in that window and then drag it to a Watch window. You can also highlight the item in the Code Editor or data tip, right-click it, and select the Add Watch command. This adds the item to the Watch window that's currently displayed. If necessary, you can display up to four Watch windows. However, you typically only need one Watch window.

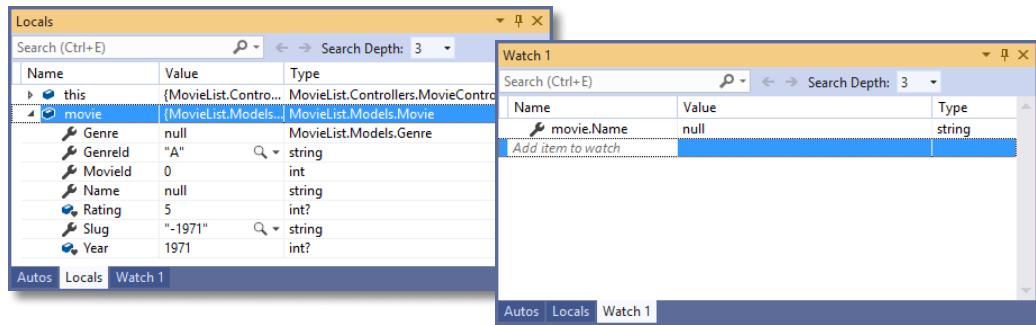
The Immediate window is useful for displaying or changing the values of variables or properties. To display a value, you type a question mark followed by the name of a variable or an expression. In this figure, for example, the first line is a command that displays the Name property of the movie object on the second line. This shows that the Name property is null.

The third line is a command that assigns a name of "Wizard of Oz" to the Name property. This immediately changes the value of the Name property in the currently executing app and is reflected by the Locals window. In addition, the fifth line displays the Slug property of the movie object on the sixth line. This also shows that the currently executing app is now using the new Name property.

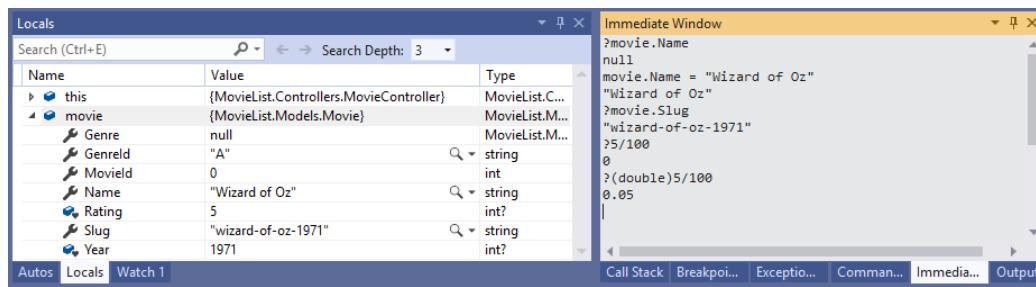
You can also use the Immediate window to test an expression. For example, the seventh line tests an expression that divides 5 by 100. Since this expression uses integer division, the result on the eighth line is 0. To use decimal division, you can adjust the expression so it casts one of the integer values to the double type as shown in the ninth line. This displays a result of 0.05.

In the Immediate window, you can execute a command that you've already entered by repeatedly pressing the Up or Down arrow to scroll through the commands. Then, when you display the command you want, you can execute it by pressing Enter. Or, if you want to remove all commands from the Immediate window, you can right-click on the window and select Clear All.

## The Locals and Watch windows



## The Immediate window



## How to use the Locals, Autos, and Watch windows

- The Locals window displays information about the variables within the scope of the current method.
- The Autos window works like the Locals window, but it only displays information about variables used by the current statement and the previous statement.
- The Watch windows let you view the values of variables and expressions that you specify, called *watch expressions*. To add a watch expression, type a variable name or expression into the Name column. To delete a row from a Watch window, right-click the row and select Delete Watch.

## How to use the Immediate window

- To display the current value of a variable or expression, type a question mark followed by a variable name or expression. Then, press Enter.
- To execute a statement, type the statement. Then, press Enter.
- To execute an existing command, press the Up or Down arrow until you have displayed the command. Then, press Enter.
- To remove all commands and output, right-click the window and select Clear All.

## How to display these windows

- If the window's tab is visible, click the tab. Otherwise, select the window from the Debug→Windows menu.

Figure 5-7 How to monitor variables and expressions

## How to use tracepoints

Visual Studio also provides a feature called *tracepoints*. A tracepoint is a special type of breakpoint that performs an action when it's encountered. For example, figure 5-8 begins by showing a tracepoint with its settings.

To set a tracepoint, you right-click a statement and select Breakpoint→Insert Tracepoint. Then, you use the Breakpoint Settings window to indicate what you want to do when the tracepoint is "hit." Typically, you want to log a message to the Output Window and continue execution.

The message you log can include variables and other expressions as well as special keywords. For example, the message shown here begins by using the \$FUNCTION keyword to log the name of the method that's being executed. Then, the message uses curly braces to include the value of the Name property of the movie object. As a result, the message that's logged to the Output window includes the name of the method and the name of the movie.

To help you understand how this works, this figure shows the Output window that displays logged messages. Here, the Output window displays several debugging messages that are logged by ASP.NET Core. In addition, it displays the message that's logged by the tracepoint shown in this figure that says:

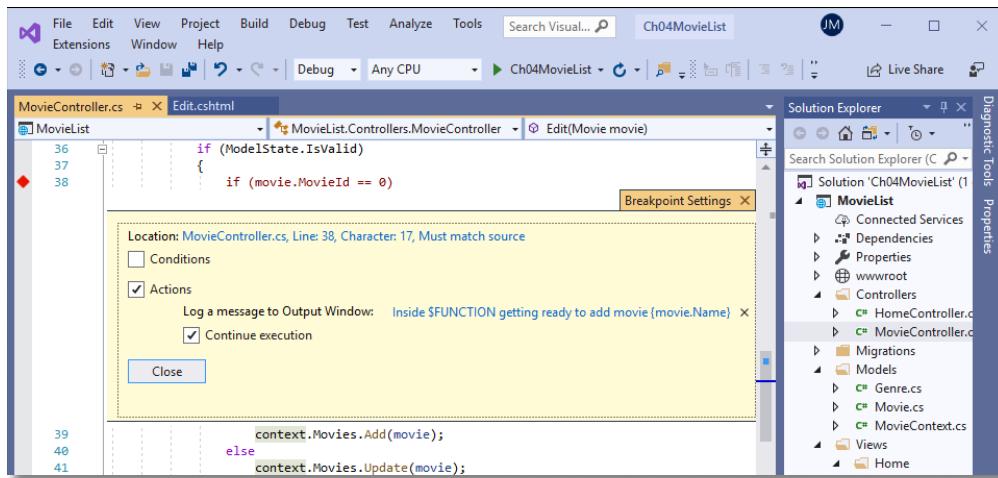
```
Inside MovieList.Controllers.MovieController.  
Edit(MovieList.Models.Movie) getting ready to add movie  
null
```

This message shows that the code is inside the Edit() method of the Movie controller and that the movie's name is null. If the Output window isn't displayed on your system, you can open it by selecting View→Output.

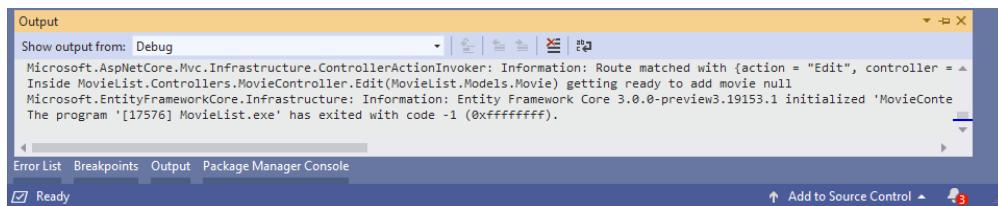
By default, program execution continues after the tracepoint action is performed. If that's not what you want, you can remove the check mark from the Continue Execution option. Since this causes the app to enter break mode, it converts the tracepoint into a breakpoint and uses the standard breakpoint icon (a red circle) instead of a tracepoint icon (a red diamond). Conversely, you can convert a breakpoint to a tracepoint by displaying its Breakpoint Settings window and selecting the Continue Execution option.

Tracepoints are useful in situations where a standard breakpoint would be cumbersome, such as in the execution of a loop. For example, suppose you have a loop that performs 100 iterations, and an error is occurring somewhere in the middle of the loop. Imagine how tedious it would be to manually continue execution until you get to the iteration that caused the error. By contrast, you can use a tracepoint to quickly view the data for each iteration of the loop.

## The Movie controller with a tracepoint that logs a message



## The Output window that displays the message



## How to set a tracepoint

- To set a new tracepoint, right-click a statement and select Breakpoint → Insert Tracepoint. Then, complete the Breakpoint Settings window.
- To convert an existing breakpoint to a tracepoint, point to the breakpoint icon, click the Settings icon that looks like a gear, and complete the Breakpoint Settings window.
- For a tracepoint, the Breakpoint Settings window should have the Continue Execution option selected. That way, it doesn't enter break mode like a breakpoint.
- When logging a message to the Output window, you can include the value of a variable or other expression by placing the variable or expression inside curly braces, and you can include special keywords such as FUNCTION by coding a dollar sign (\$) followed by the keyword.

## Description

- A *tracepoint* is a special type of breakpoint that lets you perform an action and continue execution.
- You typically use tracepoints to log messages to the Output window. These messages can include text, variables, expressions, and special keywords.
- Visual Studio uses a red diamond icon to mark tracepoints.

Figure 5-8 How to use tracepoints

## Perspective

Visual Studio provides a powerful set of tools for debugging ASP.NET Core MVC apps. For simple apps, you can usually find bugs by running the app and testing it manually. Then, you can use the techniques presented in this chapter to fix those bugs. For complex apps, though, it usually makes sense to automate testing as described in chapter 14.

## Terms

testing	debugger
debugging	breakpoint
browser incompatibilities	break mode
developer tools	data tip
F12 tools	watch expression
exception	tracepoint
stack trace	

## Summary

- When you *test* an app, you try to find all of its errors. When you *debug* an app, you find the causes of the errors and fix them.
- To test for *browser incompatibilities*, you run a web app in all of the most popular browsers to make sure they all display your app correctly.
- You can use the browser's *developer tools* to find problems in your HTML and CSS.
- *Exceptions* are errors that prevent the app from executing if they aren't handled.
- A *stack trace* is a list of methods that were active when an exception occurred.
- Visual Studio's *debugger* lets you set a *breakpoint*, step through the statements in an app when it is in *break mode*, and view the changes in the data after each statement is executed.
- *Data tips* provide an easy way to view the values of variables and expressions when an app is in break mode. However, you can also use the Autos, Locals, Watch, and Immediate windows to view the values of variables and expressions.
- *Tracepoints* are similar to breakpoints but they let you perform an action like printing a message to the Output window and then continue execution. They are often useful for debugging loops.

## Exercise 5-1 Debug the Movie List app

In this exercise, you'll debug the Movie List app presented in chapter 4. To start, you'll open a version of the app that includes two bugs.

### Use breakpoints, data tips, and the Locals window

1. Open the Ch05Ex1MovieList web app in the ex\_starts directory.
2. Press Ctrl+F5 to run the app without debugging. The default browser should display a list of movies, but the genre for each movie should not be correct.
3. Since the Index.cshtml file of the Views/Home folder displays the genre for each movie, open that file and view its code.
4. Set a breakpoint to the left of the table cell that displays the genre data.
5. Press F5 to run the app with debugging. This should cause the app to enter break mode just before the table cell that displays the genre data.
6. Hover the mouse pointer over @movie.Genre, expand the data tip that's displayed, and note that the Genre object has two properties (GenreId and Name) and that the Name property contains the genre data you want to display.
7. In the Locals window, expand the movie object, expand the Genre property, and note that it also displays the values of the GenreId and Name properties.
8. Change the code that displays the genre to @movie.Genre.Name and remove the breakpoint.
9. Press Ctrl+F5 to run the app again. This time, the browser should display the correct genre data for each movie.

### Use the Internal Server Error page and the Exception Helper

10. With the app still running, attempt to add a new movie, but leave the Name field blank and enter appropriate values for all the other fields.
11. When you click the Save button, the app should display an Internal Server Error page that displays information about an exception that indicates that the database can't insert a NULL value into the Name column of the Movies table.
12. Press F5 to run the app with debugging. Again, attempt to add a new movie with a blank Name field but appropriate values for all the other fields.
13. When you click the Save button, Visual Studio should enter break mode even though you haven't set a breakpoint. In addition, Visual Studio should highlight the statement that caused the exception and use the Exception Helper to display much of the same information about the exception as in step 11.
14. To prevent this exception, modify your app to make sure that it doesn't allow the name of a movie to be null. To do that, open the Movie class in the Models folder and add the Required attribute above the Name property like this:  
`[Required(ErrorMessage = "Please enter a name.")]`
15. Run the app again and confirm that the error has been fixed. To do that, attempt to add a movie with a blank Name field. This time, the browser should display a user-friendly validation message on the Add Movie page.



# Section 2

## Master the essential skills

The eight chapters in this section review and expand upon the skills that you learned in section 1. To start, chapter 6 shows how to work with controllers and routing to provide user-friendly URLs that can improve search engine optimization. Chapter 7 shows how to use Razor views to display the user interface of an app. And chapter 8 shows how to transfer data from controllers to views and back.

After learning those essential skills for working with controllers and views, chapter 9 shows how to use session state and cookies to manage the state of an app. Chapter 10 shows how to work with model binding. Chapter 11 shows how to validate data that's stored in the model. And chapter 12 shows how to use EF Core to store the data for the model in a database.

After presenting these skills, this section finishes by presenting the Bookstore app. This puts the skills presented in chapters 6 through 12 into the context of a complete app.

To a large extent, each of the chapters in this section is an independent unit. As a result, you don't have to read these chapters in sequence. If, for example, you want to know more about session state and cookies after you finish section 1, you can go directly to chapter 9. Eventually, though, you're going to want to read all eight chapters. So unless you have a compelling reason to skip around, we recommend reading the chapters in sequence.



# 6

## How to work with controllers and routing

In chapter 4, you learned how to create an app that uses the default routing that's available from ASP.NET Core MVC. To start, this chapter describes how this default routing works in more detail. Then, it shows how to customize the routing so the URLs for your web app follow best practices. To finish, this chapter summarizes some best practices for creating URLs.

<b>How to use the default route .....</b>	<b>198</b>
How to configure the default route .....	198
How the default route works.....	200
How to code a simple controller and its actions .....	202
How to code a controller that uses the id segment .....	204
<b>How to create custom routes .....</b>	<b>206</b>
How to include static content in a route .....	206
How to work with multiple routing patterns .....	208
<b>How to use attribute routing .....</b>	<b>210</b>
How to change the routing for an action .....	210
More skills for changing the routing for an action.....	212
How to change the routing for a controller.....	214
<b>Best practices for creating URLs .....</b>	<b>216</b>
<b>How to work with areas.....</b>	<b>218</b>
How to set up areas.....	218
How to associate controllers with areas .....	220
<b>Perspective .....</b>	<b>222</b>

## How to use the default route

This chapter begins by showing how to configure and use the default route for an ASP.NET Core MVC app. When you begin developing an app, it's common to start with this route.

### How to configure the default route

Figure 6-1 shows a Startup.cs file that configures the default route for an ASP.NET Core MVC app. To do that, it uses a routing system known as *endpoint routing* that was introduced in ASP.NET Core 2.2 and is recommended for development with ASP.NET Core 3.0 and later.

Before you can use endpoint routing, you need to add the necessary MVC services to the app. To do that, you can add one or more statements to the ConfigureServices() method. With ASP.NET Core 3.0 and later, you can call the AddControllersWithViews() method as shown in this figure.

Prior to ASP.NET Core 2.2, you had to use the AddMvc() method to add these services. However, this included some services such as Razor pages that aren't typically necessary for an MVC app. As a result, unless you need these services in your MVC app, it's better to use the AddControllersWithViews() method to add MVC services.

After you add the necessary services, you need to configure the routing. To do that, you can add the statements shown in this figure to the Configure() method. First, you call the UseRouting() method to mark where the routing decisions are made. Then, you call the UseEndpoints() method to configure the endpoints for each route.

Between the calls to the UseRouting() and UseEndpoints() methods, you can add any services that you want to run after routing decisions have been made but before they have been executed. This typically includes the services for authenticating and authorizing users that are described later in this book.

Within the UseEndpoints() method, you can call the MapControllerRoute() method to map the endpoints for each controller. To do that, you can supply a pattern argument like the one shown in this figure. This pattern specifies the default route described in the next few figures.

Alternately, you can call the MapDefaultControllerRoute() method to map the default route for controllers. This approach has the advantage of being shorter and easier to code. However, using the MapControllerRoute() method has the advantage of allowing other programmers to easily view and modify the pattern.

## The method for adding the MVC service

Method	Description
<code>AddControllersWithViews()</code>	Adds the services necessary to support an MVC app. Available with ASP.NET Core 2.2 and later.
<code>AddMvc()</code>	Adds the services necessary to support an MVC app. This includes some services that aren't necessary for MVC apps such as support for Razor pages. Available with older versions of ASP.NET.

## Two methods for enabling and configuring routing

Method	Description
<code>UseRouting()</code>	Selects the endpoint for the route if one is found.
<code>UseEndpoints(endpoints)</code>	Executes the endpoint selected by the routing.

## Two methods in the Startup.cs file that configure the default route

```
// Use this method to add services to the project.
public void ConfigureServices(IServiceCollection services) {
    services.AddControllersWithViews(); // add MVC services

    // add other services here
}

// Use this method to configure the HTTP request pipeline.
public void Configure(IApplicationBuilder app, IWebHostEnvironment env) {
    app.UseDeveloperExceptionPage();
    app.UseHttpsRedirection();
    app.UseStaticFiles();
    app.UseRouting(); // mark where routing decisions are made

    // configure middleware that runs after routing decisions have been made

    app.UseEndpoints(endpoints => // map the endpoints
    {
        endpoints.MapControllerRoute(
            name: "default",
            pattern: "{controller=Home}/{action=Index}/{id?}");
    });
}

// configure other middleware here
}
```

## Another way to map a controller to the default route

```
endpoints.MapDefaultControllerRoute();
```

### Description

- ASP.NET Core 2.2 introduced a new approach to routing known as *endpoint routing*. It's generally considered a best practice to use endpoint routing for all new development.
- Before you can use endpoint routing, you need to add the necessary MVC services to the app. Then, you need to mark where the routing decisions are made and configure the endpoints for each route.

Figure 6-1 How to configure the default route

## How the default route works

---

Figure 6-2 begins by showing a URL that has a domain name and three *segments*. Here, the first segment is “Home”, the second segment is “Index”, and the third segment is “Joel”.

The pattern for the *default route* specifies how to handle three segments like these. Here, the first segment specifies the controller, the second segment specifies the action method within the controller, and the third segment specifies an argument for the id parameter of the action method.

The pattern for the default route sets the Home controller as the default for the first segment, it sets the Index() action as the default for the second segment, and it uses a question mark (?) to specify that the third segment is optional. As a result, all three of these segments are optional. However, if you want to specify a later segment, you must also supply values for the earlier segments.

The table of request URLs shows how this works. Here, the first URL doesn’t specify any of the three segments. As a result, the app routes the request to the Index() method of the Home controller and it doesn’t pass an argument to the id parameter of this method.

Later in the table, the seventh URL specifies all three segments. As a result, the app routes the request to the List() method of the Product controller and passes an argument of “Guitars” to the id parameter of this method.

This table assumes that the List() method defines the id parameter as a string and doesn’t provide a default value for this parameter. As a result, if the URL doesn’t specify a third segment for the List() method, it sets the id parameter to null. Similarly, this table assumes that the Detail() method defines the id parameter as an int and doesn’t provide a default value for this parameter. As a result, if the URL doesn’t specify a third segment for the Detail() method, it sets the id parameter to the default int value of 0.

## A URL that has three segments

`https://localhost:5001/Home/Index/Joe`

First segment    Second segment    Third segment

## The pattern for the default route

`{controller=Home}/{action=Index}/{id?}`

## How the default route works

- The first segment specifies the controller. Since the pattern sets the Home controller as the default controller, this segment is optional.
- The second segment specifies the action method within the controller. Since the pattern sets the Index() method as the default action, this segment is optional.
- The third segment specifies an argument for the id parameter of the action method. The pattern uses a question mark (?) to specify that this segment is optional.

## How request URLs map to controller classes and their action methods

Request URL	Controller	Action	Id
<code>http://localhost</code>	Home	Index	null
<code>http://localhost/Home</code>	Home	Index	null
<code>http://localhost/Home/Index</code>	Home	Index	null
<code>http://localhost/Home/About</code>	Home	About	null
<code>http://localhost/Product</code>	Product	Index	null
<code>http://localhost/Product&gt;List</code>	Product	List	null
<code>http://localhost/Product&gt;List/Guitars</code>	Product	List	Guitars
<code>http://localhost/Product/Detail</code>	Product	Detail	0
<code>http://localhost/Product/Detail/3</code>	Product	Detail	3

## Description

- The *default route* maps a request to an action method within a controller and can optionally pass an argument to that action method.

---

Figure 6-2 How the default route works

## How to code a simple controller and its actions

---

Figure 6-3 shows how to code a simple Home controller and two of its actions. To start, the class for an MVC controller typically inherits the Controller class from Microsoft's AspNetCore.Mvc namespace. In this figure, for example, the HomeController inherits the Controller class.

Within the class for a controller, an action method typically returns an object that implements the IActionResult interface, such as a ContentResult or ViewResult object. For example, in chapters 2 and 4, you saw how to use the View() method to return ViewResult objects.

Now, this chapter shows how to use the Content() method to return a ContentResult object. This allows you to return plain text directly to the browser without going through a view, which is a useful way to test the URLs of your app before you implement the views. For example, you can test the controller shown in this chapter by running the app and entering a URL in the browser's address bar as shown in this figure.

## A method that a controller can use to return plain text to the browser

Method	Description
<code>Content(string)</code>	Creates a ContentResult object that contains the specified string.

## The Home controller

```
using Microsoft.AspNetCore.Mvc;

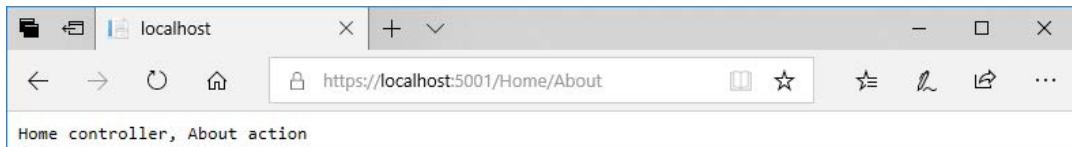
namespace GuitarShop.Controllers
{
    public class HomeController : Controller
    {
        public IActionResult Index()
        {
            return Content("Home controller, Index action");
        }

        public IActionResult About()
        {
            return Content("Home controller, About action");
        }
    }
}
```

## A browser after requesting the default page



## A browser after requesting the Home/About page



## Description

- The class for an MVC controller typically inherits the `Controller` class from Microsoft's `AspNetCore.Mvc` namespace.
- The action methods for a controller typically return an object that implements the `IActionResult` interface such as a `ContentResult` or `ViewResult` object.
- To create a simple controller, you can begin by coding action methods that use the `Content()` method to return `ContentResult` objects that contain plain text.
- To test a controller, you can run the app and enter a URL in the browser's address bar.

---

Figure 6-3 How to code a simple controller and its actions

## How to code a controller that uses the id segment

---

Figure 6-4 shows how to code a controller named ProductController that uses the id segment of the default route. Here, the Product controller begins much like the Home controller from the previous figure. However, both of its action methods provide a parameter named id that maps to the id segment of the default route.

The List() method uses the string type for the id parameter and provides a default value of “All”. As a result, if a URL doesn’t specify the id segment for this action, this method sets the id parameter to “All” as shown by the first browser window.

On the other hand, the Detail() method uses the int type for the id parameter and does not provide a default value. As a result, if a URL specifies the id segment for this action, this method sets the id parameter to the segment value as shown by the second browser window. Otherwise, this method sets the id parameter to the default int value of 0 as shown by the third browser window.

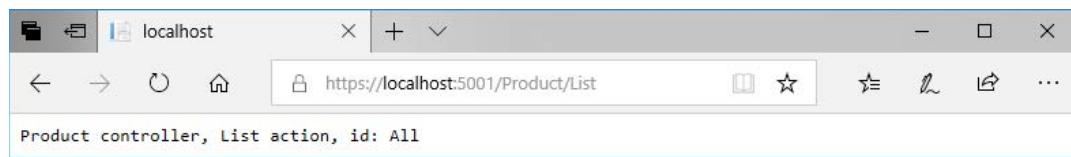
## The Product controller

```
using Microsoft.AspNetCore.Mvc;

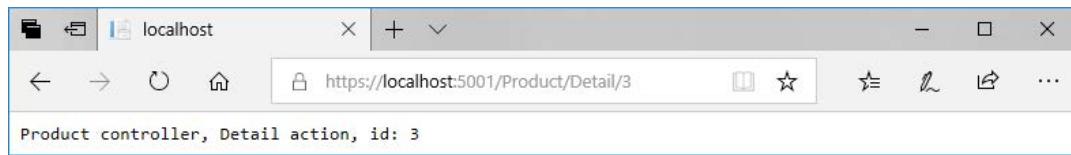
namespace GuitarShop.Controllers
{
    public class ProductController : Controller
    {
        public IActionResult List(string id = "All")
        {
            return Content("Product controller, List action, id: " + id);
        }

        public IActionResult Detail(int id)
        {
            return Content("Product controller, Detail action, id: " + id);
        }
    }
}
```

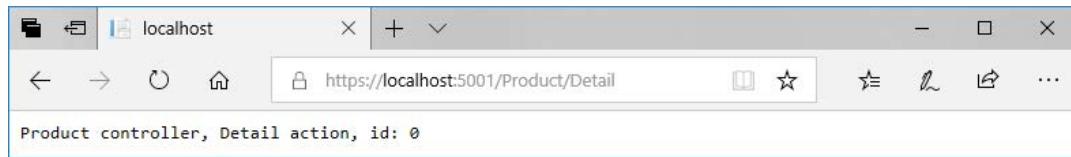
### A browser after requesting the Product/List action with no id segment



### A browser after requesting the Product/Detail action with an id segment



### A browser after requesting the Product/Detail action with no id segment



## Description

- The List() method uses the string type for the id parameter and provides a default value of “All”.
- The Detail() method uses the int type for the id parameter and does not provide a default value.

---

Figure 6-4 How to code a controller that uses the id segment

## How to create custom routes

---

Now that you understand how the default route works, you're ready to learn how to create custom routes. One way to do that is to add static content to a route.

### How to include static content in a route

---

All of the segments in the default pattern are dynamic. In other words, they specify content that can change with each URL. However, to make a URL easier to read, you may want to include static content as part of a segment. To do that, you can use a string literal as shown in the first example in figure 6-5. Here, the fourth segment of the pattern includes a string literal of "Page" just before it uses {num} to specify the name of the parameter. As a result, a segment of "Page4" passes an argument of "4" to the parameter named num.

In the pattern for the first example, none of the segments are optional or provide default values. As a result, you must supply all four segments. For the first two segments, you must specify the controller and action method. For the third segment, you must supply a category. And for the fourth segment, you must supply the static content of "Page" followed by the number for the page.

If you specify all four segments, the action method that's matched to the request URL can use its parameters to get the data from the third and fourth segments. For example, the List() method in this figure displays the data that's available from its cat and num parameters. If you don't specify all four segments, the routing system doesn't match the URL to that pattern.

If you don't want to mix static and dynamic content in a segment, you can achieve a similar result by coding a routing pattern that specifies a completely static segment followed by a completely dynamic segment. In this figure, for example, the second routing pattern uses this approach. Here, the fourth segment is completely static and the fifth segment is completely dynamic. This works similarly to the first routing pattern. However, it uses five segments instead of four.

## A pattern that mixes static and dynamic data for a segment

{controller}/{action}/{cat}/Page{num} // 4 segments

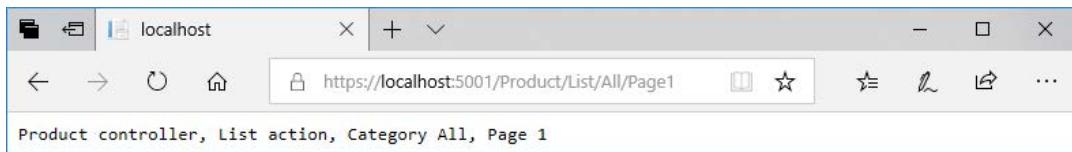
### Example URLs

Request URL	Controller	Action	Parameters
/Product>List>All\Page1	Product	List	cat>All, num=1
/Product>List>All\Page2	Product	List	cat>All, num=2

### The List() method of the Product controller

```
public IActionResult List(string cat, int num)
{
    return Content("Product controller, List action, " +
        "Category " + cat + ", Page " + num);
}
```

### A URL that requests Product/List action for page 1 of all categories



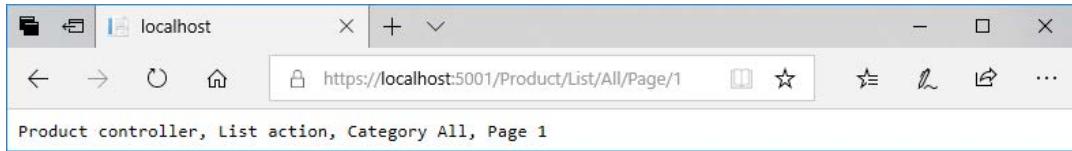
## A pattern that supplies one segment that's completely static

{controller}/{action}/{cat}/Page/{num} // 5 segments

### Example URLs

Request URL	Controller	Action	Parameters
/Product>List>All\Page/1	Product	List	cat>All, num=1
/Product>List>All\Page/2	Product	List	cat>All, num=2

### A URL that requests Product/List action for page 1 of all categories



### Description

- To include static content as part of a segment, you can use a string literal to provide the static part of the segment and use braces to identify the dynamic part of the segment.
- To include a static segment, you can code a string literal for the entire segment.

Figure 6-5 How to include static content in a route

## How to work with multiple routing patterns

---

When you create custom routes, it's common to use them in addition to the default route and other custom routes. When you map multiple routing patterns, you must code the most specific pattern first and the most general pattern last. Otherwise, the most general pattern will process all URLs.

The example in figure 6-6 shows how this works. To start, the first routing pattern is the most specific because it specifies five required segments where the fourth and fifth segments include static content for paging and sorting. As a result, a URL only matches this pattern if it includes five segments and matches the static content for paging and sorting.

The second routing pattern works like the first routing pattern, except that it doesn't specify the fifth segment for sorting. In other words, it only provides a fourth segment for paging. As a result, a URL only matches this pattern if it includes four segments and matches the static content for paging.

The third routing pattern is the least specific because it specifies the third segment as being optional and provides default values for the first two segments. As a result, a URL matches this pattern if it contains one, two, or three segments.

The `List()` method shown in this figure shows how you can access the arguments in the URL segments. Here, the first parameter is named `id` so it can access the `id` argument specified by the third segment. The second parameter is named `page` so it can access the `page` argument specified by the fourth segment. And the third parameter is named `sortby` so it can access the `sortby` argument specified by the fifth segment. The tables at the bottom of this figure show how this works.

## Code that maps three routing patterns to controllers

```
app.UseEndpoints(endpoints =>
{
    // most specific route - 5 required segments
    endpoints.MapControllerRoute(
        name: "paging_and_sorting",
        pattern: "{controller}/{action}/{id}/page{page}/sort-by-{sortby}");

    // specific route - 4 required segments
    endpoints.MapControllerRoute(
        name: "paging",
        pattern: "{controller}/{action}/{id}/page{page}");

    // least specific route - 0 required segments
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}");
});
```

## The List() method of the Product controller

```
public IActionResult List(string id = "All", int page = 1,
    string sortby = "Price")
{
    return Content("id=" + id + ", page=" + page + ", sortby=" + sortby);
}
```

## Examples that use the default route

Request URL	Controller/Action	Parameters
/	Home/Index	id=null
/Home/About	Home/About	id=null
/Product/Detail/4	Product/Detail	id=4
/Product>List/Guitars	Product/List	id=Guitars

## Examples that uses the paging route

Request URL	Parameters
/Product>List/All/Page3	id=All, page=3
/Product>List/Guitars/Page2	id=Guitars, page=2
/Product>List/Guitars/Pg2	Not found because "Pg" doesn't match static "Page".

## Example URLs that use the paging\_and\_sorting route

Request URL	Parameters
/Product>List/Guitars/Page2/Sort-By-Name	id=Guitars, page=2, sortby=Name
/Product>List/Guitars/Page2/By-Name	Not found because "By-" doesn't match "Sort-By-".

## Description

- When you map multiple routing patterns, you must code the most specific ones first and the most general one last. If you code the most general pattern first, it will process all URLs.

Figure 6-6 How to work with multiple routing patterns

## How to use attribute routing

So far, this chapter has shown how to use the Startup.cs file to specify the routing patterns for an app. However, you can override this routing by adding Route attributes to the action methods of a controller or to the class for the controller. This is known as *attribute routing*.

### How to change the routing for an action

Figure 6-7 starts by showing the class for the Home controller after attribute routing has been applied to its Index() and About() actions. To change the routing for the Index() action, this code places a Route attribute directly above the Index() method and specifies a route of “/”. This maps the Index() action to a URL that requests the root folder of the app. Similarly, this code places a Route attribute directly above the About() method that maps the About() action to a request URL of “About”.

These Route attributes specify static routes, not patterns. In addition, they override any routing patterns that are specified by the Startup.cs file. As a result, if you add these attributes, you can't use a route of /Home, /Home/Index, or /Home/About. In other words, to access the Home/Index action, you must use a request URL of /. Similarly, to access the Home/About action, you must use a request URL of /About. Since that leads to shorter URLs, that's often what you want.

When you specify a static URL such as “About” for a route, the route doesn't change if you change the name of the About() action method later. That may be what you want. However, it's common to want the route to match the name of the action. To make this possible, you can use the [action] token in the Route attribute instead of hard coding the name in the route. Then, if you change the name of the action method, the route changes to match that name, which is often what you want. Since this makes your code more flexible, it's generally considered a best practice.

The [controller] token works much like the [action] token. Although this figure doesn't present an example of the [controller] token, the next two figures do.

With ASP.NET Core 3.0 or later, attribute routing is typically enabled by default. As a result, you typically don't need to add any code to the Startup.cs file to enable it. However, if you find that attribute routing isn't working correctly on your system, you may need to enable it by calling the MapControllers() method as shown in the last example. Here, the MapControllers() method is coded before the pattern for the default route. That way, any attribute routing overrides the default routing pattern.

## The Home controller with attribute routing for both actions

```
public class HomeController : Controller
{
    [Route("/")]
    public IActionResult Index()
    {
        return Content("Home controller, Index action");
    }

    [Route("About")]
    public IActionResult About()
    {
        return Content("Home controller, About action");
    }
}
```

## How request URLs map to controller classes and their action methods

Request URL	Description
/	This maps to the Home/Index action.
/About	This maps to the Home/About action.

## Two default routes that are overridden by the attribute routing

```
/Home/Index  
/Home/About
```

## Two tokens you can use to insert variable data into a route

Token	Description
[controller]	The name of the current controller.
[action]	The name of the current action.

## A more flexible way to code the attribute for the Home/About action

```
[Route("[action])"]
```

## How to map all controllers that use attribute routing

```
app.UseEndpoints(endpoints =>
{
    // map controllers that use attribute routing - often not necessary
    endpoints.MapControllers();

    // map pattern for default route
    endpoints.MapControllerRoute(
        name: "default",
        pattern: "{controller=Home}/{action=Index}/{id?}");
});
```

## Description

- To change the routing for an action, you can code a Route attribute directly above the action method. Then, the route specified by the attribute overrides the route that's specified in the Startup.cs file. This is known as *attribute routing*.
- To specify a static route, you can code a string literal within the Route attribute.
- To insert the name of the current controller or action into a route, you can use the [controller] or [action] tokens.

Figure 6-7 How to use attributes to change the routing for an action

## More skills for changing the routing for an action

---

Figure 6-8 begins by showing the class for the Product controller after attribute routing has been applied to its List() and Detail() actions. To do that, this class places a Route attribute directly above both of these action methods.

In addition, this class contains a helper method named GetSlug(). To indicate that this method is not an action method, this figure places the NonAction attribute directly above it. This prevents a URL from using the default routing to call the GetSlug() method. Since there's no reason to allow that, this is considered a best practice.

For the List() action, the Route attribute specifies a URL where the first segment is "Products" and the second segment is an optional argument named cat. As a result, a request that specifies /Products calls this action without passing an argument, and a URL that specifies /Products/Guitars calls this action and passes an argument of "Guitars". The advantage of this approach over the default routing pattern is that it makes the URLs for accessing this action slightly shorter (/Products vs. /Product/List).

For the Detail() action, the Route attribute specifies a URL where the first segment is "Product" and the second segment is a required argument named id. As a result, a request that specifies /Product doesn't map to this action because it doesn't pass the required id argument. This causes the request to return a page not found error. However, a request of /Product/3 calls this action and passes an id argument of 3. Again, the advantage of this approach over the default routing pattern is that it makes the URLs for accessing this action slightly shorter (/Product vs. /Product/Detail).

The last two examples in this figure show that you can make the attribute routing more flexible by using the [controller] token. For example, you can code the [controller] token followed by an s for the first segment of the List() action. This adds the s to the end of the controller name (Product) to make the segment Products. And you can code the [controller] token by itself for the first segment of the Detail() action. This makes the segment Product.

As you review this figure, note that you can use many of the same skills in the Route attributes that you used in the URL patterns in Startup.cs file. For example, you can use a question mark (?) to identify an optional argument. Similarly, you can add static content to a segment by entering a string literal for the static content.

## The Product controller with attribute routing that specifies segments

```
public class ProductController : Controller
{
    [Route("Products/{cat?}")]
    public IActionResult List(string cat = "All")
    {
        return Content("Product controller, List action, Category: " + cat);
    }

    [Route("Product/{id}")]
    public IActionResult Detail(int id)
    {
        return Content("Product controller, Detail action, ID: " + id);
    }

    [NonAction]
    public string GetSlug(string name)
    {
        return name.Replace(' ', '-').ToLower();
    }
}
```

## How request URLs map to controller classes and their action methods

Request URL	Description
/Products	This maps to the Product/List action and uses the default parameter value of "All".
/Products/Guitars	This maps to the Product/List action and passes an argument of "Guitars".
/Product/3	This maps to the Product/Detail action and supplies a valid int argument of 3.
/Product	This URL is not found. It does not map to the Product/Detail action because it does not supply the required id segment.

## Two default routes that are overridden by the attribute routing

```
/Product/List  
/Product/Detail
```

## A more flexible way to code the attribute for the Product/List action

```
[Route("[controller]s/{cat?}")]
```

## A more flexible way to code the attribute for the Product/Detail action

```
[Route("[controller]/{id}")]
```

### Description

- To insert other segments into a route, you can use all of the skills for coding segments such as the skills described earlier in this chapter.
- If a controller contains methods that aren't action methods, you can code the NonAction attribute above it to prevent it from being mapped to a URL.

---

Figure 6-8 More skills for using attributes to change the routing for an action

## How to change the routing for a controller

---

In the last two figures, you learned how to use attribute routing to override the routing for a single action in a controller. However, it's also possible to use attribute routing to override the routing for all actions within a controller. To do that, you can code a Route attribute directly above the declaration for the controller's class as shown in figure 6-9.

Here, the Route attribute that's coded above the class declaration specifies four segments. The first segment is a static segment of "Retail", the second segment is the name of the controller, the third segment is the name of the action, and the fourth segment is an optional argument named id. As a result, you can use the URLs shown in the table to call the action methods of this controller and to pass arguments to them.

As with actions, the attribute routing for a controller overrides the default routing. As a result, when you use attribute routing for the controller, you can no longer use the default routing to request the actions of the controller. In this figure, for example, you can't use /Product/List to request the Product/List action. Instead, you must use /Retail/Product/List.

## The code for the Product controller

```
[Route("Retail/{controller}/{action}/{id?}")]
public class ProductController : Controller
{
    public IActionResult List(string id = "All")
    {
        return Content("Product controller, List action, Category: " + id);
    }

    public IActionResult Detail(int id)
    {
        return Content("Product controller, Detail action, ID: " + id);
    }
}
```

## How request URLs map to controller classes and their action methods

Request URL	Description
/Retail/Product/List	This maps to the Product/List action and uses the default parameter value of “All”.
/Retail/Product/List/Guitars	This maps to the Product/List action and passes an argument of “Guitars”.
/Retail/Product/Detail	This maps to the Product/Detail action and uses the default int value of 0.
/Retail/Product/Detail/3	This maps to the Product/Detail action and passes a valid int argument of 3.

## Two default routes that are overridden by the attribute routing

```
/Product/List
/Product/Detail
```

### Description

- To change the routing for all actions in a controller, you can code a Route attribute directly above the declaration for the controller’s class.

Figure 6-9 How to use attributes to change the routing for a controller

## Best practices for creating URLs

As you design the URLs for your app, you should realize that they specify the interface for your app. As a result, once a website goes into production, you shouldn't change its URLs. If you do, you should redirect the old URLs to the new ones that provide the same functionality.

It's important to put some effort into designing the URLs of an app. Figure 6-10 lists some best practices for URLs. You'll want to follow these practices for a couple of reasons.

First, well-designed URLs can improve the usability of your app for both developers and end users. For example, if you want to cut and paste a URL to share with others, a well-designed URL is easier to read and gives others confidence that it leads to the content that they want. Also, if you need to type a URL or read it to someone over the phone, it's easier to do with a well-designed URL that's short and uses hyphens than it is with a long URL that uses long identifier values.

Second, well-designed URLs can improve the search engine optimization (SEO) for your app. These days, search engines don't rely on the URL for SEO as much as they used to. Still, you can improve your SEO by including keywords that describe the content of the page in your URL.

The first three examples show three ways to identify a product. The first example uses a number of 1307 to identify the product. The second example uses a *slug*, which is a descriptive string, to identify a product. Between these two, using a slug is more user-friendly, but it usually makes database queries less efficient. As a result, it often makes sense to use a number and a slug as shown by the third example. That way, you can use the number to query the database, but the slug still makes the URL user-friendly.

The last three examples compare bad practices to best practices. To start, the fourth example shows four URLs that use query strings to pass data to the controller. This isn't recommended because query strings use the ? and & characters, which make them more difficult to read than URLs that use segments to pass data to the controller, as shown so far in this chapter. In addition, these URLs use inconsistent capitalization, they use numbers to identify the products, and they don't use keywords.

The fifth example fixes most of these issues by following best practices. It uses segments to pass the same data. It uses all lowercase letters, which are easier to type than uppercase letters. It uses keywords such as "product" and "page". And it uses hyphens to separate words such as "page-1" and "fender-stratocaster". Using hyphens to separate words like this is known as *kebab case* because the hyphens look like the skewer in a shish kebab, and the letters look like the meat that's on the skewer.

The sixth example works much like the fifth example, but its URLs are even shorter. That's because this example uses /products instead of /product/list, which is four characters shorter. Similarly, it uses /product instead of /product/detail, which is six characters shorter. That might not seem like a big deal, but if a website is being used by millions of people, it's important to make the URLs as short and easy to use as possible.

## Best practices for URLs

- Keep the URL as short as possible while still being descriptive and user-friendly.
- Use keywords to describe the content of a page, not implementation details.
- Make your URLs easy for humans to understand and type.
- Use hyphens to separate words, not other characters, especially spaces.
- Prefer names as identifiers over numbers.
- Create an intuitive hierarchy.
- Be consistent.
- Avoid the use of query string parameters if possible.

### A URL that identifies a product...

#### With a number

```
https://www.domain.com/product/1307
```

#### With a name

```
https://www.domain.com/product/fender-special-edition-standard-stratocaster
```

#### With a number and name (to keep it descriptive but short)

```
https://www.domain.com/product/1307/fender-special
```

### Four URLs that use query strings to pass data (not recommended)

```
https://www.murach.com/p>List?  
/p>List?catId=1  
/p>List?catId=1&pg=1  
/p/Detail?id=1307
```

### Four URLs that follow best practices

```
https://www.murach.com/product/list  
/product/list/guitars  
/product/list/guitars/page-1  
/product/detail/1307/fender-stratocaster
```

### Four URLs that follow best practices, but are even shorter

```
https://www.murach.com/products  
/products/guitars  
/products/guitars/page-1  
/product/1307/fender-stratocaster
```

## Description

- Well-designed URLs can improve the usability of your app for both developers and end users.
- Well-designed URLs can improve the search engine optimization (SEO) for your app.
- A *slug* is a string that describes the content of a page. Using a slug can make your URLs more user-friendly.
- Using hyphens to separate words is known as *kebab case*.

---

Figure 6-10 Best practices for URLs

## How to work with areas

An ASP.NET Core MVC app can have multiple *areas*. Each area can have its own controllers, models, and views. This can help you organize the folders and files of an app. For example, you may want to create one area that allows administrators to perform tasks such as adding and updating the products that are stored in a database. Then, you can use another area to allow customers to browse through the products and add them to a cart.

### How to set up areas

Figure 6-11 shows how to set up areas. By convention, you can create an area by adding a folder named Areas to the root folder for the app. Then, within the Areas folder, you can add a folder for the area. In this figure, for example, the folders and files for the Guitar Shop app include an area named Admin.

Within the folder for the area, you can add the necessary subfolders such as the Controllers, Models, Views, and so on. In this figure, for example, the Admin folder includes the Controllers and Views folders. Since this area is separate from the main area, it can use the same names for the controller and view files. For example, this area has a Home controller and a Home/Index view as well as a Product controller and a Product/List view. However, the Product/List view in the Admin area allows admin users to add, update, and delete products, but the Product/List view in the default area only allows customers to view products and add them to their carts.

To get the views for the Admin area to work correctly, it includes a layout named \_AdminLayout that's shared by all of the views in this area. Then, it uses a \_ViewImports file to import the same models as the default area. That way, the views can use the same Product and Category models defined in the default area. Finally, this area uses a \_ViewStart file to specify \_AdminLayout as the default layout. When you're setting up the folders and files for an Admin area, that's usually what you want. But don't worry if you don't understand the details of how these view files work. You'll learn more about them in the next chapter.

Once you've set up the folders and files for an area, you need to configure the routes for the app so they include the area. To do that, you can use the MapAreaControllerRoute() method. This method works much like the MapControllerRoute() method. However, you must use the second argument to specify a name for the area. In addition, you can use the third argument to specify a pattern for the route. In this figure, for example, the pattern uses a static segment of "Admin" to specify the first segment of the route for the Admin area. After that, the route works like the default route.

## The starting folders for a Guitar Shop app that includes an Admin area

```

GuitarShop
  /Areas
    /Admin
      /Controllers
        /HomeController.cs
        /ProductController.cs
      /Views
        /Home
          /Index.cshtml
        /Product
          /List.cshtml
          /AddUpdate.cshtml
          /Delete.cshtml
        /Shared
          /_AdminLayout.cshtml
          _ViewImports.cshtml
          _ViewStart.cshtml
    /Controllers
      /HomeController.cs
      /ProductController.cs
    /Models
      /Product.cs
      /Category.cs
    /Views
      /Home
        /Index.cshtml
        /About.cshtml
      /Product
        /List.cshtml
        /Detail.cshtml
      /Shared
        /_Layout.cshtml
        _ViewImports.cshtml
        _ViewStart.cshtml
  Program.cs
  Startup.cs

```

## A route in the Startup.cs file that works with an area

```

app.UseEndpoints(endpoints =>
{
  endpoints.MapAreaControllerRoute(
    name: "admin",
    areaName: "Admin",
    pattern: "Admin/{controller=Home}/{action=Index}/{id?}");

  endpoints.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
});

```

### Description

- An ASP.NET Core MVC app can have multiple *areas*. Each area can have its own controllers, models, and views. This can help you organize the folders and files of an app.
- To configure the route for an area, you can use the `MapAreaControllerRoute()` method to add a route that specifies the name of the area and its routing pattern.

Figure 6-11 How to set up areas

## How to associate controllers with areas

---

Before you can use a controller with an area, you must use the Area attribute to associate it with an area. To do that, you can add an Area attribute to the controller as shown in figure 6-12. This Area attribute must match the area name specified by the area route in the Startup.cs file. In this figure, for example, the area name of “Admin” specified by the controller matches the area name of “admin” specified by the Startup.cs file.

After you add the Area attribute to the controller, you can use the [area] token to specify attribute routing for the controller. In this figure, for instance, the second example uses the [area] token to apply attribute routing to the List action of the Product controller. This attribute routing specifies the name of the area as the first segment, the name of the controller plus an s as the second segment, and an optional id parameter as the third segment. As a result, you can’t use the area routing of /Admin/Product/List to call the Product/List action. Instead, you must use the attribute routing of /Admin/Products.

To help you visualize how this works, this figure presents a table of URLs and describes how they work. Here, the first two URLs use the attribute routing to map to the Product/List action of the Admin area. As a result, they use two or three segments to call this action and pass it an argument. However, the next two elements use the routing specified in the Startup.cs file to map to other actions in the Product controller of the Admin area. As a result, they use three or four segments to call an action and pass it an argument.

## The Home controller for the Admin area

```
namespace GuitarShop.Areas.Admin.Controllers
{
    [Area("Admin")]
    public class HomeController : Controller
    {
        public IActionResult Index()
        {
            return View(); // maps to /Areas/Admin/Views/Home/Index.cshtml
        }
    }
}
```

## A token you can use to insert the area into a route

Token	Description
[area]	The name of the current area.

## The Product controller for the Admin area

```
namespace GuitarShop.Areas.Admin.Controllers
{
    [Area("Admin")]
    public class ProductController : Controller
    {
        [Route("[area]/[controller]s/{id?}")]
        public IActionResult List(string id = "All") {...};

        public IActionResult Add() {...};
        public IActionResult Update(int id) {...};
        public IActionResult Delete(int id) {...};
    }
}
```

## How request URLs map to controller classes and their action methods

Request URL	Description
/Admin/Products	This maps to the Product/List action and uses the default parameter value of “All”.
/Admin/Products/Guitars	This maps to the Product/List action and passes an argument of “Guitars”.
/Admin/Product/Add	This maps to the Product/Add action.
/Admin/Product/Update/3	This maps to the Product/Detail action and passes an id argument of 3.

## Description

- Before you can use a controller with an area, you must use the Area attribute to associate it with an area.
- After you add the Area attribute to the controller, you can use the [area] token to specify attribute routing for the controller.

Figure 6-12 How to associate controllers with areas

## Perspective

---

Now that you've read this chapter, you should have a general understanding of how to set up the controllers and routing of an MVC app. In addition, you should have a good idea of how to follow best practices to create well-designed URLs for a web app. With that as background, you're ready to learn more about developing the views for an MVC app. That's why the next chapter presents some of the most useful skills for working with views. As you'll see, many of these skills are interrelated with the skills for working with controllers and routing.

Of course, there's more to controllers and routing than what's presented in this chapter. For example, you may want to constrain a segment so it only matches certain data types. You may want to map legacy URLs to the new routing system. Or, you may want to specify a pattern that supports a variable number of segments. When you're first getting started, you typically don't need these more advanced routing skills. If you find that you do need them, you should have the foundation you need to learn more about them. To do that, you can begin by searching the web.

## Terms

---

endpoint routing	slug
URL path segment	kebab case
default route	areas of an app
attribute routing	

## Summary

---

- ASP.NET Core 2.2 introduced a new approach to routing known as *endpoint routing*. With ASP.NET Core 3.0 and later, it's generally considered a best practice to use endpoint routing.
- The path for a URL consists of zero or more *segments* where each segment is separated by a slash (/).
- The *default route* maps a request to an action method within a controller and can optionally pass an argument to that action method.
- Using the *Route* attribute to specify the routing for a controller or its actions is known as *attribute routing*. Attribute routing overrides any routing patterns specified in the Startup.cs file.
- A *slug* is a string that describes the content of a page.
- Using hyphens to separate words is known as *kebab case*.
- An ASP.NET Core MVC app can have multiple *areas*. Each area can have its own controllers, models, and views.

## Exercise 6-1 Practice routing

In this exercise, you'll review the essential skills for working with routing.

### View and test the default route for the app

1. Open the Ch06Ex1RoutingPractice web app in the ex\_starts directory.
2. Open the Startup.cs file and view its code. Note that it includes the default route:  
`{controller=Home}/{action=Index}/{id?}`
3. Open the HomeController class in the Controllers folder and view its code. Note that the Index() action method displays text that says "Home" and the Privacy() action method displays text that says "Privacy".
4. Run the app. This should start your browser and automatically call this URL:  
`https://localhost:5001`  
This should display text that says, "Home".
5. Enter URLs in the browser to display the text returned by the Index() and Privacy() methods. For example, try these URLs:  
`https://localhost:5001/home`  
`https://localhost:5001/home/privacy`
6. In the HomeController class, view the code for the Display() action method. Note that this method accepts a string parameter named id. Review the code that works with this parameter.
7. Run the app and test the Display() method by entering these URLs:  
`https://localhost:5001/home/display`  
`https://localhost:5001/home/display/123abc`

The first URL should display a message that indicates that the id hasn't been supplied, and the second URL should display a message that says, "ID: 123abc".

### Add an action method that uses the default route

8. In the HomeController class, add the following action method. When you do, make sure to declare a parameter of the int type with a name of id and a default value of 0 like this:

```
public IActionResult Countdown(int id = 0)
{
    string contentString = "Counting down:\n";
    for(int i = id; i >= 0; i--)
    {
        contentString += i + "\n";
    }
    return Content(contentString);
}
```

9. Run the app and enter a URL that calls the Countdown() method like this:

```
https://localhost:5001/home/countdown
```

This should display “Counting down.” followed by a 0 on the next line. That’s because the URL omits the optional third segment of the default route. As a result, the method uses the default value of 0 for the id parameter.

10. Run the app and enter a URL that calls the Countdown() method like this:

```
https://localhost:5001/home/countdown/10
```

This should display a countdown that starts at 10 and counts down to 0 with each integer on its own line. That’s because the URL included a value of 10 for the third segment.

### Use attribute routing to customize the route for an action method

11. Add a Route attribute immediately above the Countdown() method like this:

```
[Route("[action]/{id?}")]
public IActionResult Countdown(int id = 0) {....}
```

12. Run the app and enter a URL that calls the Countdown() method like this:

```
https://localhost:5001/countdown/10
```

Note that you only have to type the name of the action (countdown) and the value of the id segment (10) in this URL, not the name of the controller (home).

13. In the Countdown() method, change the name of the parameter from id to num. However, in the Route attribute, continue to use id as the name of the second segment.

14. Run the app and test the Countdown() method without an id segment. It should work correctly.

15. Test the Countdown() method with an id segment of 10 as before. This should only display the default value of 0. That’s because the segment name in the Route attribute must match the parameter name in the method.

16. In the Route attribute, change the second segment from {id?} to {num?}.

17. Run the app and test the CountDown() method with a second segment of 10. This should display a countdown from 10 to 0.

### Add more segments to a route

18. Edit the Route attribute for the Countdown() method like this:

```
[Route("[action]/{start}/{end?}/{message?}")]
public IActionResult Countdown(int start, int end = 0,
    string message = "")
```

Note that the segment names in the Route attribute match the parameter names in the action method and that the first two segments are required but the third and fourth are optional.

19. Modify the code for the Countdown() method so it starts counting down at the start parameter, ends at the end parameter, and displays the message parameter when the countdown is done.

20. Run the app and enter a URL that calls the Countdown() method like this:

**https://localhost:5001/countdown/3/1/Liftoff!**

This should display a countdown like this:

```
Counting down:  
3  
2  
1  
Liftoff!
```

21. Enter a URL that calls the Countdown() method like this:

**https://localhost:5001/countdown**

The browser should not be able to find the web page. That's because the second segment is now required. As a result, you must specify the start segment when you enter this URL.

22. Enter a URL that calls the Countdown() method like this:

**https://localhost:5001/countdown/5**

This should use the default parameters of the Countdown method to display a countdown from 5 to 0 with no message.



# 7

## How to work with Razor views

In chapters 2 and 4, you learned the basic skills for using Razor views and layouts. Now, this chapter reviews some of those skills and expands upon them. When you're done, you should have all the skills you need to work with Razor views and layouts, including how to bind them to a model.

<b>How to use Razor syntax .....</b>	<b>228</b>
How to work with code blocks and inline expressions .....	228
How to code inline loops .....	230
How to code inline conditional statements .....	232
<b>Essential skills for Razor views .....</b>	<b>234</b>
The starting folders and files for an app .....	234
How to code controllers that return views .....	236
How to create a default layout and enable tag helpers .....	238
How to use tag helpers to generate URLs for links .....	240
Three views that use the default layout and tag helpers .....	242
The three views displayed in a browser .....	244
<b>More skills for Razor views .....</b>	<b>246</b>
More tag helpers for generating URLs for links .....	246
How to format numbers in a view .....	248
<b>How to work with a model .....</b>	<b>250</b>
How to pass a model to a view .....	250
How to display model properties in a view .....	252
How to bind model properties to HTML elements .....	254
How to bind a list of items to a <select> element .....	256
How to display a list of model objects .....	258
<b>How to work with Razor layouts .....</b>	<b>262</b>
How to create and apply a layout .....	262
How to nest layouts .....	264
How to use view context .....	268
How to use sections .....	270
<b>The Guitar Shop website .....</b>	<b>272</b>
The user interface for customers .....	272
The user interface for administrators .....	272
<b>Perspective .....</b>	<b>276</b>

## How to use Razor syntax

---

This chapter begins by reviewing the essential skills for using Razor syntax to mix C# code with HTML. This allows you to use most types of C# statements and expressions with HTML.

### How to work with code blocks and inline expressions

---

Figure 7-1 begins by showing the syntax for a Razor *code block* that contains one or more C# statements and for an *inline expression*. After the syntax, the third example shows a controller action method that sets up the data for the view in the fourth example by storing a value of “John” in the CustomerName property of the ViewBag property.

The fourth example begins with a Razor code block. Within this block, the first statement declares a string variable named message and sets it equal to a value of “Welcome!”. Then, this block uses a C# if statement to check whether the CustomerName property of the ViewBag has been set. If so, it sets the message variable to a value of “Welcome back!”.

After the code block, the rest of the example mostly contains the static HTML for a web page. However, within this HTML, three inline expressions insert dynamic data. The first inline expression displays the value of the message variable created by the code block at the top of the example. The second inline expression displays the value stored in the CustomerName property of the ViewBag. And the third inline expression displays the result of adding 2 plus 2.

Of these three inline expressions, parentheses aren’t necessary for the first two. That’s because the first expression just displays the value of a variable, and the second expression just displays a value stored in a property of the ViewBag. However, the third expression requires parentheses because it displays the result of an arithmetic expression.

In general, it’s considered a best practice to perform arithmetic calculations like this one in the controller or model and pass the result of the calculation to the view. Then, the view can use an inline expression to display the result of the calculation. However, if necessary, you can use an inline expression to perform a calculation or to do other types of processing.

## The syntax for a Razor code block

```
@{  
    // one or more C# statements  
}
```

## The syntax for an inline expression

```
@(csharp_expression)
```

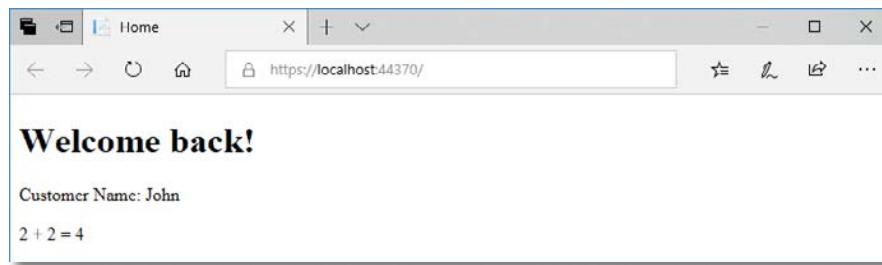
## The Index() action method of the Home controller

```
public IActionResult Index()  
{  
    ViewBag.CustomerName = "John";  
    return View();           // returns Views/Home/Index.cshtml  
}
```

## The Views/Home/Index.cshtml file

```
@{  
    string message = "Welcome!";  
    if (ViewBag.CustomerName != null)  
    {  
        message = "Welcome back!";  
    }  
}  
<!DOCTYPE html>  
<html>  
<head>  
    <meta name="viewport" content="width=device-width" />  
    <title>Home</title>  
</head>  
<body>  
    <h1>@message</h1>  
    <p>Customer Name: @ViewBag.CustomerName</p>  
    <p>2 + 2 = @(2 + 2)</p>  
</body>  
</html>
```

## The view displayed in a browser



## Description

- To execute one or more C# statements, you can declare a *Razor code block* by coding the @ sign followed by a pair of curly braces ({ }). Within the curly braces, you can code one or more C# statements.
- To evaluate a C# expression and display its result, you can code an *inline expression* by coding the @ sign and then coding the expression within a pair of parentheses. For some expressions, the parentheses are optional.

Figure 7-1 How to work with code blocks and inline expressions

## How to code inline loops

---

Figure 7-2 shows how to code an *inline loop* within a view. Within these loops, you can use HTML tags to send HTML to the view.

In the first example, for instance, an inline loop displays the numbers 1 through 12 as the options that are available from a drop-down list. To do that, a for loop declares a month variable that begins at 1, ends at 12, and is incremented by 1. Within the loop, an HTML `<option>` element uses inline expressions to set its value and content to the current value of the month variable. This displays a drop-down list that lets the user select a number for the month.

The second example sets up the data that's used by the third example. To do that, it creates a list of musical instrument categories and stores that list in the `Categories` property of the `ViewBag`.

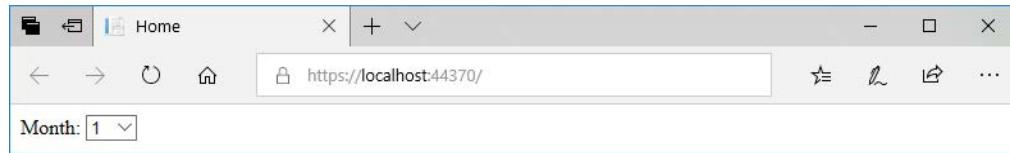
In the third example, an inline loop displays one link for each category stored in the `ViewBag`. To do that, a foreach loop declares a `category` variable that's set to each of the categories stored in the `Categories` property of the `ViewBag`. Within the loop, an HTML `<div>` element contains an `<a>` element that uses inline expressions to set the URL and content for the link to the value of the `category` variable. This displays a list of categories as links.

To keep these examples simple, this figure doesn't use Bootstrap CSS classes or tag helpers. However, you would typically use Bootstrap classes to format a drop-down list like the one shown in the first example. In addition, you would typically use tag helpers to specify the URL for the link in the third example. Later in this chapter, the examples use Bootstrap CSS classes and tag helpers to improve the formatting and function of the HTML elements.

### A for loop that displays a drop-down list of month numbers

```
<label for="month">Month:</label>
<select name="month" id="month">
    @for (int month = 1; month <= 12; month++)
    {
        <option value="@month">@month</option>
    }
</select>
```

### The result displayed in a browser



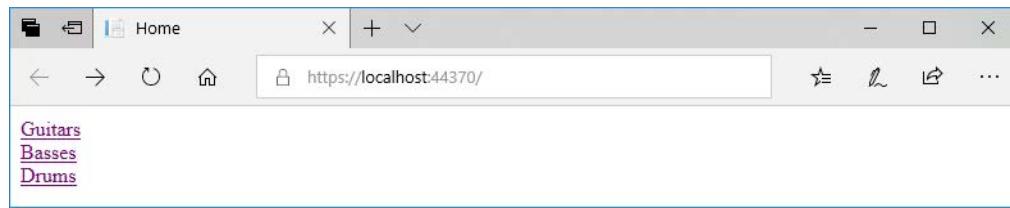
### Code in a controller that creates a list of strings

```
public IActionResult Index()
{
    ViewBag.Categories = new List<string>
    {
        "Guitars", "Basses", "Drums"
    };
    return View();
}
```

### A foreach loop that displays a list of links

```
@foreach (string category in ViewBag.Categories)
{
    <div>
        <a href="/Product>List/@category/">@category</a>
    </div>
}
```

### The result displayed in a browser



### Description

- You can code *inline loops* within a view. Within these loops, you can use HTML tags to send HTML to the view.

---

Figure 7-2 How to code inline loops

## How to code inline conditional statements

---

Figure 7-3 shows how to code *inline conditional statements* such as if/else and switch statements within a view. Like inline loops, these statements can use HTML tags to send HTML to the view.

The first example uses a C# if-else statement to check the ProductID property of the ViewBag. Then, it displays the <p> tag that matches the value stored in this property. This should give you a good idea of how you can use if-else statements within a view to display or not display content.

The second example works like the first example. However, it uses a C# switch statement instead of an if-else statement.

The third example uses a C# if statement to check whether the SelectedCategoryName property of the ViewBag is equal to a string of “All”. If so, it adds a Bootstrap CSS class named active to the class attribute of the <a> element. Since this <a> element is an item in a Bootstrap CSS list group, this marks the element as the active item. To make this work, the example uses the <text> tag to send the active class as plain text.

The fourth example converts the simple if statement in the third example so it uses an *inline conditional expression* instead of a statement. This has the benefit of making the code shorter. To do that, this code uses C#’s conditional operator (? :) to separate three arguments. The first argument checks the same condition as the previous example, the second argument specifies the value to return if the condition is true, and the third value specifies the value to return if the condition is false. As a result, if the condition is true, this expression returns a string of “active”. Otherwise, it returns an empty string. Like the arithmetic expression from figure 7-1, this inline expression requires parentheses.

Although the fourth example uses an inline conditional expression to send plain text to the view, it could also send HTML tags. For example, the true value could be one <div> element and the false value could be another <div> element. However, it’s more common to use conditional expressions to set attribute values.

### An if-else statement in a view

```
@if (ViewBag.ProductID == 1)
{
    <p>Fender Stratocaster</p>
}
else if (ViewBag.ProductID == 2)
{
    <p>Gibson Les Paul</p>
}
else
{
    <p>Product Not Found</p>
}
```

### A switch statement in a view

```
@switch (ViewBag.ProductID)
{
    case 1:
        <p>Fender Stratocaster</p>
        break;
    case 2:
        <p>Gibson Les Paul</p>
        break;
    default:
        <p>Product Not Found</p>
        break;
}
```

### An if statement that adds a Bootstrap CSS class if true

```
<a asp-controller="Product" asp-action="List" asp-route-id="All"
    class="list-group-item
@if (ViewBag.SelectedCategoryName == "All") {
    <text>active</text>
}>
    All
</a>
```

### A conditional expression that adds a Bootstrap CSS class if true

```
<a asp-controller="Product" asp-action="List" asp-route-id="All"
    class="list-group-item
@(ViewBag.SelectedCategoryName == "All" ? "active" : "")>
    All
</a>
```

### Description

- You can code *inline conditional statements* such as if-else and switch statements within a view. These statements can use HTML tags to send HTML to the view.
- To send plain text to a view, you can use the `<text>` tag. This is useful for sending part of an HTML tag such as an HTML attribute or its value.
- You can code an *inline conditional expression* by using C#'s conditional operator (`? :`) to send HTML tags or plain text to the view. This works like other complex inline expressions that require parentheses.

---

Figure 7-3 How to work with inline conditional statements and expressions

## Essential skills for Razor views

---

Now that you understand the basics for using Razor syntax, you're ready to learn some essential skills for working with views. To start, you're ready to learn how to set up the starting folders and files for an app.

### The starting folders and files for an app

---

Figure 7-4 shows the starting folders and files for a website named Guitar Shop that sells musical instruments. These folders and files follow the ASP.NET conventions for storing the models, views, controllers, and static files for an app. By now, you should be familiar with this structure, but let's review it with a focus on how it impacts views.

To start, this folder and file structure maps the actions of each controller to the appropriate view. For example, the `List()` action method of the `Product` controller maps to the `Views/Product/List.cshtml` file. Similarly, the `About()` action method of the `Home` controller maps to the `Views/Home/About.cshtml` file.

Within the `Views` folder, the `Shared` folder stores any layouts that can be shared by views. In this figure, there is only one layout named `_Layout.cshtml`. However, you can add additional layouts if necessary, as described later in this chapter.

The `Views` folder also stores the `_ViewImports.cshtml` and `_ViewStart.cshtml` files. These files make it easier to work with views by importing models and tag helpers and by specifying the default layout.

The `wwwroot` folder stores the CSS and JavaScript files that can be used by the views. In this figure, the `lib` folder contains the CSS file for Bootstrap and the `css` folder contains a custom CSS file that can be used to provide CSS formatting that overrides the Bootstrap CSS whenever that's necessary.

The root folder for the app stores the `Startup.cs` file. This file contains code that configures the middleware for the app, including the routing that specifies how controllers and their action methods are mapped to URLs. In this figure, the controller uses the default routing to map the first segment of the URL to a controller, the second segment to the controller's action method, and the third segment to an optional argument named `id`.

The root folder also stores the `Program.cs` file. This file contains code that sets up the app itself. That includes code that defines the `Startup` class in the `Startup.cs` file as the class that configures the middleware.

## The starting folders and files for the Guitar Shop app

```
GuitarShop
  /Controllers
    /HomeController.cs
    /ProductController.cs
  /Models
    /Category.cs
    /Product.cs
  /Views
    /Home
      /Index.cshtml      -- the view for the Home/Index action
      /About.cshtml      -- the view for the Home/About action
    /Product
      /List.cshtml       -- the view for the Product/List action
      /Details.cshtml   -- the view for the Product/Details action
      /Update.cshtml    -- the view for the Product/Update action
    /Shared
      /_Layout.cshtml   -- a layout that can be shared by views
      _ViewImports.cshtml -- imports models and tag helpers for views
      _ViewStart.cshtml  -- specifies the default layout for views
  /wwwroot
    /css
      /custom.css
    /lib
      /bootstrap/css/bootstrap.min.css
Startup.cs          -- configures middleware that may impact views
Program.cs          -- sets up the app
```

## The routing that's specified in the Startup.cs file

```
app.UseRouting();
app.UseEndpoints(endpoints =>
{
  endpoints.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
});
```

## Description

- By convention, you store the views for an app in a series of folders and files whose names correspond to the controllers and action methods that return the views.
- The Startup.cs file typically contains code that configures the middleware for the app including the routing that specifies how controllers and their action methods are mapped to URLs.
- The Program.cs file sets up the app, including defining the Startup class.

---

Figure 7-4 The starting folders and files for an app

## How to code controllers that return views

---

Figure 7-5 shows how to code controllers that return views. To do that, the action methods of the controllers typically use the `View()` method to create `ViewResult` objects. Then, they return this object.

By default, the `View()` method creates a `ViewResult` object from the view file that corresponds to the controller and action method. In this figure, for example, the `Index()` action method of the `Home` controller uses the `View()` method to create a `ViewResult` object from the view file that's in the `Home/Index.cshtml` file.

However, if you don't want to use this default behavior, you can specify the name of the view in the `View()` method. In this figure, for example, the `Index()` action method of the `Product` controller uses the `View()` method to create a `ViewResult` object from the `Product/List.cshtml` view file. This is the same view file that's used by the `List()` action method of the `Product` controller. In other words, this provides a way for two actions to use the same view file, which is sometimes useful.

In this figure, the `List()` and `Details()` action methods of the `Product` controller store the `id` argument in the `ViewBag` property so it can be accessed by the view. To do that, you can just use Razor syntax as described earlier in this chapter.

All the action methods in this figure specify a return type of the `IActionResult` interface even though these action methods all return `ViewResult` objects. This is possible because the `ViewResult` object described in this figure implements the `IActionResult` interface. Similarly, the `ContentResult` object described in the previous chapter also implements this interface. This shows that coding the `IActionResult` interface as the return type makes your action methods flexible by allowing them to return different object types.

## A method that a controller can use to return a view result to the browser

Method	Description
<code>View()</code>	Creates a ViewResult object that corresponds to the name of the current controller and action method.
<code>View(name)</code>	Creates a ViewResult object that corresponds to the current controller and the specified view name.

### The Home controller

```
public class HomeController : Controller
{
    public IActionResult Index()
    {
        return View();           // Views/Home/Index.cshtml
    }
}
```

### The Product controller

```
public class ProductController : Controller
{
    public IActionResult Index()
    {
        return View("List");    // Views/Product/List.cshtml
    }

    public IActionResult List(string id = "All")
    {
        ViewBag.Category = id;
        return View();          // Views/Product/List.cshtml
    }

    public IActionResult Details(string id)
    {
        ViewBag.ProductSlug = id;
        return View();          // Views/Product/Details.cshtml
    }
}
```

### Description

- A controller typically contains action methods that are mapped to the view files in the Views folder.

---

Figure 7-5 How to code controllers that return views

## How to create a default layout and enable tag helpers

---

Figure 7-6 shows the code that's necessary to create a default layout and enable tag helpers. This makes it possible to share code between multiple view files.

To start, you can add a Razor layout to your project as described in chapter 2. A Razor layout typically specifies code that's shared between multiple views such as the `<html>`, `<head>`, and `<body>` elements for a view. In addition, it often imports the CSS and JavaScript files for a view. In this figure, for example, the Razor layout named `_Layout` specifies all of these HTML elements and imports the CSS files for the Bootstrap library as well as a custom CSS file.

Within a Razor layout, the `ViewBag` is often used to display a title that's set in the view. In this figure, for example, the `<title>` element uses an inline Razor expression to display the `Title` property of the `ViewBag`. So far, this book has only shown how to use the `ViewBag` property. However, the next chapter describes the  `ViewData` property, which works similarly, and is also often used to display similar types of data.

Within a Razor layout, the `RenderBody()` method renders the body of the view. In other words, this method inserts the HTML elements and Razor code defined by a view into the layout.

In a typical web app, most of the views share the same layout. As a result, it usually makes sense to specify a default layout for all views. To do that, you can add a `_ViewStart` file to the project and set the `Layout` property to the name of the Razor layout. In this figure, for example, the `_ViewStart` file sets the default layout to the layout named `_Layout`. That way, when you code a view, you only needs to specify a layout if you want to override this default as shown later in this chapter.

When you work with views, it's common to use the tag helpers that are available from ASP.NET Core MVC in most views. As a result, it typically makes sense to enable MVC tag helpers for all views. The easiest way to do that is to add a `_ViewImports` file to your project like the one shown in this figure.

## The \_Layout.cshtml file in the Views/Shared folder

```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link rel="stylesheet" href="~/lib/bootstrap/css/bootstrap.min.css" />
    <link rel="stylesheet" href="~/css/custom.css" />
</head>
<body>
    @RenderBody()
</body>
</html>
```

## A \_ViewStart.cshtml file that sets the default layout

```
@{
    Layout = "_Layout";
}
```

## A \_ViewImports.cshtml file that enables all ASP.NET Core MVC tag helpers

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

## How to add a Razor layout, view start, or view imports file

1. Right-click on the folder where you want to add the file, and select the Add→New Item item.
2. In the resulting dialog, select the ASP.NET Core→Web category.
3. Select the Razor item you want to add and respond to the resulting dialog boxes.

## Description

- To specify code that's shared between multiple view files, add a Razor layout to the Views/Shared folder of your project.
- Within a Razor layout, the ViewBag or ViewData property is often used to display a title that's set in the view.
- Within a Razor layout, the RenderBody() method renders the body of the view.
- To specify a default layout for all views, add a \_ViewStart file to the Views folder of your project and set the Layout property to the name of the Razor layout.
- To enable all ASP.NET Core MVC tag helpers for all views, add a \_ViewImports file to the Views folder of your project that contains the code shown above.

---

Figure 7-6 How to create a default layout and enable tag helpers

## How to use tag helpers to generate URLs for links

---

When you code an `<a>` element for a link, you can use the `href` attribute to hard code its URL. However, in general, it's considered a best practice to use tag helpers to generate the URL for a link.

In figure 7-7, the first example shows two ways to code a link to the same URL. Here, the first approach uses the `href` attribute and a hard-coded string value, and the second approach uses tag helpers to specify the controller, action method, and `id` argument for the route.

Between these approaches, the first approach is short and easy to read, but the second approach is recommended because it is more flexible. For example, if you change the routing for the app so the `List()` action method of the `Product` controller maps to `/Products` instead of `/Product/List`, the second approach adjusts automatically. The first approach, on the other hand, would break and need to be fixed manually.

The second example shows that you don't need to code the `asp-controller` tag helper when you're coding a link to another action method in the same controller. Here, the example assumes that the link is coded in the `Index` view that's displayed by the `Home` controller. As a result, it displays the `About` view of the `Home` controller. Of course, if you want to display a view that's in another controller, you can just add the `asp-controller` tag helper to the link as shown in the third example.

The fourth example shows what happens if you use the `asp-route` tag helper to specify a value for a parameter name that exists in one of the app's routes. Here, the `asp-route` tag helper specifies a value for the `id` parameter that's the third segment of the default route. As a result, the app uses the parameter value as the third segment of the generated URL.

The fifth example shows what happens if you use the `asp-route` tag helper to specify a value for a parameter name that doesn't exist in the app's route. Here, the `asp-route` tag helper specifies values for the parameters named `page` and `sort_by`. However, these parameters don't exist in the default route. As a result, the app adds the parameter names and values to the end of the URL as part of the URL's query string.

As you learned in the last chapter, using query strings in a URL makes the URL difficult to read. As a result, it's considered a best practice to avoid using them whenever possible. To do that for this example, you could add a route to the app for the `List()` action method of the `Product` controller that includes the `page` and `sort_by` parameters.

### Three tag helpers that you can use to generate URLs

Tag helper	Description
<code>asp-controller</code>	Specifies the controller. This is only necessary if you want to specify a URL for an action method from another controller.
<code>asp-action</code>	Specifies the action method.
<code>asp-route-param_name</code>	Specifies a route parameter where param_name is the name of the parameter. If you specify a parameter name that exists in one of the app's routes, the app uses the parameter value as a segment of the URL. Otherwise, it adds the parameter name and value to the end of the URL as part of the URL's query string.

### Two ways to code a link

#### Use HTML to hard code the URL in the href attribute

```
<a href="/Product>List/Guitars">View guitars</a>
```

#### Use ASP.NET tag helpers to generate the URL

```
<a asp-controller="Product" asp-action="List"
asp-route-id="Guitars">View guitars</a>
```

#### The URL for both links

/Product>List/Guitars

### How to code a link to an action method in the same controller

```
<a asp-action="About">About Us</a>
```

#### The URL that's generated

/Home/About

### How to code a link to an action method in a different controller

```
<a asp-controller="Product" asp-action="List">View all products</a>
```

#### The URL that's generated

/Product/List

### How to code a link that includes a parameter that's in a route

```
<a asp-controller="Product" asp-action="List"
asp-route-id="Guitars">View guitars</a>
```

#### The URL that's generated

/Product>List/Guitars

### A link that specifies a route parameter that doesn't exist

```
<a asp-controller="Product" asp-action="List"
asp-route-page="1" asp-route-sort_by="price">Products - Page 1</a>
```

#### The URL that's generated

/Product/List?page=1&sort\_by=price

### Description

- In general, it's considered a best practice to use tag helpers to generate the URL for a link.

---

Figure 7-7 How to use tag helpers to generate URLs for links

## Three views that use the default layout and tag helpers

---

To put the previous figures into a larger context, figure 7-8 shows the code for three views that use the default layout. Here, the Razor code block at the top of each view stores a title for the page in the ViewBag property. That way, the layout for the view can display the title in the <title> element.

However, this Razor code block doesn't set the Layout property to override the default layout. As a result, these views use the default layout file (\_Layout) that's specified by the \_ViewStart file. Both of these files are shown earlier in this chapter.

After the Razor code block, the Home/Index view specifies the level-1 heading for the page followed by a <div> element that uses a Bootstrap class to identify its content as a list group. This is possible because the layout for the view includes a link to the library for the Bootstrap CSS classes.

Within this <div> element, there are three links. Here, the first link uses tag helpers to specify the Product controller and the List() action method. The second link uses tag helpers to specify the Product controller, the List() action method, and a route id parameter of "Guitars". And the third link uses tag helpers to specify the Product controller, the Details() action method, and a route id parameter of "Fender-Stratocaster".

The Product/List view works much like the Home/Index view. However, within the <div> element, it displays the Category property of the ViewBag that was set by the List() action method. In addition, this view's first link doesn't use a tag helper to specify a controller since that link is to an action that's also in the Product controller.

The Product/Details view works much like the Product/List view. However, within the <div> element, it displays the ProductSlug property of the ViewBag that was set by the Details() action method.

## The Home/Index view

```
@{
    ViewBag.Title = "Home";
}
<h1>Home</h1>
<div class="list-group">
    <a asp-controller="Product" asp-action="List">View all products</a>

    <a asp-controller="Product" asp-action="List"
        asp-route-id="Guitars">View guitars</a>

    <a asp-controller="Product" asp-action="Details"
        asp-route-id="Fender-Stratocaster">View Fender Stratocaster</a>
</div>
```

## The Product/List view

```
@{
    ViewBag.Title = "Product List";
}
<h1>Product List</h1>
<div class="list-group">
    <p>Category: @ViewBag.Category</p>

    <a asp-action="Details"
        asp-route-id="Fender-Stratocaster">View Fender Stratocaster</a>

    <a asp-controller="Home" asp-action="Index">Home</a>
</div>
```

## The Product/Details view

```
@{
    ViewBag.Title = "Product Details";
}
<h1>Product Details</h1>
<div class="list-group">
    <p>Slug: @ViewBag.ProductSlug</p>

    <a asp-controller="Home" asp-action="Index">Home</a>
</div>
```

## Description

- Since the Razor code block at the top of the page doesn't set the Layout property, these views use the layout (\_Layout) that's specified by the \_ViewStart file.
- These views use tag helpers to specify the controller class, action method, and route-id parameter for <a> elements.

---

Figure 7-8 Three views that use the default layout and tag helpers

## The three views displayed in a browser

---

To make it easier to visualize the three views presented in the previous figure, figure 7-9 shows them after they are displayed in a browser. Here, the tab at the top of the browser window displays the title for the page. The address bar shows the URL for each page. And the rest of the browser displays the web page that's generated by the Razor view and its layout.

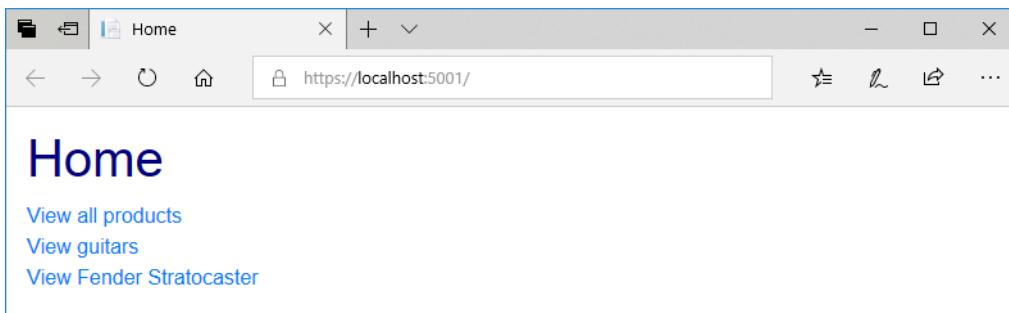
The Home page displays a level-1 heading for the page followed by three links. Here, the first two links lead to the Product List page and the third link leads to the Product Details page.

The Product List page works much like the Home page. However, after the heading, it displays the category that's set by the List() action method of the Product controller. In this figure, it displays a category of "Guitars" since the user clicked on the "View guitars" link. However, if the user clicks on the "View all products" link, it would display a category of "All".

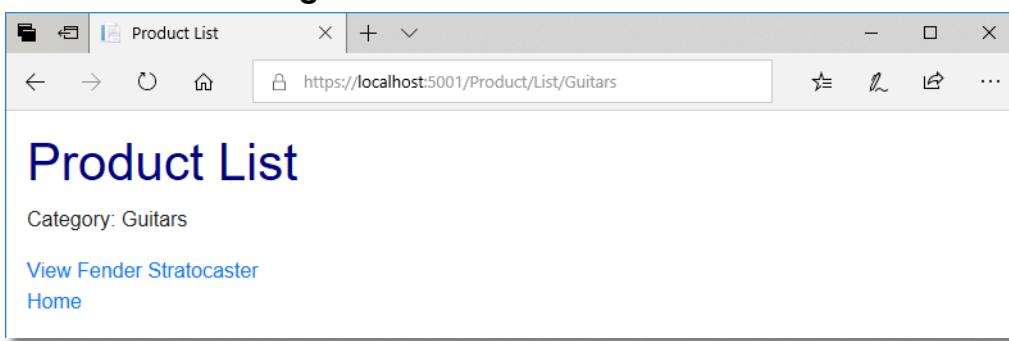
The Product Details page works like the Product List page. However, after the heading, it displays the product slug that's set by the Details() action method of the Product controller. In this figure, it displays a slug of "Fender-Stratocaster" since the user clicked on the "View Fender Stratocaster" link.

Although these pages might not seem useful, they show you the basics of how you can use controllers, views, and layouts to create pages. Later in this chapter, you'll learn how to display a list of products that matches the category. Similarly, you'll learn how to display the details of a product that matches the slug. That allows you to use web pages to display useful information to your users.

### A browser displaying the Home page



### A browser after clicking the View Guitars link



### A browser after requesting the View Fender Stratocaster link

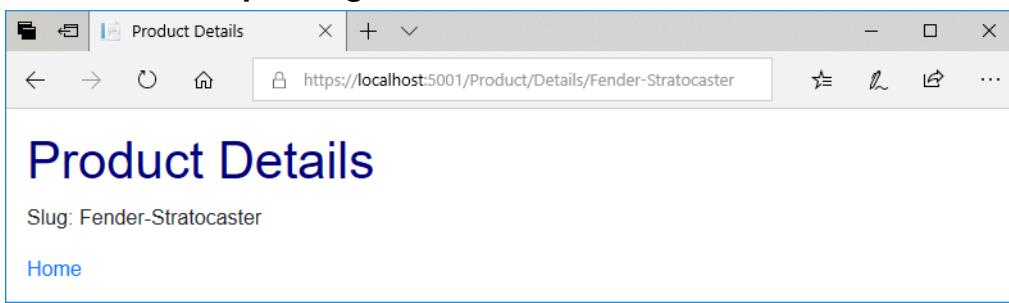


Figure 7-9 The three views displayed in a browser

## More skills for Razor views

Now that you know some essential skills for working with Razor views, you're ready to learn some more skills for working with views.

### More tag helpers for generating URLs for links

Earlier in this chapter, you learned how to use three tag helpers to generate URLs for links. When coding links, these are the tag helpers that you'll use most of the time. However, in some cases, you may need to use the tag helpers described in figure 7-10 to generate URLs for links.

To start, if you organize your app into multiple routing areas as described in chapter 6, you may need to use the `asp-area` tag as shown in the first example in this figure. Here, the `asp-area` tag helper specifies an area of "Admin". This accesses an area that allows admin users to manage the list of products that's available from the website. This area is different from the area that allows regular end users such as customers to view products and add them to their carts.

Second, if you have a long page, you may want to use a *URL fragment* to allow the user to jump to a specified placeholder on the page as shown by the second example. Here, the `<h2>` element includes an `id` attribute that marks a placeholder named "Fender" on the page. Then, the link uses the `asp-fragment` tag helper to jump to that placeholder. If the user clicks on this link, it causes the browser to scroll up or down until it displays the part of the page that contains the placeholder. In a URL, a fragment is preceded by the hash mark (#) as shown by both the second and third examples.

When coding a link for a website, it's common to use a *relative URL*, which is a URL that's relative to the app's root directory. So far, this book has only shown how to code relative URLs. However, if you need to code a link to a different domain, you may need to code an *absolute URL* that specifies the name of the host. In the third example, for instance, the link uses the `asp-protocol` and `asp-host` tag helpers to specify the HTTP protocol and host for the link. As a result, no matter what host your app is running on, this link jumps to the `https://murach.com` domain.

Relative URLs are more flexible than absolute URLs because they allow your app to work on multiple domains. For example, if you use relative URLs, your web app runs equally well on `localhost` or `murach.com`. That way, you can develop and test the app on `localhost` and deploy it to `murach.com` when it's ready for production. However, if you use absolute URLs, your web app only runs on the specified host. As a result, you should only use an absolute URL when you want to code a link to a domain outside your web app.

In addition to specifying the protocol and host for the link, the third example specifies a controller, action method, route parameter, and fragment. This shows that you can use tag helpers to generate all parts of a URL.

## More tag helpers for generating URLs

Tag helper	Description
<code>asp-area</code>	Specifies the area for the URL. For info about setting up areas, see chapter 6.
<code>asp-fragment</code>	Specifies the placeholder that you want to jump to.
<code>asp-protocol</code>	Specifies the protocol. By default, it is set to HTTP. However, it's common for apps to automatically redirect to HTTPS, even if you specify HTTP.
<code>asp-host</code>	Specifies the name of the server.

### How to code a link to an area

```
<a asp-area="Admin" asp-controller="Product"  

asp-action="List">Admin - Product Manager</a>
```

#### The URL that's generated

/Admin/Product/List

### How to work with placeholders

#### How to code an HTML placeholder

```
<h2 id="Fender"

```

#### How to code a URL that jumps to an HTML placeholder on the same page

```
<a asp-fragment="Fender"

```

#### The URL that's generated

/#Fender

### How to code an absolute URL

```
<a asp-protocol="https"  

asp-host="murach.com"  

asp-controller="Shop"  

asp-action="Details"  

asp-route-id="html5-and-css3"  

asp-fragment="reviews">Murach's HTML5 and CSS3 - Reviews</a>
```

#### The URL that's generated

<https://www.murach.com/Shop/Details/html5-and-css3#reviews>

### Description

- A *URL fragment* allows you to jump to a specified placeholder on a web page. In a URL, a fragment is preceded by the hash mark (#).
- A *relative URL* is a URL that's relative to the app's root directory.
- An *absolute URL* is a URL that specifies the host.
- Relative URLs are more flexible than absolute URLs because they allow your app to work on multiple hosts. As a result, you should only use an absolute URL when coding a link to a specific host outside your app.

---

Figure 7-10 More skills for using tag helpers to generate URLs for links

## How to format numbers in a view

---

In general, it's considered a best practice for views to be responsible for formatting numbers so they can be displayed to the user. In figure 7-11, for instance, the examples show how to apply currency, number, and percent formatting.

In a Razor expression, you can use C#'s format specifiers to specify a format and number of decimal places for a number. By default, these specifiers return two decimal places, but you can also specify the number of decimal places. In other words, both C and C2 return two decimal places. Some programmers prefer to use C2 because it's clear that it returns two decimal places. Others prefer to use C because it's shorter.

If necessary, the format specifiers round the number to the nearest decimal place. In most cases, that's what you want. If it isn't, you can apply rounding before you display the view. For example, you can apply rounding in the model.

This figure presents three of the most common format specifiers, but many others exist. So, if you need to apply formatting that goes beyond the formatting specified in this figure, you can search the Internet for more information.

In general, it's considered a best practice for a controller or model to be responsible for providing the data for a view. In other words, a view shouldn't retrieve data from a database because that's the job of the controllers. Similarly, a view shouldn't calculate a value to be displayed because that should be the job of a model object. However, once the model object calculates the value, the view can format that value because formatting and displaying data is the job of the view.

## Format specifiers you can use to format numbers

Specifier	Name	Description
C	Currency	Formats a number as a currency value for the current locale.
N	Number	Formats a number using a separator for the thousandths places.
P	Percent	Formats a number as a percentage.

## Code in a controller that stores numbers in the ViewBag property

```
ViewBag.Price = 12345.67;
ViewBag.DiscountPercent = .045;
ViewBag.Quantity = 1234;
```

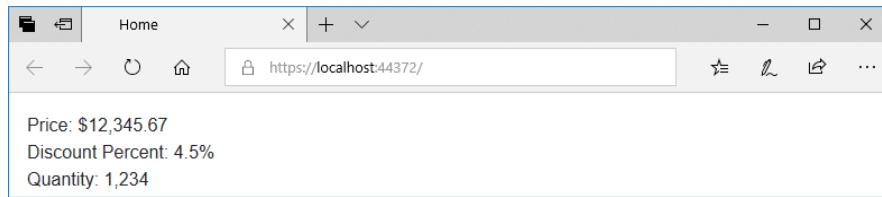
## Razor expressions that format and display the numbers

Expression	Result
@ViewBag.Price.ToString("C")	\$12,345.67
@ViewBag.Price.ToString("C1")	\$12,345.7
@ViewBag.Price.ToString("C0")	\$12,346
@ViewBag.Price.ToString("N")	12,345.67
@ViewBag.Price.ToString("N1")	12,345.7
@ViewBag.Price.ToString("N0")	12,346
@ViewBag.DiscountPercent.ToString("P")	4.50%
@ViewBag.DiscountPercent.ToString("P1")	4.5%
@ViewBag.DiscountPercent.ToString("P0")	5%

## Code in a view that displays the numbers

```
<div>Price: @ViewBag.Price.ToString("C")</div>
<div>Discount Percent: @ViewBag.DiscountPercent.ToString("P1")</div>
<div>Quantity: @ViewBag.Quantity.ToString("N0")</div>
```

## The view displayed in a browser



## Description

- In a Razor expression, you can use C#'s format specifiers to format a number as currency or a percent. One way to do that is to call the `ToString()` method from the numeric data type and pass it the format specifier.
- By default, the format specifiers return two decimal places, but you can specify the number of decimal places.
- If necessary, the format specifiers round numbers to the nearest decimal place.
- This figure presents three of the most common format specifiers, but many others exist. For more information, you can search the Internet.

Figure 7-11 How to format numbers in a view

## How to work with a model

So far, this chapter has shown how to work with views that aren't bound to a model. However, when using the MVC pattern, the controller typically stores the data for the view in a model and binds that model to the view. This makes it easier to pass the data back and forth between the controller and the view.

### How to pass a model to a view

Figure 7-12 begins by showing a C# class that defines a *Product* object that can be used as a *model*. This class defines four properties to store the ID, category, name, and price for the product. In addition, it defines a read-only property that creates a slug for the product by replacing any spaces in the product's name with dashes. This property is read-only because it uses a lambda expression to specify a return value but doesn't specify a way to set that value.

This figure continues by showing a class that defines a *Category* object that can be used as a model. In this figure, you can use the *Category* property of the *Product* object to store the model for a category within a product. In this way, you can define a model of the data that's available to a view.

After you define the model, you can create an object from the model and fill that model with data. To do that, you often begin by reading the data for the model from the database. In the third example, for instance, the *Product/Details()* action method begins by reading a *Product* object that corresponds to the *id* parameter from the database. To do that, this example uses the *context* variable to access the *DbContext* object that provides the data for the app. This uses EF and LINQ to get the *Product* object that corresponds to the *id* argument as described in chapter 4.

After you fill the model with data, you can pass it to the view. To do that, you can use the *View()* method that's available to all controllers as shown in the third example. This creates a *ViewResult* object that can be returned by the action method.

Most of the time, you can use the first overload of the *View()* method shown in this figure to pass the model to the view. This passes the model to the view that corresponds to the name of the action method. However, in some cases, you may want to use the second overload of the *View()* method to pass the model to the view specified by the first argument. For instance, you may want the *Product/Add()* action method to pass a new *Product* model to the *Update* view as shown in the fourth example. To do that, this code specifies the name of the *Update* view as the first argument of the *View()* method. That way, the *Update* view can display the empty fields that need to be entered for a new product. Similarly, you may want the *Product/Update()* action method to pass a *Product* model to the *Update* view. That way, the *Update* view can display the data for an existing product that can be updated.

## The Product model

```
namespace GuitarShop.Models
{
    public class Product
    {
        public int ProductID { get; set; }
        public Category Category { get; set; }
        public string Name { get; set; }
        public decimal Price { get; set; }
        public string Slug => Name.Replace(' ', '-');
    }
}
```

## The Category model

```
namespace GuitarShop.Models
{
    public class Category
    {
        public int CategoryID { get; set; }
        public string Name { get; set; }
    }
}
```

## A method that a controller can use to pass a model to a view

Method	Description
<code>View(model)</code>	Passes the specified model to the corresponding view and creates a ViewResult object.
<code>View(name, model)</code>	Passes the specified model to the view with the specified name and creates a ViewResult object.

## The Product/Details() action method that passes a model to a view

```
public IActionResult Details(int id)
{
    Product product = context.Products.Find(id);
    return View(product);
}
```

## A Product/Add() action method that passes a model to the specified view

```
[HttpGet]
public IActionResult Add()
{
    return View("Update", new Product());
}
```

### Description

- A *model* is a regular C# class that defines an object.
- In a controller, you can use the `View()` method to pass a model object to a view. Then, an action method can return the `ViewResult` object.
- In this figure, the `context` variable is a `DbContext` object that provides access to the data for the app as described in chapter 4.

---

Figure 7-12 How to pass a model to a view

## How to display model properties in a view

---

Figure 7-13 shows how to display model properties in a view. To start, you can add an @using directive to the \_ViewImports file that specifies the namespace for the model classes. That way, it's easy to access these classes in your views as shown in this figure. If you don't add an @using directive, you can still access a model, but you must fully qualify the name of its class.

At the beginning of a view, you can use the @model directive to identify the class for the model. In this figure, for instance, the view begins by identifying the Product class as its model class. This binds the model to the view.

After that, you can use the built-in Model property of the view to access the model object. In this figure, for instance, the first use of @Model displays the ProductID property of the Product object. The second use of @Model displays the Name property of the Product object. The third use of @Model displays the Name property of the Category object that's nested in the Product object. And the fourth use of @Model displays the Price property of the Product object.

Before this example displays the product's price, it uses the ToString() method to convert the price from a number to a string. As it does this, it uses C#'s format specifiers to format it as currency with two decimal places as described earlier in this chapter.

To help you remember the correct capitalization, remember that a property typically begins with an uppercase letter. For example, you use @Model to access the built-in Model property, and you use @ViewBag to access the built-in ViewBag property. On the other hand, a directive like @model or @using typically begins with a lowercase letter.

## A directive and a property you can use to display model properties in a view

Razor syntax	Description
@model	A directive that specifies the data type (class) for the model and binds the model to the view.
@Model	Accesses the built-in Model property that accesses the model object that's bound to the view.

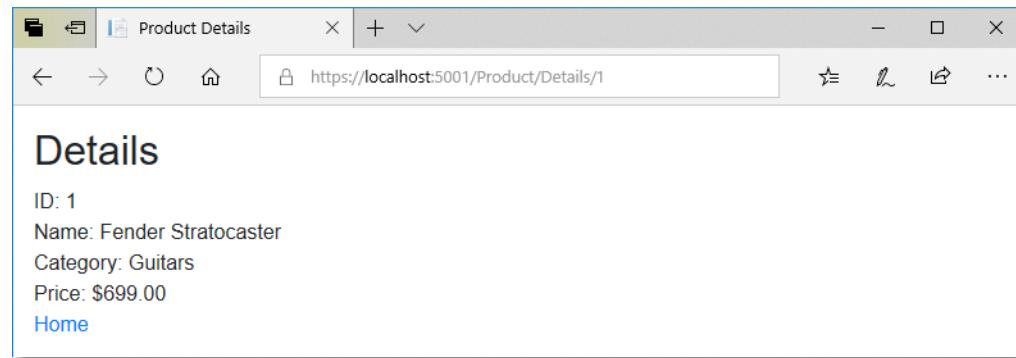
## A \_ViewImports file that imports the Models namespace for all views

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

### The code for the Product/Details view

```
@model Product
 @{
     ViewBag.Title = "Product Details";
 }
 <h1>Product Details</h1>
 <div>ID: @Model.ProductID</div>
 <div>Name: @Model.Name</div>
 <div>Category: @Model.Category.Name</div>
 <div>Price: @Model.Price.ToString("C2")</div>
 <a asp-controller="Home" asp-action="Index">Home</a>
```

### The view displayed in a browser



### Description

- In a view, you can use the @model directive to bind a model to a view. After that, you can use the Model property to access the properties and methods of the model object.

Figure 7-13 How to display model properties in a view

## How to bind model properties to HTML elements

When you code a form within a view, you typically use the `asp-for` tag helper to bind model properties to HTML elements as shown in figure 7-14. This makes it easier to write and maintain the code for the views.

The first example in this figure shows how this works. Here, the view begins by using the `@model` directive to specify that it's using a `Product` model. Then, within the `<form>` element, this view binds the first `<label>` and `<input>` elements to the `Name` property of the model. This displays the name of the product in the first text box. Next, this view binds the second `<label>` and `<input>` elements to the `Price` property of the model. This displays the price of the product in the second text box. Finally, this view binds the third and fourth `<input>` elements to the `ProductID` and `Category.Name` properties of the model. Since these `<input>` elements are hidden, this data isn't displayed to the user. However, it is passed to the controller.

The second example shows the HTML that's generated for the `Price` property of the `Product` model. This shows that the `asp-for` tag helpers generate the HTML attributes necessary for client-side and server-side code to access this element. In other words, it generates the `for`, `id`, and `name` attributes.

The `asp-for` tag helper for the `<label>` element also generates the content for the label if it isn't specified. The content is the name of the bound property, in this case, `Price`. You can also specify the name to be displayed by coding a `[Display]` attribute for the property as shown in chapter 11. To be sure that the name is displayed correctly, most of the examples in this book include the name as the content for the `<label>` element.

The `asp-for` tag helper for the `<input>` element also generates the `value` attribute necessary to display the product's price, as well as the `type` attribute that identifies the type of field to be displayed. If you compare the `<input>` element that uses the tag helper to the `<input>` element that's generated by the tag helper, you can see that the tag helper makes the `<input>` element shorter and it has less code duplication. As a result, it makes your code easier to write and maintain.

Note that the value of the `type` attribute is determined by the data type of the property that the `<input>` element is bound to. In this example, the `Price` property has a data type of `decimal`, so the `type` attribute is set to "text" so a normal text field is displayed. If you want to change the type of field that's displayed, you can code the `type` attribute with the value you want on the `<input>` element. For example, if an `<input>` element is bound to a property with a data type of `DateTime`, the `type` attribute will be set to "datetime-local" and a control that allows the user to enter both a date and a time will be displayed. If you just want the user to enter a date, though, you can set the `type` attribute to "text" to display a normal text box or to "date" to display a control that allows the user to enter just a date.

## A tag helper you can use to bind HTML elements to model properties

Tag helper	Description
asp-for	Specifies the model property for the HTML element.

### The code for a view

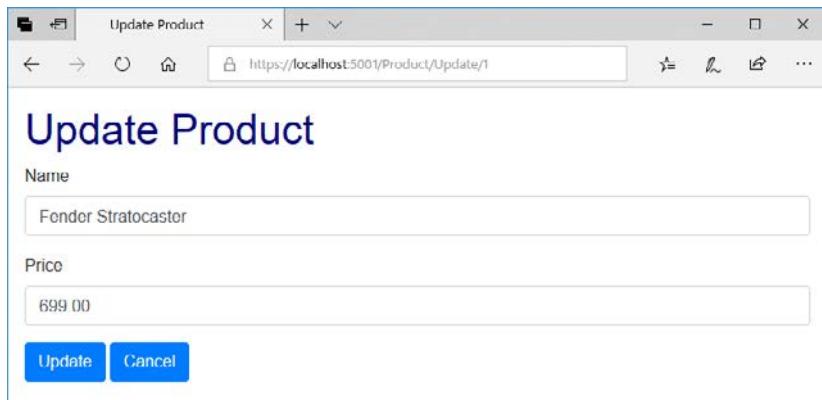
```
@model Product
 @{
     ViewBag.Title = "Update Product";
 }
<h1>Update Product</h1>
<form asp-action="Update" asp-route-id="@Model.Slug" method="post">
    <div class="form-group">
        <label asp-for="Name"></label>
        <input asp-for="Name" class="form-control">
    </div>

    <div class="form-group">
        <label asp-for="Price"></label>
        <input asp-for="Price" class="form-control">
    </div>

    <input type="hidden" asp-for="ProductID" />
    <input type="hidden" asp-for="Category.Name" />

    <button asp-action="Update" type="submit" class="btn btn-primary">
        Update</button>
    <a asp-action="List" class="btn btn-primary">Cancel</a>
</form>
```

### The view displayed in a browser



### The HTML that's generated for the Price <label> and <input> elements

```
<label for="Price">Price</label>
<input class="form-control" type="text"
      id="Price" name="Price" value="699.00">
```

### Description

- In a form, you typically use the asp-for tag helper to bind model properties to HTML elements such as the <label> and <input> elements.

Figure 7-14 How to bind model properties to HTML elements

## How to bind a list of items to a <select> element

If you need to code a drop-down list in a form, you can use the `asp-items` tag helper to bind items to a `<select>` element as shown in figure 7-15. This creates the `<option>` elements for the `<select>` element. This is usually easier and less error-prone than using an inline foreach loop to manually create the `<option>` elements.

The first example in this figure shows an action method named `Update()` that creates the model and passes it to the view. To start, the first statement uses the `DbContext` object to get a list of `Category` objects and stores it in the `ViewBag` property so it's available to the view. Then, the second statement uses the `DbContext` object to find the `Product` object that corresponds to the specified `id` argument. After that, the third statement uses the `View()` method to pass the `Product` object to the view.

The second example shows the code that binds the list of `Category` objects to a `<select>` element. To start, the `<label>` and `<select>` elements use the `asp-for` tag helper to bind the `CategoryID` property of the model to the view as described in the previous figure. Then, this example uses the `asp-items` tag helper to bind the list of `Category` objects to the `<select>` element.

To bind the list of `Category` objects, this code uses a Razor expression to create a new `SelectList` object. To do that, it uses a `SelectList()` constructor that accepts four arguments. The first argument specifies the list of `Category` objects. The second argument specifies the `CategoryID` property as the data value. The third argument specifies the `Name` property as the display text. And the fourth argument specifies the category ID of the current `Product` as the selected value. As a result, when the user displays this view, the drop-down list selects the correct category for the product that's being updated.

When you call the `SelectList()` constructor, the first argument is the only required argument. As a result, for some simple drop-down lists, you might not need to code the other three arguments. In most cases, though, you'll need to code all four arguments.

The third example shows the HTML that's generated for the `<select>` element. This shows that the `asp-items` tag helper generates the `<option>` elements necessary to display the three `Category` objects. Here, the `value` attribute uses the `CategoryID` property to get its value, the `Name` property is used as the display text, and the `selected` attribute is set for the first option. As a result, when the view displays the drop-down list, the `Guitars` category is selected.

## A tag helper you can use to add options to a <select> element

Tag helper	Description
asp-items	Specifies the items for a <select> element.

## The constructor of the SelectList class

Constructor	Description
<code>SelectList(list, value, text, selectedValue)</code>	Specifies the enumerable collection that contains the objects for each item, an optional property name that specifies the data value for the item, an optional property name that specifies the display text for the item, and an optional value that specifies the value of the selected item.

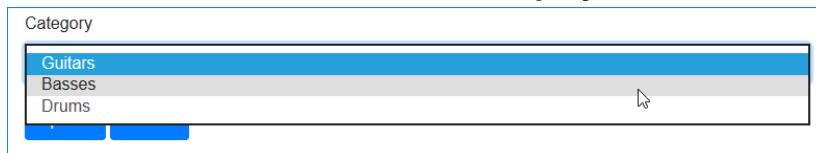
## An action method that creates the model and passes it to the view

```
public IActionResult Update(int id)
{
    ViewBag.Categories = context.Categories.ToList();
    Product product = context.Products.Find(id);
    return View(product);
}
```

## The code that binds items to a <select> element

```
<div class="form-group">
    <label asp-for="CategoryID">Category</label>
    <select asp-for="CategoryID"
        asp-items="@{new SelectList(ViewBag.Categories,
            "CategoryID", "Name",
            Model.CategoryID.ToString())}"
        class="form-control"></select>
</div>
```

## The <select> element and its label displayed in a browser



## The HTML that's generated for the <select> element

```
<select class="form-control" id="Category_CategoryID"
    name="Category.CategoryID">
    <option selected="selected" value="1">Guitars</option>
    <option value="2">Basses</option>
    <option value="3">Drums</option>
</select>
```

## Description

- You can use the asp-items tag helper to bind items to a <select> element. This creates the <option> elements for the <select> element.
- In this figure, the context variable is a DbContext object that provides access to the data for the app as described in chapter 4.

Figure 7-15 How to bind a list of items to a <select> element

## How to display a list of model objects

---

If you need to display a list of model objects, you can display them in a table. For example, figure 7-16 shows how to display a list of Product objects in a table.

The first example in this figure shows an action method named List() that creates the model and passes it to the view. To start, the first statement gets a list of Category objects and stores it in the ViewBag property so this list is available to the view. Then, the second statement stores the id parameter in the ViewBag property so the selected category is available to the view. Next, the third statement uses the id parameter to get the list of Product objects for the selected category. After that, the fourth statement uses the View() method to pass the list of Product objects to the view.

The second example shows the code for a view that displays the list of Product objects in a table. To start, this example uses the @model directive to bind the list of Product objects to the view. Next, this example uses an inline if statement to check whether the Categories property of the ViewBag has been set. If so, it uses an inline foreach statement to display each category as a series of links just below the Product List heading. These links use tag helpers to pass the category to the Product/List() action method as its id parameter. Similarly, the link that follows the foreach loop also uses these tag helpers to pass a category of “All” to the Product/List() action method as its id parameter.

After displaying links below the Product List heading, this code displays the name of the selected category as a heading. To do that, it uses an inline @if statement to check whether the SelectedCategory property of the ViewBag has been set to a value of “All”. If so, it displays the All Products heading. Otherwise, it displays a heading that corresponds to the name of the selected category.

## An action method that creates the model and passes it to the view

```
public IActionResult List(string id = "All")
{
    ViewBag.Categories = context.Categories.ToList();
    ViewBag.SelectedCategory = id;
    List<Product> products = context.Products
        .Where(p => p.Category.Name == id).ToList();
    return View(products);
}
```

## The code for the view

```
@model List<Product>
 @{
    ViewBag.Title = "Product List";
}

<h1>Product List</h1>
@if (ViewBag.Categories != null)
{
    foreach (Category c in ViewBag.Categories)
    {
        <a asp-controller="Product" asp-action="List"
            asp-route-id="@c.Name">@c.Name</a><text> | </text>
    }
}
<a asp-controller="Product" asp-action="List"
    asp-route-id="All">All</a>
<hr />

@if (@ViewBag.SelectedCategory == "All")
{
    <h2>All Products</h2>
}
else
{
    <h2>@ViewBag.SelectedCategory</h2>
}
```

---

Figure 7-16 How to display a list of model objects (part 1)

Part 2 of this figure shows the `<table>` element that displays the list of Product objects. To start, the `<table>` element uses its class attribute to specify some Bootstrap CSS classes to format the table. Then, it uses the `<thead>` element to identify the header row for the table. Within this element, the `<tr>` and `<th>` elements define a header row that contains four columns: a Name heading, a Price heading, and two blank headings.

After the `<thead>` element, the `<tbody>` element identifies the body for the table. Within this element, an inline foreach loop displays all Product objects in the model. In other words, it displays all products in the current category.

Within the foreach loop, the `<tr>` and `<td>` elements define a row that contains four columns. The first column displays the Name property of the Product object. The second column displays the Price property of the Product object after currency formatting has been applied to it. The third column displays a View link that provides a way to view details about the product. And the fourth column displays an Update link that provides a way to update the data for the product.

The third and fourth columns use tag helpers to specify the action method for the link. For example, the third column specifies that clicking the View link calls the `Product/Details()` action method. In addition, both of these columns use the `asp-route-id` tag helper to specify the `id` argument for the action method. In this example, both of these tag helpers specify that the slug for the product should be used as the `id` argument. This results in a user-friendly URL since it displays the product slug as the third segment of the URL.

This figure finishes by showing the view after it has been displayed in a browser. Here, the URL displays the product category as the third segment of the URL. In this case, the view displays all products in the Guitars category. However, if the user clicks on the Basses link at the top of the page, it would display all products in the Basses category. And so on.

### The code for the view (continued)

```
<table class="table table-bordered table-striped">
    <thead>
        <tr>
            <th>Name</th>
            <th>Price</th>
            <th></th>
            <th></th>
        </tr>
    </thead>
    <tbody>
        @foreach (var product in @Model)
        {
            <tr>
                <td>@product.Name</td>
                <td>@product.Price.ToString("C")</td>
                <td>
                    <a asp-controller="Product" asp-action="Details"
                        asp-route-id="@product.ProductID">View</a>
                </td>
                <td>
                    <a asp-controller="Product" asp-action="Update"
                        asp-route-id="@product.ProductID">Update</a>
                </td>
            </tr>
        }
    </tbody>
</table>
```

### The view displayed in a browser

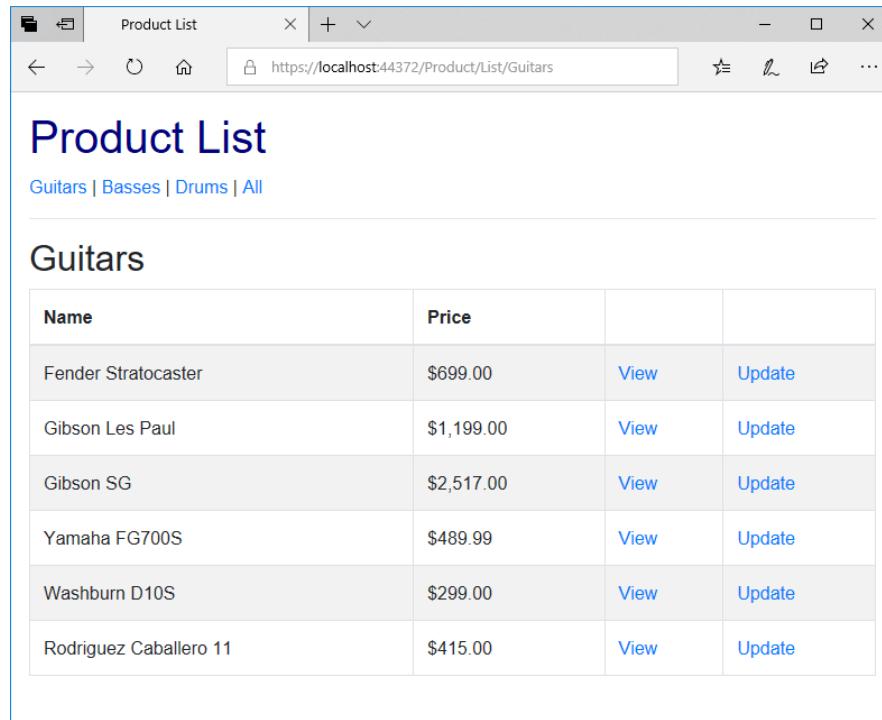


Figure 7-16 How to display a list of model objects (part 2)

## How to work with Razor layouts

---

So far in this book, you have learned how to create web apps that use a single Razor layout, and you've been setting this layout as the default. As a result, all views in the app use this layout. However, as your web app becomes more complex, you may need to create additional Razor layouts. That way, different views can use different layouts.

### How to create and apply a layout

---

In chapter 2, you learned how to use Visual Studio to add a Razor layout to your web app. In particular, you learned how to add a layout like the `_Layout` file presented earlier in this chapter in figure 7-6. Now, figure 7-17 shows the code for a layout file named `_MainLayout`. Like the `_Layout` file, this file is stored in the `Views/Shared` folder. That makes sense as both of these files are shared by multiple views.

The first example shows the code for the `_MainLayout` file. This code works much like the code for the `_Layout` file. However, within the `<body>` element, it includes `<header>` and `<footer>` elements that define a header and footer for the page. Here, the header contains three links (Home, Products, and About) and an `<hr>` element that displays a horizontal rule. The footer also uses an `<hr>` element to display a horizontal rule, along with copyright info for the website. This info consists of the HTML entity for the copyright symbol (`&copy;`), a Razor expression that gets the current year, and the name of the website.

The second example shows the code in the `_ViewStart` file that sets the `Layout` property to the layout named `_Layout`. This causes the layout named `_Layout` to be the default layout for all views.

The third example shows how to override this default for a specific view. To do that, you can use the Razor code block at the beginning of the view to set the `Layout` property to another layout. In this example, the Home/Index view sets the `Layout` property to the `_MainLayout` file shown at the top of this figure. That's why the Home page shown in the browser includes the header and footer.

## The Views/Shared/\_MainLayout.cshtml file

```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link rel="stylesheet" href="~/lib/bootstrap/css/bootstrap.min.css" />
    <link rel="stylesheet" href="~/css/custom.css" />
</head>
<body>
    <header>
        <a asp-controller="Home" asp-action="Index">Home</a> | 
        <a asp-controller="Product" asp-action="List">Products</a> | 
        <a asp-controller="Home" asp-action="About">About</a>
        <hr />
    </header>

    @RenderBody()

    <footer>
        <hr />
        <p>&copy; @DateTime.Now.Year - Guitar Shop</p>
    </footer>
</body>
</html>
```

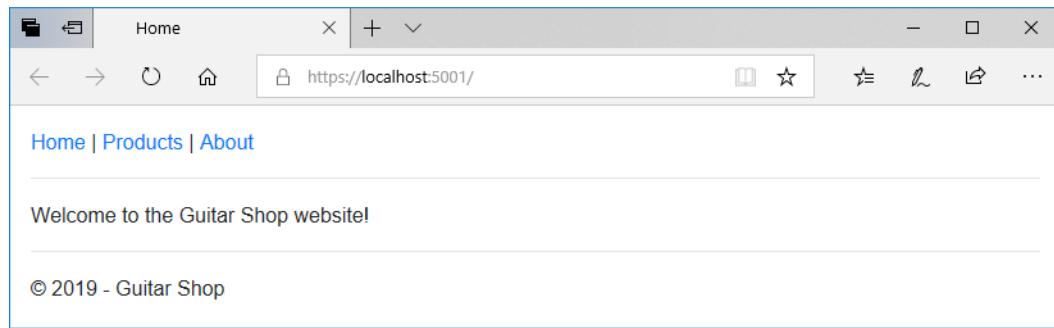
## The code for a \_ViewStart.cshtml file that sets the default layout

```
@{
    Layout = "_Layout";
}
```

## The code for a Home/Index view that explicitly specifies a layout

```
@{
    Layout = "_MainLayout";
    ViewBag.Title = "Home";
}
<p>Welcome to the Guitar Shop website!</p>
```

## The Home page



## Description

- A project can contain multiple Razor layouts. That way, different views can use different layouts.

Figure 7-17 How to create and apply a layout

## How to nest layouts

---

If you compare the layout named `_Layout` in figure 7-6 with the layout named `_MainLayout` in figure 7-17, you'll see that these two layouts contain some duplicate code. For example, they both use the same `<head>` element. In some cases, this duplication is fine. In general, though, it's considered a good practice to avoid code duplication because it usually makes your code easier to maintain.

When working with layouts, you can avoid code duplication by nesting one layout within another as shown in figure 7-18. To illustrate how this works, this figure shows the code for three layouts. To start, the layout named `_Layout` specifies the `<html>`, `<head>`, and `<body>` elements that can be used by a simple view and by other layouts.

The layout named `_MainLayout` sets its `Layout` property to the layout named `_Layout`, which causes it to be nested within this layout. As a result, the `<header>` and `<footer>` elements specified by `_MainLayout` are rendered within the `<body>` element of the layout named `_Layout`.

Similarly, the layout named `_ProductLayout` is nested within the layout named `_MainLayout`. As a result, the `<h1>` element, the series of links, and the `<hr>` element specified by `_ProductLayout` are rendered between the `<header>` and `<footer>` elements of `_MainLayout`.

## The \_Layout file

```
<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link rel="stylesheet" href="~/lib/bootstrap/css/bootstrap.min.css" />
    <link rel="stylesheet" href="~/css/custom.css" />
</head>
<body>
    @RenderBody()
</body>
</html>
```

## The \_MainLayout file

```
@{
    Layout = "_Layout";
}

<header>
    <a asp-controller="Home" asp-action="Index">Home</a> |
    <a asp-controller="Product" asp-action="List"
       asp-route-id="All">Products</a> |
    <a asp-controller="Home" asp-action="About">About</a>
    <hr />
</header>

@RenderBody()

<footer>
    <hr />
    <p>&copy; @DateTime.Now.Year - Guitar Shop</p>
</footer>
```

## The \_ProductLayout file

```
@{
    Layout = "_MainLayout";
}

<h1>Product Manager</h1>
@if (ViewBag.Categories != null)
{
    foreach (Category c in ViewBag.Categories)
    {
        <a asp-controller="Product" asp-action="List"
           asp-route-id="@c.Name">@c.Name</a><text> | </text>
    }
    <a asp-controller="Product" asp-action="List"
       asp-route-id="All">All</a>
}
<hr />

@RenderBody()
```

## Description

- To avoid code duplication, you can build layouts by nesting one layout within another.

---

Figure 7-18 How to nest layouts (part 1)

To help you visualize how this works, part 2 begins by showing the code for a Product List view that uses \_ProductLayout. Here, the code for \_MainLayout displays the header that includes the Home, Products, and About links as well as the footer that displays the copyright info. Then, the code for \_ProductLayout displays the Product List heading and the links for the product categories.

As you review this figure, take a moment to consider how you might use these three layouts for the pages of your web app. You can use \_Layout for any custom views that need to link to the CSS files but don't need the standard header and footer for the web app. You can use \_MainLayout to provide the standard header and footer for most pages of the web app. And you can use \_ProductLayout to provide the Product Manager heading and category links for any pages that are used to view and update products.

### The code for a view that uses the \_ProductLayout view

```
@model List<Product>
{
    Layout = "_ProductLayout";
    ViewBag.Title = "Product List";
}

@if (@ViewBag.SelectedCategory == "All")
{
    <h2>All Products</h2>
}
else
{
    <h2>@ViewBag.SelectedCategory</h2>
}

<!-- The rest of the code is the same as the &lt;table&gt; element
     in part 2 of figure 7-16 --&gt;</pre>
```

### The view displayed in a browser

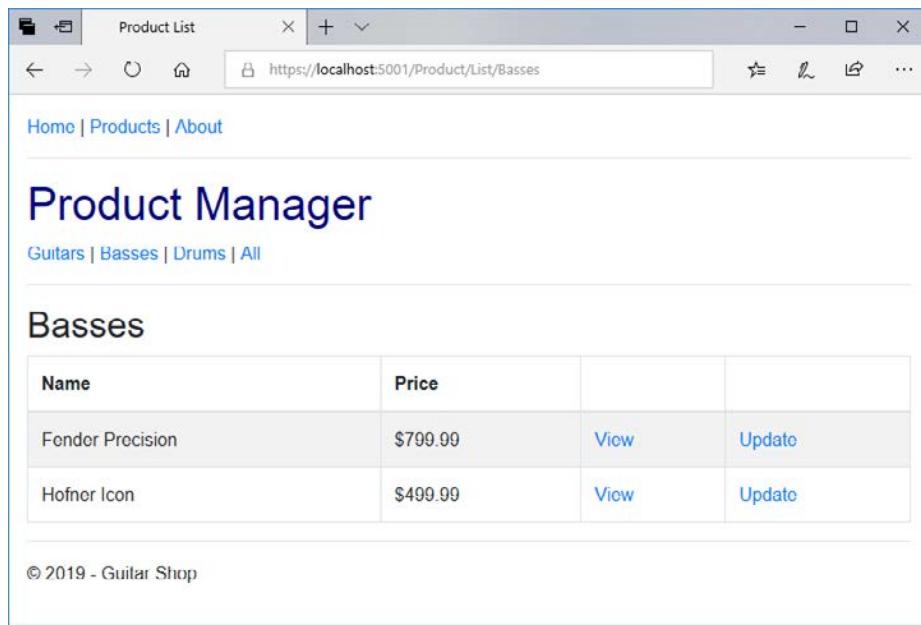


Figure 7-18 How to nest layouts (part 2)

## How to use view context

---

The ViewContext property is available to all views and layouts. A layout can use this property to get data about the context of the current view, such as its route. Then, the layout can use this data to perform formatting tasks such as setting the active navigation link.

To show how this works, figure 7-19 presents a modified version of the \_MainLayout that was presented earlier in this chapter. To start, the Razor code block for this layout uses the ViewContext property to get the current controller and action. Then, it uses a navbar to display the Home, Products, and About links using Bootstrap skills described in chapter 3. For each link, this layout uses an inline conditional expression to display the active attribute for the current controller and action.

For example, if the current controller is the Product controller and the current action is the List() action, this code sets the active attribute for the Products link. This displays the Products link as the active link. Otherwise, this code doesn't set the active attribute for the Products link. This displays the Products link as an inactive link.

To help you visualize how this works, this figure finishes by showing this layout after it has been displayed in a browser. Here, a navbar is displayed across the top of the screen. Within the navbar, the Products link is displayed in a brighter color to indicate that it's the active link, and the Home and About links are displayed in a more muted color to show that they are inactive links.

## The \_MainLayout file after it has been modified to use view context

```

@{
    Layout = "_Layout";
    string controller =
        ViewContext.RouteData.Values["controller"].ToString();
    string action =
        ViewContext.RouteData.Values["action"].ToString();
}

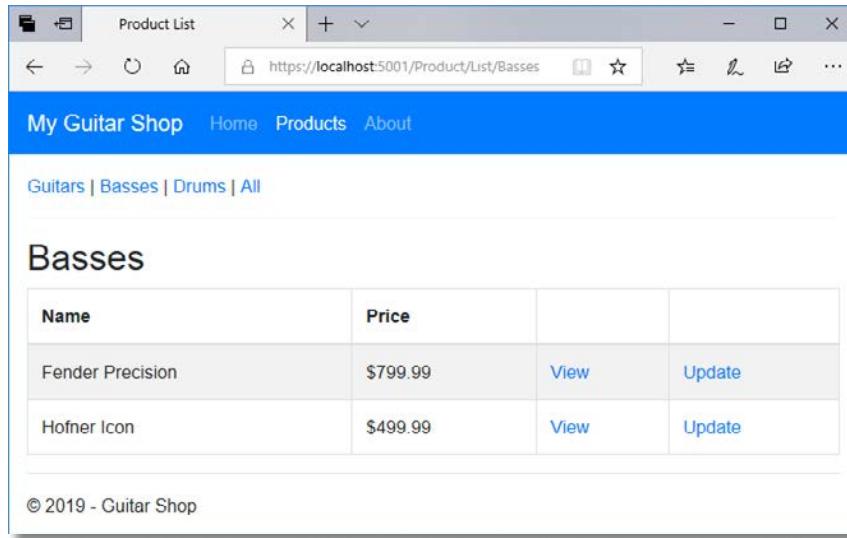
<nav class="navbar navbar-dark bg-primary fixed-top">
    <a class="navbar-brand" href="/">My Guitar Shop</a>
    <div class="navbar-nav">
        <a class="nav-link"
            @(controller == "Home" && action == "Index" ? "active" : "")>
            asp-controller="Home" asp-action="Index">Home</a>
        <a class="nav-link"
            @(controller == "Product" ? "active" : "")>
            asp-controller="Product" asp-action="List">Products</a>
        <a class="nav-link"
            @(controller == "Home" && action == "About" ? "active" : "")>
            asp-controller="Home" asp-action="About">About</a>
    </div>
</nav>

@RenderBody()

<footer>
    <hr />
    <p>© &copy; @DateTime.Now.Year - Guitar Shop</p>
</footer>

```

## A view that uses this layout



## Description

- A layout can use the ViewContext property to get data about the route of the current view. Then, it can use this data to perform tasks such as setting the active navigation link.

Figure 7-19 How to use view context in a layout

## How to use sections

---

Within a view, you can use Razor code to specify a *section*, which is a block of content that has a name. In figure 7-20, for instance, the first example specifies a section named scripts that contains two `<script>` elements. These elements link to the jQuery libraries that are necessary to perform client-side data validation. As a result, it makes sense to add them for a view that needs to perform data validation, such as a view that updates existing data or adds new data.

Within a layout, you can use the `RenderSection()` method to insert the content from a section into the layout. For instance, the second example uses the `RenderSection()` method to insert the section named scripts into the layout file. This inserts the two `<script>` elements for data validation libraries after the other three `<script>` elements.

In this example, the `RenderSection()` method uses the second argument to specify that the section is optional. As a result, views that need to add the data validation scripts can insert them into the layout. However, views that don't need these files don't need to insert anything here.

Sections are commonly used to insert `<script>` elements for additional JavaScript files into a layout as shown in this figure. This allows the views that don't need these scripts to load more quickly.

Although sections aren't necessary, they make your code more flexible because they make it easy to change the location of the section, even if multiple views use a section. For example, if you decide that you'd like to load your scripts after most of the HTML has been loaded, you can move the `RenderSection()` method for the scripts section to the end of the `<body>` element instead of coding it at the end of the `<head>` element. Since this loads the HTML more quickly, it may appear to improve the load time of the view. However, it may also cause HTML elements to jump around after the view finishes loading.

## Code in a view file that specifies a section

```

@model Product
{
    ViewBag.Title = "Update Product";
}

@section scripts {
    <script src="~/lib/jquery-validate/jquery.validate.min.js"></script>
    <script src="~/lib/jquery-validation-unobtrusive/
        jquery.validate.unobtrusive.min.js"></script>
}

<h2>Update</h2>

// the HTML elements for the rest of the view body go here

```

## A method that can insert content from a section into a layout

Method	Description
<code>RenderSection(name,.isRequired)</code>	Inserts the content from the section with the specified name into the layout. If the section is required, all views must include a section with the specified name or attempting to view the page will result in an error.

## A layout file that specifies an optional section that can be inserted

```

<!DOCTYPE html>
<html>
<head>
    <title>@ViewBag.Title</title>
    <link rel="stylesheet" href="~/lib/bootstrap/css/bootstrap.min.css" />
    <link rel="stylesheet" href="~/css/custom.css" />

    <script src="~/lib/jquery/jquery.min.js"></script>
    <script src="~/lib/popper.js/popper.min.js"></script>
    <script src="~/lib/bootstrap/js/bootstrap.min.js"></script>
    @RenderSection("scripts", false)
</head>
<body>
    @RenderBody()
</body>
</html>

```

## Description

- Within a view, you can use Razor code to specify a *section*, which is a block of content that has a name.
- Within a layout, you can use the `RenderSection()` method to insert the content from a section into the layout.
- Sections are commonly used to insert `<script>` elements for additional JavaScript files into a layout. That way, the views that need those files can insert them into the layout, and the views that don't need these files don't insert anything into the layout.

Figure 7-20 How to use sections in a layout

## The Guitar Shop website

---

To show how the skills described in chapters 6 and 7 fit together within a semi-realistic website, this chapter ends by showing the user interface for the Guitar Shop website that's included with the download for this book. If you study this user interface, you might be able to guess how it uses some of the skills presented in chapters 6 and 7.

But really, the best way to study this web app is to open it in Visual Studio, run it to see how it works, and study the code files. To help you understand the code, these files have more comments than would typically be included in a web app. In addition, the exercise at the end of this chapter guides you through the process of reviewing this code and modifying it to make improvements to the website.

### The user interface for customers

---

Figure 7-21 shows two pages of the user interface that's available to end users, or customers, of the Guitar Shop website. Both of these pages display the same navigation bar at the top of the page and the same footer at the bottom. That's because the views for these pages use the same layout.

The Product List page includes a column of categories on the left side of the page that customers can use to select a category of products. Then, if customers want to learn more about a product, they can click the View Details link to display the Product Details page. This page displays details about the product, including an image. On both pages, the customer can add the product to the cart, though this feature hasn't been implemented yet by this website.

If users click the Admin link on the right side of the navigation bar, the website displays the Admin pages shown in figure 7-22. Typically, a website would require admin users, or administrators, to log in to make sure the user is authorized to view the Admin pages. However, this feature hasn't been implemented yet by this website.

### The user interface for administrators

---

Figure 7-22 shows four pages of the user interface for administrators. Like the pages for end users, the views for these pages use the same layout. However, this layout is different from one that's displayed for end users. That's because the Admin pages provide different functionality.

The Product Manager page looks much like the Product List page for end users. However, it provides links that let admin users update or delete a product. In addition, it provides a button that lets admin users add a new product. To perform these tasks, admin users can use the Add Product, Update Product, and Delete Product pages shown in this figure.

## The Product List page for customers

This screenshot shows the 'Product List' page for customers. The browser title is 'Product List - Guitar Shop'. The URL in the address bar is 'https://localhost:5001/Products/Guitars'. The page header includes 'My Guitar Shop' and 'Admin' links. The main content area is titled 'Product List' and contains a table of guitars. The table has columns for 'Name' and 'Price'. Each row includes 'View Details' and 'Add to Cart' buttons. A sidebar on the left lists categories: 'All', 'Guitars' (which is selected and highlighted in blue), 'Basses', and 'Drums'. At the bottom of the page is a copyright notice: '© 2019 - Guitar Shop'.

Name	Price		
Fender Stratocaster	\$699.00	<a href="#">View Details</a>	<a href="#">Add to Cart</a>
Gibson Les Paul	\$1,199.00	<a href="#">View Details</a>	<a href="#">Add to Cart</a>
Gibson SG	\$2,517.00	<a href="#">View Details</a>	<a href="#">Add to Cart</a>
Yamaha FG700S	\$489.99	<a href="#">View Details</a>	<a href="#">Add to Cart</a>
Washburn D10S	\$299.00	<a href="#">View Details</a>	<a href="#">Add to Cart</a>
Rodriguez Caballero 11	\$415.00	<a href="#">View Details</a>	<a href="#">Add to Cart</a>

## The Product Details page

This screenshot shows the 'Product Details' page for the 'Gibson Les Paul' guitar. The browser title is 'Product Details - Guitar Shop'. The URL in the address bar is 'https://localhost:5001/Product/Details/2/Gibson-Les-Paul'. The page header includes 'My Guitar Shop' and 'Admin' links. The main content area is titled 'Product Details' and shows a product image of a Gibson Les Paul guitar. To the right, the product name 'Gibson Les Paul' is displayed. Below the image, there is a table with three rows: 'List Price:' (\$1,199.00), 'Discount Percent:' (20%), and 'Your Price:' (\$959.20). A note indicates a savings of '\$239.80'. A large 'Add to Cart' button is located at the bottom of this section. At the very bottom of the page is a copyright notice: '© 2019 - Guitar Shop'.

Figure 7-21 The Guitar Shop user interface for customers

### The Product Manager page for administrators

The screenshot shows a web browser window titled "Product Manager - Guitars". The URL is https://localhost:5001/Admin/Products/Guitars. The page has a header with "My Guitar Shop" and "Admin Products Categories" and a "View Site" link. Below the header is the title "Product Manager". On the left, there is a sidebar with buttons for "All", "Guitars" (which is selected and highlighted in blue), "Basses", and "Drums". The main content area displays a table with columns "Name" and "Price". The table contains the following data:

Name	Price	Update	Delete
Fender Stratocaster	\$699.00	<a href="#">Update</a>	<a href="#">Delete</a>
Gibson Les Paul	\$1,199.00	<a href="#">Update</a>	<a href="#">Delete</a>
Gibson SG	\$2,517.00	<a href="#">Update</a>	<a href="#">Delete</a>
Yamaha FG700S	\$489.99	<a href="#">Update</a>	<a href="#">Delete</a>
Washburn D10S	\$299.00	<a href="#">Update</a>	<a href="#">Delete</a>
Rodriguez Caballero 11	\$415.00	<a href="#">Update</a>	<a href="#">Delete</a>

At the bottom of the table is a blue button labeled "Add New Product". The footer of the page includes the copyright notice "© 2019 - Guitar Shop".

### The Update Product page

The screenshot shows a web browser window titled "Update Product - Guitars". The URL is https://localhost:5001/Admin/Product/Update/2. The page has a header with "My Guitar Shop" and "Admin Products Categories" and a "View Site" link. Below the header is the title "Update Product". The page contains a form with fields for "Category" (set to "Guitars"), "Code" (set to "les\_paul"), "Name" (set to "Gibson Les Paul"), and "Price" (set to "1199.00"). At the bottom of the form are two buttons: "Update" (highlighted in blue) and "Cancel". The footer of the page includes the copyright notice "© 2019 - Guitar Shop".

Figure 7-22 The Guitar Shop user interface for administrators (part 1)

## The Add Product page

The screenshot shows a browser window titled "Add Product - Guitar Sh". The address bar displays "https://localhost:5001/Admin/Product/Add". The page header includes "My Guitar Shop", "Admin", "Products", "Categories", and "View Site". The main content area is titled "Add Product". It contains four input fields: "Category" (dropdown menu showing "Guitars"), "Code" (text input showing "test"), "Name" (text input showing "Test"), and "Price" (text input showing "650"). At the bottom are two buttons: "Add" and "Cancel". A copyright notice "© 2019 - Guitar Shop" is at the bottom.

## The Delete Product page

The screenshot shows a browser window titled "Delete Product - Guitar". The address bar displays "https://localhost:5001/Admin/Product/Delete/13". The page header includes "My Guitar Shop", "Admin", "Products", "Categories", and "View Site". The main content area is titled "Confirm Deletion" and displays the name "Test". It contains two buttons: "Delete" and "Cancel". A copyright notice "© 2019 - Guitar Shop" is at the bottom.

Figure 7-22 The Guitar Shop user interface for administrators (part 2)

## Perspective

---

Now that you've read this chapter, you should have a general understanding of how to set up the Razor views and layouts of an MVC app. In addition, you should have a good idea of how to create a model and bind it to a view. With that as background, you're ready to learn more about transferring data between controllers and views as described in the next chapter.

### Terms

---

code block	fragment
inline expression	relative URL
inline loop	absolute URL
inline conditional statement	model
inline conditional expression	section

### Summary

---

- Within a view, you can use a Razor *code block* to execute one or more C# statements.
- Within a view, you can code an *inline expression* to evaluate a C# expression and display its result.
- Within a view, you can code *inline loops*. Within these loops, you can use HTML tags to send HTML to the view.
- Within a view, you can code *inline conditional statements* such as if/else and switch statements. Within these statements, you can use HTML tags to send HTML to the view.
- Within a view, you can code an *inline conditional expression* by using C#'s conditional operator (? :) to send HTML tags or plain text to the view. This works like other complex inline expressions that require parentheses.
- A URL *fragment* allows you to jump to a specified placeholder on a web page. In a URL, it is preceded by the hash mark (#).
- A *relative URL* is a URL that's relative to the app's root directory.
- An *absolute URL* is a URL that specifies the name of the host.
- A *model* is a regular C# class that defines an object that stores and processes the data for an app.
- Within a view, you can specify a *section*, which is a block of content that has a name. Then, you can use the RenderSection() method to insert the section into a layout.

## Exercise 7-1 Modify the Guitar Shop website

In this exercise, you'll review the folders and files that are used by the Guitar Shop website. Then, you'll modify some of these files to see how those changes affect the app.

### Open the Guitar Shop website and review its files

1. Open the Ch07Ex1GuitarShop website in the ex\_starts folder.
2. Run the app. It should display the Home page in your browser.
3. Click the “View our products” link. It should display the Product List page.  
If you get an error message that says, “Cannot open database”, create the database by running the Update-Database command as described in chapter 4.
4. The left column of the Product List page should display product categories of All, Guitars, Basses, and Drums. Click on each of these and note both the items listed for each category and the URL for each category.
5. Open the ProductController class and the corresponding Product/List.cshtml view file. Make sure to open the files in the main area, not the files in the Admin area.
6. In the ProductController class, note that the List() method has a Route attribute that allows it to be called with a URL of “/Products”, not “/Product/ List”. Note also that the id parameter of this method has a default value of “All”.
7. In the List.cshtml file, view the code and note that the @model directive at the top of the code specifies a list of Product objects as the model.
8. In the ProductController class, note that the List() method declares and populates the list of Product objects and passes it to the view as the model. To do that, it uses an if statement to check the id parameter. Then, it uses LINQ method syntax to populate the list with products from the correct category.
9. In the List.cshtml file, note that the foreach loop iterates through the Model property (which contains the list of products) and displays the appropriately formatted data related to each product.

### Modify the Product controller so it can display stringed instruments

10. In the ProductController class, modify the List() action method by adding an else clause to the if statement that checks whether the id parameter is equal to “Strings”. In this context, “Strings” refers to stringed instruments such as guitars and basses, but not to other kinds of instruments such as drums.
11. In the else clause that you added to the if statement, code a LINQ method chain that gets all products where the category name is equal to Guitars or Basses and sorts these products by their ProductID. Look at the other clauses in the if statement for ideas, but note that you need to use string literals of “Guitars” and “Basses” to specify the categories.
12. Run the app and view the products again.

13. Manually edit the URL so it specifies an id segment of “Strings” like this:

`https://localhost:5001/Products/Strings`

This should display all the guitars and basses, but not the drums.

14. In the ProductController class, modify the List() action method so it assigns a literal value of “Strings” to the ViewBag.SelectedCategoryName if the id parameter equals “Strings”. To do that, you can add an if statement after the statement that sets the SelectedCategoryName to the value of the id parameter.

### Modify the view so it can display stringed instruments

15. In the List.cshtml file, find the Razor code that uses a foreach loop to iterate through the categories stored in the ViewBag.

16. After the closing curly brace of the foreach loop, add a link for the Strings category. To do that, you can add an <a> element that displays text that says “Strings”, a route ID of “Strings”, and a class attribute like this:

```
class="list-group-item  
@(ViewBag.SelectedCategoryName == "Strings" ? "active" : "")"
```

This attribute adds the active class to the Strings link if the Strings category is selected.

17. Run the app and view the Product List page again. It should display a Strings link in the list of categories that’s displayed in the left column.

18. Click the Strings link. It should display the stringed instruments. In other words, it should display the basses and guitars, but not the drums.

19. Note that this app works fine for now since the Guitar Shop only carries guitars, basses, and drums. However, if the Guitar Shop added another category of stringed instruments such as banjos, you would need to update the code in the controller to add banjos to the list of Product objects.

# 8

## How to transfer data from controllers

So far, this book has focused on using the action methods of a controller to get data from the model layer and then transfer that data to a view. Now, this chapter presents more details about how that works. In addition, it presents several new ways to transfer data from a controller to a view or to another controller.

<b>How to use ActionResult objects .....</b>	<b>280</b>
An introduction to ActionResult subtypes .....	280
How to return ActionResult objects .....	282
<b>How to use the ViewData and ViewBag properties .....</b>	<b>284</b>
How to use the ViewData property .....	284
How to use the ViewBag property.....	286
<b>The NFL Teams 1.0 app.....</b>	<b>288</b>
The user interface .....	288
The model layer .....	290
The Home controller.....	292
The layout .....	294
The Home/Index view .....	296
<b>How to work with view models.....</b>	<b>298</b>
How to create a view model .....	298
The updated Index() action method .....	300
The updated Home/Index view .....	300
<b>How to redirect a request .....</b>	<b>302</b>
How to use the ActionResult classes for redirection.....	302
How to use the Post-Redirect-Get pattern.....	304
<b>How to use the TempData property.....</b>	<b>306</b>
How to get started with TempData.....	306
How to use methods of the TempData dictionary .....	308
<b>The NFL Teams 2.0 app .....</b>	<b>310</b>
The user interface .....	310
The view model classes .....	312
The Details() action method.....	312
The Home/Index view .....	314
The Home/Details view.....	316
<b>Perspective .....</b>	<b>318</b>

## How to use ActionResult objects

---

The apps you've seen so far in this book have transferred data from a controller to a view by passing a model object to the View() method. This method returns a ViewResult object, which is a subtype of the ActionResult class. Now, this chapter presents more details about how you can use the ActionResult class to transfer data from a controller to a view.

### An introduction to ActionResult subtypes

---

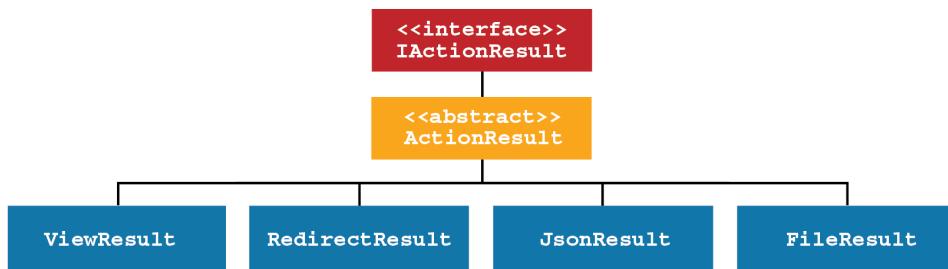
The ActionResult type provides information to MVC about the type of HTTP response an action method should return. The response can be content, like HTML or JSON, or it can be an HTTP status code such as 404 Not Found.

Figure 8-1 begins by presenting the hierarchy of the ActionResult class. Here, the abstract ActionResult class implements the IActionResult interface. Then, several subtypes inherit and extend the ActionResult class.

The table in this figure lists some of the most common subtypes of the ActionResult type. Each subtype has a specific purpose. For instance, the ViewResult class renders a view and sends the resulting HTML to the browser. The JsonResult class sends JSON to the browser. The FileResult class sends a file to the browser. The RedirectToActionResult class uses HTTP to instruct the browser to redirect to a URL that corresponds to the specified action method. And so on.

This table is not an exhaustive list. If you'd like to learn more about all the subtypes of the ActionResult class, the Microsoft documentation provides a full list, and you can find it at the URL shown in this figure.

## The ActionResult hierarchy



## Common ActionResult subtypes

Class	Description
<code>ViewResult</code>	Renders a specified view as HTML and sends it to the browser.
<code>RedirectResult</code>	Performs an HTTP redirection to the specified URL.
<code>RedirectToActionResult</code>	Performs an HTTP redirection to a URL that's created by the routing system using specified controller and action data.
<code>JsonResult</code>	Serializes an object to JSON and sends the JSON to the browser.
<code>FileResult</code>	Returns a file to the browser.
<code>StatusCodeResult</code>	Sends an HTTP response with a status code to the browser.
<code>ContentResult</code>	Returns plain text to the browser.
<code>EmptyResult</code>	Returns an empty response to the browser.

## A URL that provides a full list of the ActionResult subtypes

<https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.mvc.actionresult>

### Description

- Within a controller, an action method can return any type of ActionResult object. The ActionResult class is an abstract class that implements the IActionResult interface.
- Since the ActionResult class has many subtypes, an action method can return many different types of result objects.

---

Figure 8-1 An introduction to the ActionResult subtypes

## How to return ActionResult objects

---

The ActionResult object returned by an action method tells MVC the type of response that should be sent to the browser. The Controller class provides several methods that create an ActionResult object. To create an appropriate result object, you can call these methods. Then, you can return the result object.

The first table in figure 8-2 presents some of methods of the Controller class that return an ActionResult object. The names of these methods correspond to the ActionResult subtype they return. For instance, the View() method creates a ViewResult object, the RedirectToAction() method creates a RedirectToActionResult object, and so on.

Most of these methods have several overloads. The second table illustrates this by reviewing the overloads of the View() method. The first overload accepts no arguments and renders the default view for the current controller and action method. For instance, if an Index() method in the Home controller calls the View() method with no arguments, MVC renders the HTML for the Home/Index view.

The second overload accepts a model object that's used by the default view. For instance, if an Index() action method in the Home controller passes an object to the View() method, MVC transfers that object to the Home/Index view. Then, the view uses that object to render its HTML.

The third overload accepts a string that tells MVC which view file to use. MVC starts by looking for this file in the folder for the current controller. Then, it looks in the Shared folder. For instance, if an Index() action method in the Home controller passes the string "Error" to the View() method, MVC looks for the Error.cshtml file in the Views/Home folder. Then, it looks in the Views/Shared folder.

The fourth overload accepts a string and a model object. Thus, if an Index() action method in the Movie controller passes the string "Edit" to the View() method, MVC passes the specified model object to the Edit.cshtml file in the Views/Movie folder.

When Visual Studio generates action methods, it uses a return type of IActionResult. If the action method might return different ActionResult subtypes, this approach works well, as shown by the third example in this figure. However, if your action method is only going to return one type of ActionResult subtype, it's considered a good practice to use a more specific subtype as the return type as shown by the first two examples. This can improve performance, and it makes your code more clear.

A problem can arise if you want to use a string as a model object. In that case, MVC might interpret the string as the name of a view file to look for, not as the model object. To fix this problem, you can cast the string to the object type before passing it to the View() method as shown by the third code example.

## Some methods of the Controller class that return an ActionResult object

Method	Creates
<code>View()</code>	ViewResult object
<code>Redirect()</code>	RedirectResult object
<code>RedirectToAction()</code>	RedirectToActionResult object
<code>File()</code>	FileResult object
<code>Json()</code>	JsonResult object

## Some of the overloads of the View() method

Method	Description
<code>View()</code>	Renders the default view for that controller and action method.
<code>View(model)</code>	Transfers a model object to the default view and renders that view.
<code>View(name)</code>	Renders the specified view. This method starts by searching for the specified view in the view folder for the current controller. Then, it searches in the Views/Shared folder.
<code>View(name, model)</code>	Transfers a model object to the specified view and renders that view.

## An action method that returns a ViewResult object

```
public ViewResult List() {
    var names = new List<string> { "Grace", "Ada", "Charles" };
    return View(names);
}
```

## An action method that returns a RedirectToActionResult object

```
public RedirectToActionResult Index() => RedirectToAction("List");
```

## An action method that may return different types of result objects

```
[HttpPost]
public IActionResult Edit(string id) {
    if (ModelState.IsValid)
        return RedirectToAction("List");
    else
        return View((object)id); // cast string model to object
}
```

## Description

- The Controller class provides several methods that create objects of ActionResult subtypes. Most of these methods include several overloads that allow you to customize the ActionResult object that's created.
- When Visual Studio generates action methods, it uses IActionResult as the return type.
- If an action method only returns one type of ActionResult object, it's a good practice to specify that subtype as the return type.
- If an action method may return multiple types of ActionResult objects, you can specify the ActionResult abstract class or the IActionResult interface as the return type.
- If you want to use a string as a model, you must cast it to an object first.

Figure 8-2 How to return ActionResult objects from controller action methods

## How to use the ViewData and ViewBag properties

---

Up until now, you've seen how to use the ViewBag property of the Controller class to transfer data from an action method to a view. In a moment, this chapter will review the ViewBag property and explain it in more detail. But first, it shows how to use the closely-related ViewData property of the Controller class, which also lets you transfer data to a view.

### How to use the ViewData property

---

The ViewData property of the Controller class has a data type of ViewDataDictionary. This means that the property is a collection of key/value pairs. For this dictionary type, the key is a string and the value is an object.

The first code example in figure 8-3 shows how to add data to the ViewData property. Here, a string value and a double value are added to the dictionary, each with a unique string key.

The ViewData property is shared with the associated view, including the view's layout. As a result, any data stored in the ViewData dictionary is available to the view and its layout.

The ViewDataDictionary class also has some properties that a view can use. The table in this figure summarizes some of these properties. Then, the second code example shows how a view can use these properties to display the data in the ViewData dictionary to the user. To do that, the code uses the Count property to display the number of items, and it uses a KeyValuePair object to get each item and display its key and value.

If you just want to display a ViewData value in the browser, you don't need to do anything special to it. That's because Razor automatically calls the ToString() method of an object in the dictionary. However, if you want to work with a value in the ViewData dictionary in code, you must first cast it to the appropriate data type. For instance, the third code example gets the value for the Price item, casts it to the double type, and then uses the ToString() method to apply currency formatting to the double value.

However, before you call a method from a value, you should check to make sure the value isn't null. Otherwise, your app might throw exceptions. For instance, the fourth code example shows how to check if the Price value is null. If it is, the code sets the value to an empty string. Otherwise, it casts the value to a double type and applies currency formatting. Since this value is of the non-nullable double type, this code uses the conditional operator (? :) to perform this check.

In addition, the fourth example shows how to check that the value for the "Book" key isn't null. Since this value is of the nullable string type, this code uses the safe navigation operator (?.) to perform this check. As a result, if the value for the "Book" key is null, the Razor expression returns null. Otherwise, it returns the string that's stored in the Book value.

## Controller code that adds two items to the ViewData property

```
public ViewResult Index() {
    ViewData["Book"] = "Alice in Wonderland";
    ViewData["Price"] = 9.99;
    return View();
}
```

## Some of the properties of the ViewDataDictionary class

Property	Description
Count	Returns the number of key/value pairs in the dictionary.
Keys	Returns a collection of strings containing the keys in the dictionary.
Values	Returns a collection of objects containing the values in the dictionary.

## Razor code that displays all the items in the ViewData object

```
<h2>@ViewData.Count items in ViewData</h2>
@foreach (KeyValuePair<string, object> item in ViewData) {
    <div>@item.Key - @item.Value</div>
}
```

### The view in the browser

#### 2 items in ViewData

Book - Alice in Wonderland  
Price - 9.99

## Razor code that casts a ViewData object to the double type

```
<h4>Price: @((double)ViewData["Price"]).ToString("c")</h4>
```

### The view in the browser

Price: \$9.99

## Razor code that checks ViewData values for null

### Non-nullable type (double)

```
<h4>Price: @(ViewData["Price"] == null) ? "" :
    ((double)ViewData["Price"]).ToString("c")</h4>
```

### Nullable type (string)

```
<h4>Book: @ViewData["Book"]?.ToString()</h4>
```

## Description

- The Controller class has a ViewData property that lets you transfer data to a view.
- The ViewData property is of the ViewDataDictionary type, which is a collection of key/value pairs where the key is a string and the value is an object.
- When you display a ViewData value in a view, Razor automatically calls the object's ToString() method.
- If you want to work with a ViewData value in code, you must cast the value to its data type. But first, you should check to make sure the value isn't null.

Figure 8-3 How to use the ViewData property

## How to use the ViewBag property

---

The ViewBag property of the Controller class uses C#'s *dynamic type*. As a result, you can add properties to the ViewBag, and .NET determines the type of those properties at runtime.

The first code example at the top of figure 8-4 shows how to add data to the ViewBag property. Here, as in the previous figure, this example adds a string property and a double property.

Like the ViewData property, the ViewBag property is shared with the associated view, including its layout. As a result, any data stored in the ViewBag is available to the view and its layout.

Under the hood, the ViewBag property stores its data in the ViewData dictionary. Thus, you can use the ViewData property to get values that were added to the ViewBag property. For instance, the second code example uses ViewData to display the values that the first example added to the ViewBag. Because the ViewBag uses the ViewData dictionary like this, it's often thought of as an *alias* for the ViewData dictionary.

Since the ViewBag is dynamically typed, you don't need to explicitly cast property values to work with the data that they contain. This is illustrated by the third example. However, it's still a good idea to check for null values. This is illustrated by the fourth example. In both cases, these tasks are easier to do with ViewBag.

In general, the ViewBag is easier to work with than the ViewData dictionary. As a result, we recommend using it in most scenarios. However, there are a few scenarios where it makes sense to use ViewData. For example, you can use ViewData if you need to use names for your keys that aren't valid in C# or if you need to be able to access the properties of the ViewDataDictionary class. In addition, if you use Visual Studio to generate code, it may use the ViewData dictionary. In that case, it may be easier to leave the generated code than to change it to use the ViewBag.

One disadvantage of both the ViewBag and ViewData properties is that Visual Studio doesn't provide compile-time checking or IntelliSense for these properties. This is true for both the controller and the view. Because of this, some programmers avoid using either of these properties and use view models instead as described later in this chapter.

## Controller code that adds two dynamic properties to the ViewBag property

```
public ViewResult Index() {
    ViewBag.Book = "Alice in Wonderland";
    ViewBag.Price = 9.99;
    return View();
}
```

## Razor code that uses ViewData to display ViewBag properties

```
<h2>@ViewData.Count ViewBag properties</h2>
@foreach (KeyValuePair<string, object> item in ViewData) {
    <div>@item.Key - @item.Value</div>
}
```

### The view in the browser

#### 2 ViewBag properties

Book - Alice in Wonderland  
Price - 9.99

## Razor code that works with a ViewBag property without casting

```
<h4>Price: @ViewBag.Price.ToString("c")</h4>
```

### The view in the browser

Price: \$9.99

## Razor code that checks ViewBag properties for null

### Non-nullable type (double)

```
<h4>Price: @ViewBag.Price?.ToString("c")</h4>
```

### Nullable type (string)

```
<h4>Book: @ViewBag.Book?.ToString()</h4>
```

## Use ViewData instead of ViewBag when you need to...

- Use a key name that isn't valid in C#, such as a key that contains spaces.
- Call properties and methods of the ViewDataDictionary class, such as its Count property or its Clear() method.
- Loop through all items in the ViewData dictionary.

## Description

- The Controller class has a ViewBag property that lets you transfer data to a view.
- The ViewBag uses the C# *dynamic type* to let you add properties. Later, .NET determines the type of those properties at runtime.
- Under the covers, the ViewBag uses the ViewData dictionary to store its dynamic properties. Thus, you can think of the ViewBag property as an *alias* for ViewData.
- In most scenarios, ViewBag is easier to use. However, there are a few scenarios where it makes more sense to use ViewData.

---

Figure 8-4 How to use the ViewBag property

## The NFL Teams 1.0 app

---

The next few figures present the NFL Teams app. This simple app illustrates some of the skills presented earlier in this chapter.

### The user interface

---

To understand the NFL Teams app, you need to know a little about the conferences and divisions of the National Football League (NFL). This information is presented at the top of figure 8-5. Then, this figure presents the user interface of this app.

The first page shows how the NFL Teams app looks when it first loads. In this case, logos for all the NFL teams are displayed in alphabetical order. When users move the mouse over a team logo, that team's name, conference, and division are displayed in a tool tip. For example, on the first page, the user's mouse is over the first logo. This logo is for the Arizona Cardinals, whose conference is the NFC and whose division is West.

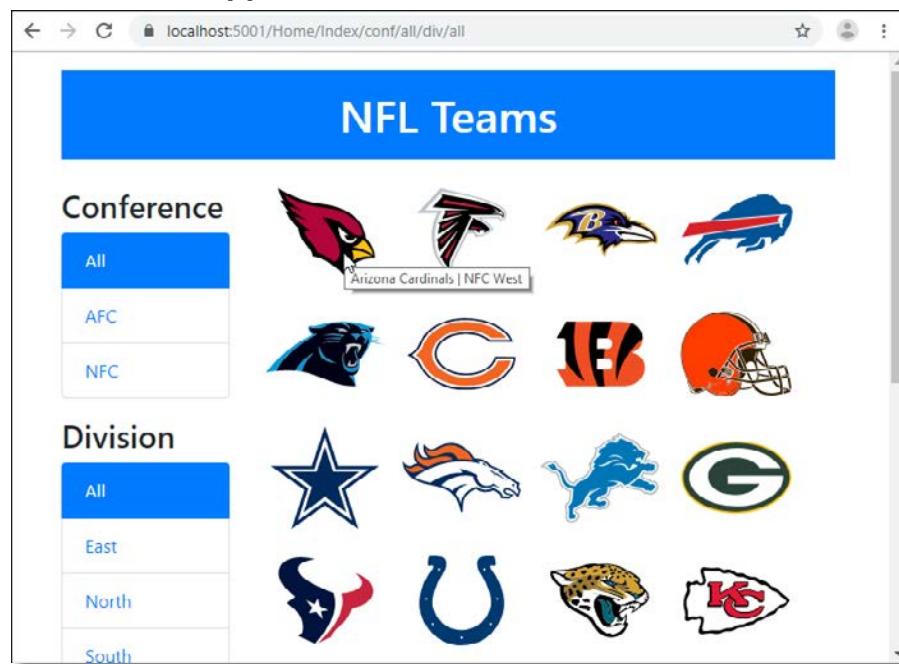
The left side of the page displays two sections with links the user can click to filter the teams by conference or division. Each section also includes an "All" option that allows the user to display all the teams in a conference or division.

The second page shows the app after the user has clicked on the AFC button in the Conference section and the East button in the Division section. As a result, the app only displays the logos of the teams in the East division of the AFC conference, or the AFC East for short.

## How the teams of the National Football League (NFL) are organized

- There are two conferences, the NFC and the AFC.
- Each conference contains four divisions named North, South, East, and West.
- Each division contains four teams.

### The NFL Teams app on first load



### The NFL Teams app after a conference and division are selected

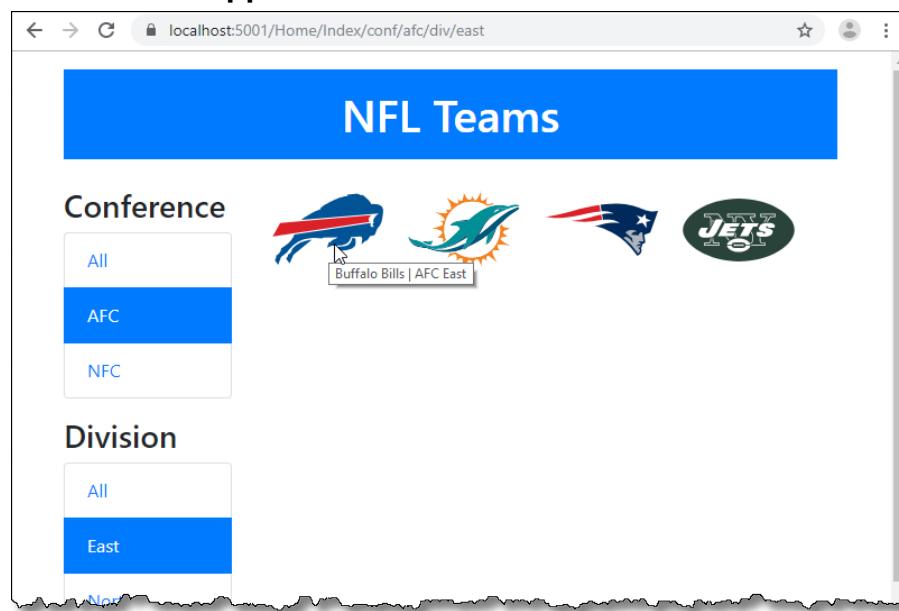


Figure 8-5 The user interface of the NFL Teams 1.0 app

## The model layer

---

Figure 8-6 presents the code for the model layer of the NFL Teams app. This code is stored in the Models folder.

The Conference class is an entity class that represents a conference. It has two string properties, one for an ID value and the other for the name of the conference.

The Division class is an entity class that represents a division. Like the Conference class, it has two string properties, one for an ID value and the other for the name of the division.

The Team class is an entity class that represents a team. It has string properties for an ID value and the team name. In addition, it has two properties of the Conference and Division types to hold information about the conference and division for the team. Finally, it has a string property for the filename of the logo image.

The TeamContext class is a DB context class like the MovieContext class you learned about in chapter 4. The NFL Teams app uses this class to communicate with a database named NFLTeams. This database holds the information about the NFL teams. While not shown here, this app also has an appsettings.json file and a Startup.cs file configured for database access. You can refer to figure 4-6 to review how these files work.

The TeamContext class has three DbSet properties named Teams, Conferences, and Divisions. The TeamContext class also overrides the OnModelCreating() method of the DbContext class to seed initial data about the conferences, divisions, and teams.

### The Conference class

```
public class Conference
{
    public string ConferenceID { get; set; }
    public string Name { get; set; }
}
```

### The Division class

```
public class Division
{
    public string DivisionID { get; set; }
    public string Name { get; set; }
}
```

### The Team class

```
public class Team
{
    public string TeamID { get; set; }
    public string Name { get; set; }
    public Conference Conference { get; set; }
    public Division Division { get; set; }
    public string LogoImage { get; set; }
}
```

### The TeamContext class

```
public class TeamContext : DbContext
{
    public TeamContext(DbContextOptions<TeamContext> options)
        : base(options) { }

    public DbSet<Team> Teams { get; set; }
    public DbSet<Conference> Conferences { get; set; }
    public DbSet<Division> Divisions { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        base.OnModelCreating(modelBuilder);

        modelBuilder.Entity<Conference>().HasData(
            new Conference { ConferenceID = "afc", Name = "AFC" },
            ...
        );

        modelBuilder.Entity<Division>().HasData(
            new Division { DivisionID = "north", Name = "North" },
            ...
        );

        modelBuilder.Entity<Team>().HasData(
            new Team { TeamID = "ari", Name = "Arizona Cardinals",
                ConferenceID = "nfc", DivisionID = "west",
                LogoImage = "ARI.png" },
            ...
        );
    }
}
```

---

Figure 8-6 The model layer

## The Home controller

---

Figure 8-7 presents the code for the Home controller of the NFL Teams app. This controller starts by defining a private TeamContext property named context. Then, it defines a constructor that accepts a TeamContext object. When the app starts, MVC automatically creates a TeamContext object and passes it to the constructor. Then, the constructor stores the TeamContext object in the private context property.

The Home controller has a single action method named Index(). This action method returns a ViewResult object, not any other type of ActionResult object. To make that clear, this code specifies ViewResult as the return type, not ActionResult or IActionResult. As an added benefit, the compiler doesn't need to determine the type of object to return at runtime, which can improve performance.

The Index() action method has two string parameters named activeConf and activeDiv. These parameters are the IDs of the currently selected conference and division. Each has a default value of "all". So, if the user doesn't select a conference or division, this action returns all teams.

Since this action accepts two parameters, it doesn't work well with the default route, which is designed to work with a single parameter. To fix this, this app adds the custom route shown at the bottom of this figure. This route uses two static segments of "conf" and "div" as well as two required parameters named activeConf and activeDiv. Since this route is more specific than the default route, it must be coded before the default route.

Within the Index() action method, the code starts by creating two ViewBag properties named ActiveConf and ActiveDiv. Then, it stores the activeConf and activeDiv parameters in these properties. This is how the controller transfers the selected conference and division IDs to the view.

After storing the selected IDs, the code retrieves all the conferences and divisions from the database and stores that data in two lists. To do that, this code uses the Conferences and Division properties of the TeamContext object. Then, it adds the value "All" at the beginning of each list. Next, it stores these two lists in two more ViewBag properties named Conferences and Divisions. This is how the controller transfers these lists to the view.

After storing the conferences and divisions, the code gets the teams from the database. It starts by building a query expression that returns all teams. Then, if the active conference is not "all", it adds a where clause to get the teams in the selected conference. After that, if the active division is not "all", it adds a where clause to get the teams in the selected division.

After building the query, the code calls ToList() to execute the query and retrieve a list of Team objects. Then, it returns the ViewResult object that's created by passing the list of Team objects to the View() method. This is how the controller transfers the list of Team objects to the view.

## The Home controller

```
public class HomeController : Controller
{
    private TeamContext context;

    public HomeController(TeamContext ctx)
    {
        context = ctx;
    }

    public ViewResult Index(string activeConf = "all",
                           string activeDiv = "all")
    {
        // store selected conference and division IDs in view bag
        ViewBag.ActiveConf = activeConf;
        ViewBag.ActiveDiv = activeDiv;

        // get list of conferences and divisions from database
        List<Conference> conferences = context.Conferences.ToList();
        List<Division> divisions = context.Divisions.ToList();

        // insert default "All" value at front of each list
        conferences.Insert(0, new Conference { ConferenceID = "all",
                                              Name = "All" });
        divisions.Insert(0, new Division { DivisionID = "all",
                                         Name = "All" });

        // store lists in view bag
        ViewBag.Conferences = conferences;
        ViewBag.Divisions = divisions;

        // get teams - filter by conference and division
        IQueryables<Team> query = context.Teams;
        if (activeConf != "all")
            query = query.Where(
                t => t.Conference.ConferenceID.ToLower() ==
                      activeConf.ToLower());
        if (activeDiv != "all")
            query = query.Where(
                t => t.Division.DivisionID.ToLower() ==
                      activeDiv.ToLower());

        // pass teams to view as model
        var teams = query.ToList();
        return View(teams);
    }
}
```

## The custom route in the Startup.cs file

```
endpoints.MapControllerRoute(
    name: "custom",
    pattern: "{controller}/{action}/conf/{activeConf}/div/{activeDiv}");

endpoints.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");
```

Figure 8-7 The Home controller and endpoints in the Startup.cs file

## The layout

---

Figure 8-8 presents the code for the layout of the NFL Teams app. This layout is similar to other layouts that you've seen in earlier chapters.

To start, this layout uses Razor code to get the Title property from the ViewBag. It's possible to use the ViewData dictionary instead of the ViewBag to get the same data. However, since using the ViewBag yields code that's shorter and cleaner, this layout uses ViewBag, not ViewData.

Within the <body> element, the view uses Bootstrap classes to style the header that appears at the top of every page. This creates a header that says "NFL Teams" and is centered with white text, a blue background, some extra margin on top, and some extra padding on all four sides.

## The layout

```
<!DOCTYPE html>

<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <link href="~/lib/bootstrap/dist/css/bootstrap.min.css" rel="stylesheet" />
    <link href="~/css/site.css" rel="stylesheet" />
    <title>@ViewBag.Title</title>
</head>
<body>
    <div class="container">
        <header class="text-center text-white">
            <h1 class="bg-primary mt-3 p-3">NFL Teams</h1>
        </header>
        <main>
            @RenderBody()
        </main>
    </div>
</body>
</html>
```

---

Figure 8-8 The layout

## The Home/Index view

---

Figure 8-9 presents the code for the Home/Index view of the NFL Teams app. This view begins with a directive indicating that the model for this view is a collection of Team objects. This works because when the Index() action method calls the View() method, it passes a collection of Team objects to this view.

The Razor code block starts with a statement that assigns the value “NFL Teams” to the ViewData dictionary with a key of “Title”. This is the same value that the view’s layout retrieves from the ViewBag for the <title> element. Typically, if the layout uses the ViewBag, it would make sense to use the ViewBag in the view too, just for consistency. However, using the ViewBag in the layout and the ViewData dictionary in the view shows that you can use these properties interchangeably if you want.

The Razor code block continues by defining a helper method named Active() that returns a string. This method accepts two strings and compares them. If they match, the method returns the string “active”. Otherwise, it returns an empty string.

The HTML in this view contains a Bootstrap row with two columns. The first column displays the Conference and Division sections on the left side of the page.

The Conference section starts with an <h3> element that displays the title of the section. Then, it has a <div> element styled as a Bootstrap list group. Within the <div> element, a Razor foreach statement loops through all the Conference objects in the ViewBag and builds a link for each one. Here, the asp-route-activeConf value changes for each Conference object, but the asp-route-activeDiv value is set to the active division from the ViewBag.

In addition, this loop determines whether the link is the active link by calling the Active() method and passing it the ID of the current Conference object in the loop and the active conference ID from the ViewBag. If the current conference ID matches the active conference ID, this code adds the active class. Otherwise, it doesn’t add any classes.

The Division section works much like the Conference section. This time, of course, it loops through the Division objects in the ViewBag. However, it also uses the IDs of the active division and conference to build the links and determine the active division.

The second column contains the team logos in a <ul> element. This element uses the Bootstrap list-inline class so the logos display side by side. Within the <ul> element, a Razor foreach statement loops through all the Team objects in the model and builds an <li> element that contains an <img> element for each one. This works because the images folder of the wwwroot folder contains image files that correspond to the filename stored in the LogoImage property of the Team object.

## The Home/Index view

```
@model IEnumerable<Team>
@{
    ViewData["Title"] = "NFL Teams";
    string Active(string filter, string selected)
    {
        return (filter.ToLower() == selected.ToLower()) ? "active" : "";
    }
}


<div class="col-sm-3">
        <h3 class="mt-3">Conference</h3>
        <div class="list-group">
            @foreach (Conference conf in ViewBag.Conferences)
            {
                <a asp-action="Index"
                    asp-route-activeConf="@conf.ConferenceID"
                    asp-route-activeDiv="@ViewBag.ActiveDiv"
                    class="list-group-item @Active(conf.ConferenceID,
                        ViewBag.ActiveConf)">
                    @conf.Name
                </a>
            }
        </div>
        <h3 class="mt-3">Division</h3>
        <div class="list-group">
            @foreach (Division div in ViewBag.Divisions)
            {
                <a asp-action="Index"
                    asp-route-activeConf="@ViewBag.ActiveConf"
                    asp-route-activeDiv="@div.DivisionID"
                    class="list-group-item @Active(div.DivisionID,
                        ViewBag.ActiveDiv)">
                    @div.Name
                </a>
            }
        </div>
    </div>
    <div class="col-sm-9">
        <ul class="list-inline">
            @foreach (Team team in Model)
            {
                <li class="list-inline-item">
                    <img src("~/images/@team.LogoImage" alt="@team.Name"
                        title="@team.Name |
                            @team.Conference.Name @team.Division.Name" />
                </li>
            }
        </ul>
    </div>
</div>


```

---

Figure 8-9 The Home/Index view

## How to work with view models

The applications you've seen so far have used the View() method to transfer a single entity object or a collection of entity objects to a view. Sometimes, however, the data needed by a view doesn't match the data in an entity model.

For example, the model object for the Index view of the NFL Teams app is a collection of Team objects. However, that view also needs a collection of Conference objects and a collection of Division objects to build the links to filter by conference and division. In addition, it needs the IDs of the active conference and division.

In the NFL Teams 1.0 app, the controller used the ViewBag to transfer much of the data to the view. For this data, IntelliSense doesn't work, and the compiler doesn't warn you of problems in your code. Instead, you have to wait until runtime to discover these problems. That's why it's generally considered a best practice to use a view model to transfer data to a view.

### How to create a view model

A *view model* is a regular C# class that holds all of the data that a specific view requires. By convention, this kind of class ends with the suffix "ViewModel", but that isn't a requirement.

The code example in figure 8-10 presents the TeamListViewModel class. The first property in this class is the collection of Team objects that the view needs. The next two properties are the ID values of the active conference and division.

The last two properties are the collections of Conference and Division objects the view needs. However, these properties aren't auto-implemented properties like the first three. Rather, they are standard properties with private backing fields. This is necessary so the setter for each property can add the value "All" at the beginning of the collection. This works much like the code in the Home controller presented in figure 8-7.

Typically, view models are lightweight classes that contain only data. However, they can also contain simple helper methods for the view. For instance, the TeamListViewModel class provide two methods named CheckActiveConf() and CheckActiveDiv() that the view can use to determine the active link in the conference and division sections. That's acceptable because this logic only impacts how the view is displayed. However, a view model shouldn't have logic that impacts other parts of the app such as the entity model or controller.

## A view model for the NFL Teams app

```
public class TeamListViewModel
{
    public List<Team> Teams { get; set; }
    public string ActiveConf { get; set; }
    public string ActiveDiv { get; set; }

    // make next two properties standard properties so the setter
    // can make the first item in each list "All"
    private List<Conference> conferences;
    public List<Conference> Conferences {
        get => conferences;
        set {
            conferences = value;
            conferences.Insert(0,
                new Conference { ConferenceID = "all", Name = "All" });
        }
    }

    private List<Division> divisions;
    public List<Division> Divisions {
        get => divisions;
        set {
            divisions = value;
            divisions.Insert(0,
                new Division { DivisionID = "all", Name = "All" });
        }
    }

    // methods to help view determine active link
    public string CheckActiveConf(string c) =>
        c.ToLower() == ActiveConf.ToLower() ? "active" : "";

    public string CheckActiveDiv(string d) =>
        d.ToLower() == ActiveDiv.ToLower() ? "active" : "";
}
```

### Description

- A *view model* is a regular C# class that defines a model of the data that's needed by a view.
- By convention, the name of a view model class ends with a suffix of “ViewModel”, but this isn't required.
- Most view models only provide data. However, a view model can also contain simple methods that help the view display that data.

---

Figure 8-10 How to create a view model

## The updated Index() action method

---

Now that you have a view model, you can update the NFL Teams app to use it. The first code example in figure 8-11 shows the updated Index() action method of the Home controller.

This action method starts by initializing a new TeamListViewModel object with the active conference and division IDs as well as collections of Conference and Division objects from the database. Since the code that adds “All” to the beginning of each list has been moved to the setters, this controller code is cleaner and easier to read. That’s one of the benefits of a view model class. Another benefit is that IntelliSense is available when you work with the TeamListViewModel object. That’s not true when you work with the ViewBag or ViewData properties.

After setting the first four properties, the action method builds a query expression like it did earlier in this chapter. Now, though, when the code executes the query, it stores the collection of teams from the database in the Teams property of the TeamListViewModel object. Then, it uses the View() method to pass the view model to the view.

## The updated Home/Index view

---

The second code example in this figure presents the updated code for the Home/Index view. The statement at the top of the Index view now indicates that the model for this view is a TeamListViewModel object.

In this view, the Razor code block only contains the ViewData code that sets the <title> element in the layout. In other words, this code doesn’t need the helper function named Active() that was presented earlier in this chapter.

The HTML in this updated view works much like it did in the previous version. The main difference is that you use the properties of the view model rather than the properties of the ViewBag. This may not seem like a big difference, but using a view model like this provides compile-time error checking. As a result, if you make a typo in the name of a view model property, the IDE will warn you before you run the app. With ViewBag properties, you won’t discover this kind of error until you actually run the app.

Another difference is that this view uses the CheckActiveConf() and CheckActiveDiv() helper methods of the TeamListViewModel class to mark the correct link as the active link. This code is shorter and cleaner than the code presented earlier in this chapter.

Finally, the Razor foreach statement that loops through the list of Team objects transferred from the controller now gets those objects from the Teams property of the view model. However, the HTML and Razor code that builds the logo images is the same as in figure 8-9.

## The updated Index() action method of the Home controller

```
public ViewResult Index(string activeConf = "all", string activeDiv = "all")
{
    var model = new TeamListViewModel {
        ActiveConf = activeConf,
        ActiveDiv = activeDiv,
        Conferences = context.Conferences.ToList(),
        Divisions = context.Divisions.ToList()
    };
    IQueryable<Team> query = context.Teams;
    if (activeConf != "all")
        query = query.Where(t =>
            t.Conference.ConferenceID.ToLower() == activeConf.ToLower());
    if (activeDiv != "all")
        query = query.Where(t =>
            t.Division.DivisionID.ToLower() == activeDiv.ToLower());
    model.Teams = query.ToList();
    return View(model);
}
```

## The updated Home/Index view

```
@model TeamListViewModel
 @{
    ViewData["Title"] = "NFL Teams"; // helper function no longer needed
}
<div class="row">
    <div class="col-sm-3">
        <h3>Conference</h3>
        <div class="list-group">
            @foreach (Conference conf in Model.Conferences) {
                <a asp-action="Index"
                    asp-route-conference="@conf.ConferenceID"
                    asp-route-division="@Model.ActiveDiv"
                    class="list-group-item"
                    @Model.CheckActiveConf(conf.ConferenceID)">@conf.Name</a>
            }
        </div>
        <h3>Division</h3>
        <div class="list-group">
            @foreach (Division div in Model.Divisions) {
                <a asp-action="Index"
                    asp-route-conference="@Model.ActiveConf"
                    asp-route-division="@div.DivisionID"
                    class="list-group-item"
                    @Model.CheckActiveDiv(div.DivisionID)">@div.Name</a>
            }
        </div>
    </div>
    <div class="col-sm-9">
        <ul class="list-inline">
            @foreach (Team team in Model.Teams) {
                <!-- same as figure 8-9 -->
            }
        </ul>
    </div>
</div>
```

Figure 8-11 The updated Index() action method and Home/Index view

## How to redirect a request

In addition to using a view to send a web page to the browser, you can use some of the subtypes of the ActionResult class to redirect an HTTP request to another URL. You'll learn how to do that in the topics that follow.

### How to use the ActionResult classes for redirection

When a browser makes a request to a server, the server returns an HTTP status code along with the response. For instance, when a server returns HTML, it also returns a 200 OK status code. Or, when a browser requests a URL that doesn't exist, the server returns a 404 Not Found status code.

The first table in figure 8-12 presents two of the HTTP status codes for redirection. Both the 302 Found and the 301 Moved Permanently status codes direct the browser to make a GET request to another URL. The difference is that the 301 Moved Permanently status code tells the browser that this move is permanent and all future requests should be directed to the other URL. Most browsers cache this response and never request the original URL again.

The second table presents the ActionResult subtypes you can use for redirection. The Controller class provides 302 Found and 301 Moved Permanently versions of methods that return these subtypes. Both versions perform the redirection, but the permanent version returns a 301 status code. Typically, you use the 301 status code to support old URLs in your app.

The third table provides guidance on when to use each subtype. To navigate within your app, it's common to use RedirectToActionResult. That's because the methods that create it build the URL based on your app routes. To navigate based on route names, you can use RedirectToRouteResult. However, this is less common, and some developers consider it a bad practice.

You can use LocalRedirectResult to pass a URL that the user should return to later. For example, they might return to the URL after they log in. This ActionResult subtype makes sure that the return URL is part of your app. Thus, it prevents *open redirection attacks* that attempt to redirect to a malicious external site.

Most of the methods that return ActionResult objects for redirection have several overloads. The fourth table illustrates this by presenting some of the overloads for the RedirectToAction() method. These overloads require you to specify the action method you want to redirect to, and allow you to optionally specify the controller and any route parameters.

The code below the last table shows examples of the overloads of the RedirectToAction() method. To pass a route parameter to a method, you can code an anonymous object with a property that specifies the name and value of a route segment. If the parameter name in the method is the same as the route segment name, you can use a shortcut as shown in the fourth example. Or, if you prefer, you can use a string-string dictionary to pass route parameters as shown in the last example.

## Two of the HTTP status codes for redirection

Code	Description
302 Found	Directs the browser to make a GET request to another URL.
301 Moved Permanently	Directs the browser to make a GET request to another URL for this and all future requests.

## The ActionResult subtypes for redirection

Subtype	302 Found method	301 Moved Permanently method
RedirectResult	Redirect()	RedirectPermanent()
LocalRedirectResult	LocalRedirect()	LocalRedirectPermanent()
RedirectToActionResult	RedirectToAction()	RedirectToActionPermanent()
RedirectToRouteResult	RedirectToRoute()	RedirectToRoutePermanent()

## How to know which subtype to use for redirection

Subtype	Use when...
RedirectResult	Redirecting to an external URL, such as <a href="https://google.com">https://google.com</a> .
LocalRedirectResult	Making sure you redirect to a URL within the current app.
RedirectToActionResult	Redirecting to an action method within the current app.
RedirectToRouteResult	Redirecting within the current app by using a named route.

## Some of the overloads available for the RedirectToAction() method

Arguments	Redirect to...
(a)	The specified action method in the current controller.
(a, c)	The specified action method in the specified controller.
(a, routes)	The specified action method in the current controller with route parameters.
(a, c, routes)	The specified action method in the specified controller with route parameters.

## Code that redirects to another action method

### Redirect to the List() action method in the current controller

```
public RedirectToActionResult Index() => RedirectToAction("List");
```

### Redirect to the List() action method in the Team controller

```
public RedirectToActionResult Index() => RedirectToAction("List", "Team");
```

### Redirect to the Details() action method in the current controller with a parameter

```
public RedirectToActionResult Index(string id) =>
    RedirectToAction("Details", new { ID = id });
```

### Use a shortcut when variable name and route segment name match

```
public RedirectToActionResult Index(string id) =>
    RedirectToAction("Details", new { id });
```

### Use a string-string dictionary to supply a parameter

```
public RedirectToActionResult Index(string id) =>
    RedirectToAction("Details",
        new Dictionary<string, string>(){ { "ID", id } } );
```

Figure 8-12 How to use the ActionResult classes for redirection

## How to use the Post-Redirect-Get pattern

GET requests are designed to retrieve data but be *idempotent*, which means that they make no changes on the server. Because of that, a user can resubmit a GET request multiple times by clicking the browser's Refresh button, and it doesn't affect any data on the server.

POST requests, by contrast, are designed to post, or write, data to the server. Typically, you don't want your users to resubmit a POST request. For example, if a POST request submits an order to the server, you don't want the user to accidentally resubmit that order by clicking the browser's Refresh button.

To help prevent that, most browsers display a message like the one shown in figure 8-13 if a user resubmits a POST request. However, if the user chooses to continue despite the browser warning, the user can submit the POST request again, which may not be what you want.

To prevent the browser from displaying the form resubmission message, it's common to use the *Post-Redirect-Get (PRG) pattern* when you're making changes on the server. In this figure, the code example shows how the Movie List app presented in chapter 4 implements the PRG pattern. Here, the Delete() action method handles the POST request that deletes a movie from the database. The first statement in this method removes the specified movie from the DB context, and the second statement saves the change to the database.

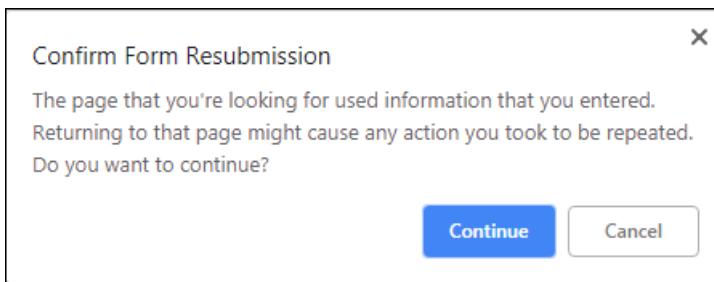
However, the third statement uses the RedirectToAction() method to return a 302 Found status code to the browser. This tells the browser to issue a GET request for the Index() action method of the Home controller, which displays a list of movies in the database. As a result, if the user clicks the Refresh button, it displays the list of movies in the database again. Since the Index() action method doesn't change any data on the server, requesting it is an idempotent request.

The downside of the PRG pattern is that the redirection requires a second roundtrip to the server. However, the upside of avoiding problems that can be caused by resubmitting a form typically outweighs the downside.

If you have a POST action method that makes changes on the server and then returns a view, you should consider using the PRG pattern to split it into two action methods. That way, you can use a POST action method to make changes to the data on the server. Then, you can redirect to a GET action method to display a view. That way, the GET action method is idempotent and can be run one or more times without changing the data that's stored on the server.

For example, you wouldn't want a POST action method to delete a movie from the server and then return a view. Instead, you would want to use the PRG pattern to divide the processing into two action methods as shown in this figure. That way, the POST action method can delete the movie and redirect to an idempotent GET action method that returns a view.

## A browser message that's displayed when you refresh a page that's displayed by a POST request



## Two action methods that uses the Post-Redirect-Get (PRG) pattern

### A Delete() action method for a POST request

```
[HttpPost] // Post
public IActionResult Delete(Movie movie)
{
    context.Movies.Remove(movie);
    context.SaveChanges();
    return RedirectToAction("Index", "Home"); // Redirect
}
```

### The Index() action method of the Home controller for a GET request

```
[HttpGet] // Get
public IActionResult Index()
{
    var movies = context.Movies
        .Include(m => m.Genre)
        .OrderBy(m => m.Name)
        .ToList();
    return View(movies);
}
```

## Description

- To prevent resubmission of POST data, you can use the *Post-Redirect-Get (PRG) pattern*. With this pattern, a POST action writes data to the server and then redirects to a GET action to read data from the server.

---

Figure 8-13 How to use the Post-Redirect-Get (PRG) pattern

## How to use the TempData property

So far, this chapter has shown how to transfer data from a controller to a view. Sometimes, though, you need to transfer data from a controller to another controller. To do that, you can use the built-in TempData property that's available from controllers and views.

### How to get started with TempData

The TempData property of the Controller class works much like the ViewData property described earlier in this chapter. To start, it uses a dictionary to store a collection of key/value pairs where the key is a string and the value is an object. As a result, if you want to work with a value in the TempData dictionary, you must cast it to the appropriate data type. But first, you should check the value to make it isn't null.

The TempData property differs from the ViewData property in how long its data persists. With ViewData, the data persists only for the life of the current request. That is, once the server sends its response to the browser, all the data in the ViewData (and ViewBag) property is lost. With TempData, on the other hand, the data persists across multiple requests. In fact, this data persists until it is read. Once a TempData item is read, it's marked as read. Then, at the end of each request, every item that's marked as read is removed from the TempData dictionary.

Because the items in TempData persist beyond the current request, TempData is often used in conjunction with the PRG pattern. The code in figure 8-14 illustrates how this works. Here, an action method named Delete() handles a POST request that deletes a movie from the database.

Within the action method for the POST request, the first two statements delete the specified movie from the database. Then, the third statement sets the value of an item with a key of "message" in the TempData property. Finally, the fourth statement causes the browser to make a GET request that displays a list of movies. As a result, the POST action method can't use the ViewData property to transfer the message to the GET action method. However, the POST action method can use the TempData property because its data persists across requests.

The view shown in this figure uses a Razor if statement to check whether the TempData property contains an item with a key of "message". That's possible because TempData is a dictionary. As a result, it has normal dictionary properties like Keys and Values that have a Contains() method. If this TempData property contains the item, the view uses an `<h4>` element to display its value. This element is styled with Bootstrap CSS classes so that it appears as shown in the figure. For this to work properly, the view that calls the Delete() action method must pass the movie's ID and name as arguments. That way, the message can include the movie's name.

By default, TempData can only store data that can be serialized, such as primitive types like int or string. However, the next chapter presents a technique that allows you to store more complex data types in TempData.

## An action method that uses TempData with the Post-Redirect-Get pattern

```
[HttpPost]
public IActionResult Delete(Movie movie)
{
    context.Movies.Remove(movie);
    context.SaveChanges();
    TempData["message"] = $"{movie.Name} deleted from database.";
    return RedirectToAction("Index", "Home");
}
```

## Code that reads a TempData value in a Layout view

```
...
<header class="jumbotron">
    <h1>My Movies</h1>
</header>...
@if (TempData.Keys.Contains("message"))
{
    <h4 class="bg-info text-center text-white p-2">
        @TempData["message"]
    </h4>
}
...

```

## The temporary message displayed in a browser



## Description

- The Controller class has a property named TempData that lets you transfer data to another controller or view.
- Data in TempData persists across multiple requests until it is read. By contrast, data in ViewData and ViewBag only persists until the end of the current request.
- TempData is often used with the PRG pattern because that pattern takes place across two requests (the POST request and the subsequent GET request).
- TempData can only store data that can be serialized such as primitive types.
- Because TempData is a dictionary, it has normal dictionary properties like Keys and Values, which in turn have a Contains() method you can use to check for values.
- By default, ASP.NET Core 3.0 and later automatically enable TempData when you call the AddControllersWithViews() method in the Configure() method of the Startup.cs file.

---

Figure 8-14 How to get started with TempData

## How to use methods of the TempData dictionary

---

When you read an item from the TempData dictionary normally, it is marked as read. Then, at the end of the current request, any items marked as read are removed. Sometimes, though, you may want an item to persist even after you've read it. Or, you may want to peek at the value but still have it persist. In these cases, you can use the Keep() and Peek() methods of the TempDataDictionary class as described in figure 8-15.

The Keep() method has two overloads. If you call it with no arguments, it marks all the items in TempData as unread, even if they have been read. As a result, none of the items are removed at the end of the request. If you pass a key to this method, it marks the value for that key as unread, even if it has been read. Again, this means that the value won't be removed at the end of the request.

The Peek() method accepts a key and returns the value for that key. However, it doesn't mark that item as read. Thus, it remains in the TempData dictionary.

Generally, you use the Peek() method when you know that you don't want to mark the value as read. However, if you might want to mark the value as read, you can read the value normally. Then, you can call the Keep() method if the condition for keeping that value is met.

In the code example, the methods use a view model named TeamViewModel. This view model works much like the TeamListViewModel presented earlier in this chapter. In short, it uses ActiveConf and ActiveDiv properties to store the active conference and division IDs, and it uses a Team property to store the selected Team object.

The POST request for the Details() action method assumes that the TeamViewModel parameter includes all of the data needed to view the details for the specified team (the active conference and division IDs and the specified team ID). To start, the POST action method calls a static method from a class named Utility to write the team's ID to the server. Then, this action method needs to make the request idempotent by passing that data to the GET request for the Details() method. To do that, it stores the active division and conference IDs in the TempData property. Then, it uses the RedirectToAction() method to pass the ID for the team to the GET request.

The GET request builds a TeamViewModel from this data. To do that, it uses the team ID to get the correct Team object from the database and set the Team property. Then, it uses the TempData property to get the active division and conference IDs and set the ActiveConf and ActiveDiv properties. As it does this, it uses the Peek() method to read the ActiveConf and ActiveDiv keys of the TempData property without marking them as read. That way, these properties are available to the view, even if the user resubmits the GET request one or more times.

If you find that you often need to keep TempData items across multiple requests, you should probably use session state instead. Using session state allows you to keep items until the user closes the browser, and it's described in the next chapter.

## Two methods of the TempDataDictionary class

Method	Description
<code>Keep()</code>	Marks all the values in the dictionary as unread, even if they've already been read.
<code>Keep(key)</code>	Marks the value associated with the specified key as unread, even if it has already been read.
<code>Peek(key)</code>	Reads the value associated with the specified key but does not mark it as read.

## The overloaded Details() action method of the Home controller

```
[HttpPost]
public RedirectToActionResult Details(TeamViewModel model)
{
    Utility.LogTeamClick(model.Team.TeamID);
    TempData["ActiveConf"] = model.ActiveConf;
    TempData["ActiveDiv"] = model.ActiveDiv;
    return RedirectToAction("Details", new { ID = model.Team.TeamID });
}

[HttpGet]
public ViewResult Details(string id)
{
    var model = new TeamViewModel {
        Team = context.Teams
            .Include(t => t.Conference)
            .Include(t => t.Division)
            .FirstOrDefault(t => t.TeamID == id),
        ActiveConf = TempData.Peek("ActiveConf").ToString(),
        ActiveDiv = TempData.Peek("ActiveDiv").ToString()
    };
    return View(model);
}
```

## When to use the Keep() and Peek() methods

- Use Peek() when you know you want the value to stay marked as unread.
- Use a normal read and Keep() when you want to use a condition to determine whether to mark the value as unread.

## Description

- With a normal read, an item from TempData is marked for deletion and deleted at the end of the current request. However, the TempDataDictionary class provides some methods to keep an item in TempData even after it's been read.
- If you consistently need to keep items in the TempData dictionary, you should consider storing the item in a session instead as described in the next chapter.
- When you redirect an HTTP request, MVC automatically calls the Keep() method under the covers.

---

Figure 8-15 How to use methods of the TempData dictionary

## The NFL Teams 2.0 app

---

The next few figures present an updated version of the NFL Teams app. This version of the app illustrates some of the skills for working with redirection, the PRG pattern, and TempData.

### The user interface

---

The first screen in figure 8-16 shows the updated NFL Teams app after the user selects a conference and division. This looks no different than the version you saw in figure 8-5. However, in this version of the app, the team logo is a clickable image button rather than just an image. In addition, clicking on this image button starts a POST request rather than a GET request. That way, the app can keep a log of how many users click on each team logo. Since this action changes something on the server (that is, it updates a log count), it's not an idempotent request. Thus, it should be handled by a POST request.

After the user clicks on a team logo and a POST request logs the click, the code uses the PRG pattern to redirect the browser to a Details page for the selected team as shown by the second screen. Since the GET request that produces this page is idempotent, users can refresh this page as many times as they like without causing any extra logging or browser warnings.

Before the app redirects from the POST action, it stores the IDs of the active conference and division in TempData. Then, the GET action method retrieves this data and uses it to build the “Return to Home Page” link. That way, the app can “remember” the conference and division the user selected without having to store them in the URL. Then, when the user returns to the Home page, the app displays the previously selected conference and division. That’s why the first and third screens in this figure display the same logos.

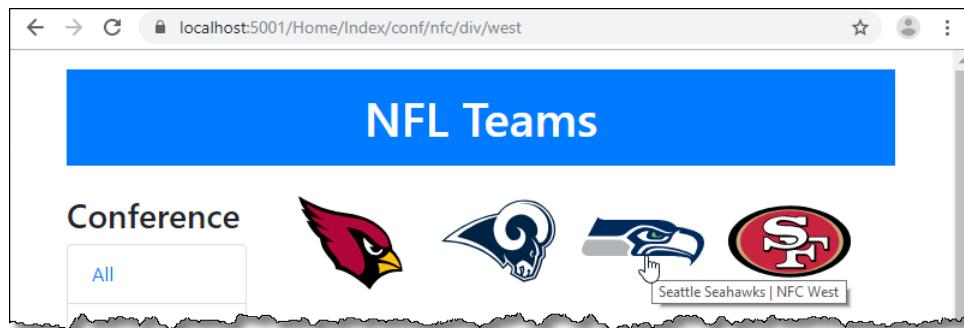
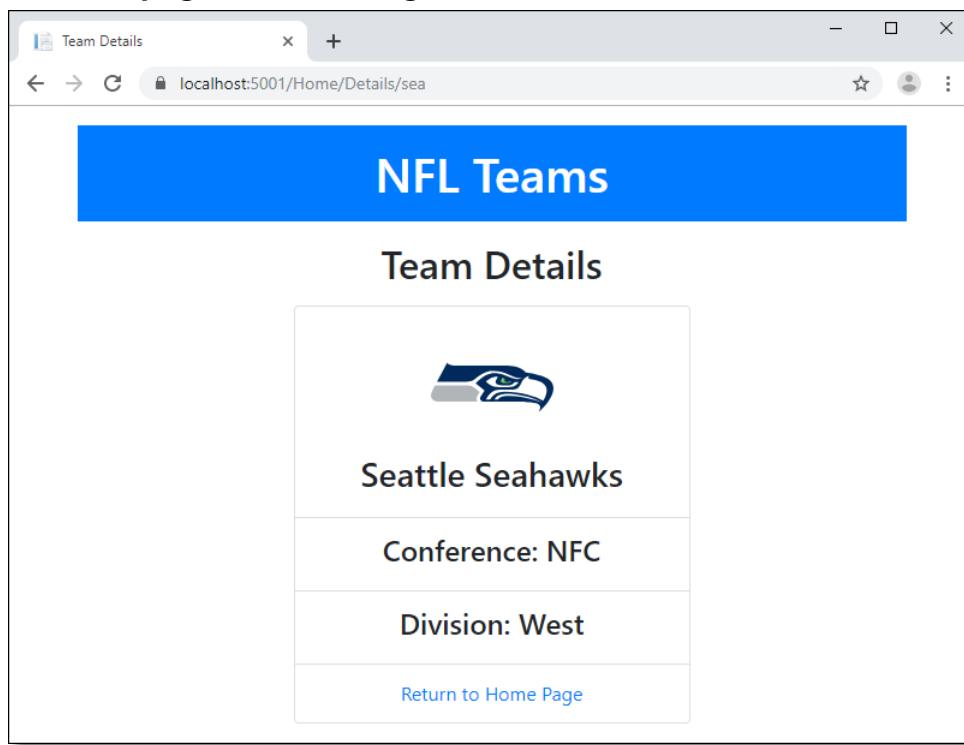
**The Home page after selecting a conference and division****The Details page after clicking on a team****The Home page after clicking on the "Return to Home Page" link**

Figure 8-16 The user interface of the NFL Teams 2.0 app

## The view model classes

---

Earlier in this chapter you learned how to add a view model class to the NFL Teams app. The purpose of this class is to provide everything the Home view needs to display the team logos.

The updated version of the NFL Teams app adds a Details view that displays details about a single team. As a result, the updated app adds a new view model for this view.

Figure 8-17 starts by presenting the code for the new TeamViewModel class. This is the view model that the new Details() action method uses to pass data from the controller to the new Details view.

Since the TeamViewModel and TeamListViewModel classes both share string properties named ActiveConf and ActiveDiv, it makes sense for the TeamListViewModel class to inherit the TeamViewModel class as shown by the second example. This prevents some code duplication, which can make your code easier to maintain.

## The Details() action method

---

The third code example in figure 8-17 shows the overloaded Details() action method. To start, the POST action method calls a utility method named LogTeamClicked() to keep track of the number of times the button for a team is clicked. Then, the POST action method needs to transfer the active conference and division IDs to the Details() action method that handles GET requests. To do that, the POST action method stores this information in the TempData property. That way, the data persists past the current request.

This POST action method doesn't add the TeamViewModel object itself to the TempData dictionary. That's because TempData can't hold a complex data type like a view model. Instead, it stores two strings as individual TempData items.

The POST action method finishes by calling the RedirectToAction() method and passing it two arguments. For the first argument, it passes the name of the action method to redirect to. For the second argument, it passes the ID of the currently selected team. Thus, the browser redirects to the Details() action method in the current controller and includes the ID of the clicked team as a route parameter.

The Details() action method that handles GET requests starts by creating a TeamViewModel object for the current team. To do that, it uses the FirstOrDefault() method to execute a query that gets the team with the ID that's passed to the method. This query also uses the Include() method to load information about the team's conference and division into the context so it can be displayed by the view. In addition to the team, the active conference and division IDs are retrieved from the TempData dictionary and included in the TeamViewModel object. Or, if the values for these IDs are null, it sets the active conference and division IDs to a default value of "all". Finally, it passes the view model object to the View() method and returns the resulting object.

## The TeamViewModel class

```
public class TeamViewModel
{
    public Team Team { get; set; }
    public string ActiveConf { get; set; } = "all";
    public string ActiveDiv { get; set; } = "all";
}
```

## The updated TeamListViewModel class

```
public class TeamListViewModel : TeamViewModel
{
    // Team, ActiveConf, and ActiveDiv properties are inherited
    // Teams, Conferences, and Divisions properties same as figure 8-10
    // CheckActiveConf() and CheckActiveDiv() methods same as figure 8-10
}
```

## The Details() action method of the Home controller

```
[HttpPost]
public RedirectToActionResult Details(TeamViewModel model)
{
    Utility.LogTeamClick(model.Team.TeamID);

    TempData["ActiveConf"] = model.ActiveConf;
    TempData["ActiveDiv"] = model.ActiveDiv;
    return RedirectToAction("Details", new { ID = model.Team.TeamID });
}

[HttpGet]
public ViewResult Details(string id)
{
    var model = new TeamViewModel
    {
        Team = context.Teams
            .Include(t => t.Conference)
            .Include(t => t.Division)
            .FirstOrDefault(t => t.TeamID == id),
        ActiveDiv = TempData?["ActiveDiv"]?.ToString() ?? "all",
        ActiveConf = TempData?["ActiveConf"]?.ToString() ?? "all"
    };
    return View(model);
}
```

## Description

- This version of the NFL Teams app uses two view models named TeamViewModel and TeamListViewModel.
- The TeamListViewModel inherits the TeamViewModel. That way, the ActiveConf and ActiveDiv properties are only coded in the TeamViewModel class.
- Since the overloaded Details() action method uses the PRG pattern, it uses the TempData dictionary to transfer data from the action method that handles POST requests to the action method that handles GET requests.

---

Figure 8-17 The view model classes and the Details() action method

## The Home/Index view

---

Figure 8-18 presents the code for the Home/Index view of the updated NFL Teams app. Most of this code works the same as the Home/Index view presented in figure 8-11. However, in the Razor loop that displays the team logos, the code places the `<img>` element within a `<button>` element to create an image button. In addition, it places both of these elements within a `<form>` element. In other words, it places each image button inside its own form.

After creating the image buttons, the Razor code adds three hidden fields. The first is bound to the `TeamID` property of the current `Team` property in the loop. The next two contain the IDs of the active conference and division. Together, these three hidden fields make sure that the IDs for the selected team, conference, and division are posted to the server.

To review, this foreach loop generates HTML that contains multiple `<form>` elements, each of which posts to the `Details()` action method of the current controller. Each form consists of a submit button that displays a team logo, and three hidden fields that post the IDs of the selected team, conference, and division.

Note that the model for the Index view is a `TeamListViewModel` object, but the `Details()` action method for POST requests accepts a `TeamViewModel` object. The Index view can post data to that action method successfully because the `TeamListViewModel` class inherits the `TeamViewModel` class. As a result, the `TeamViewModel` object in the `Details()` action method contains the same properties that are bound to these three hidden fields.

Each `<form>` element generated by this view contains two hidden fields (`ActiveConf` and `ActiveDiv`) whose values are the same as the hidden fields of all the other `<form>` elements. Since this app only uses 32 `<form>` elements (one for each team), this duplication shouldn't cause any problems. However, if this app needed to send hundreds or thousands of `<form>` elements to the browser, it might cause your page to render more slowly. Plus, it's generally considered a best practice to avoid duplication whenever possible. That's why the next two chapters present some ways to avoid duplicating these hidden fields.

When you create image buttons, the browser might format them differently than regular images. For example, it might add a border around the image. To prevent that, you can add Bootstrap classes to the `<button>` element like the ones shown in this figure to override the default formatting.

## The Home/Index view

```
@model TeamListViewModel  
{  
    ViewData["Title"] = "NFL Teams";  
  
    <div class="row">  
        <div class="col-sm-3">  
            <!-- Conference and Division ul elements same as figure 8-11 -->  
        </div>  
        <div class="col-sm-9">  
            <ul class="list-inline">  
                @foreach (Team team in Model.Teams)  
                {  
                    <li class="list-inline-item">  
                        <form asp-action="Details" method="post">  
                            <button class="bg-white border-0" type="submit">  
                                  
                            </button>  
  
                            <input type="hidden" asp-for="@team.TeamID" />  
                            <input type="hidden" asp-for="ActiveConf" />  
                            <input type="hidden" asp-for="ActiveDiv" />  
                        </form>  
                    </li>  
                }  
            </ul>  
        </div>  
    </div>
```

### Description

- The Home/Index view receives a TeamListViewModel object from the Index() action method of the Home controller.
- The <form> element for each team binds three hidden fields to the ActiveConf, ActiveDiv, and Team.TeamID properties of the view model.

---

Figure 8-18 The Home/Index view

## The Home/Details view

---

Figure 8-19 presents the code for the Home/Details view of the NFL Teams app. This view begins with a directive indicating that this view uses the TeamViewModel object as its model. Remember, that's what the Details() action method for GET requests transfers to this view.

The Details view uses the data in the Team property of the view model to display information about the selected team. Then, it uses the ActiveConf and ActiveDiv properties of the view model to construct a link the user can click to return to the Home page.

As you recall, the ActiveConf and ActiveDiv properties contain data that was posted to the Details() action method, redirected to the GET action method, and transferred from the GET action method to this view. As a result, when the user clicks on the “Return to Home Page” link, the app “remembers” the conference and division that the user selected before viewing the Details page. For instance, if a user had selected the NFC conference and the West division and then clicked on the Seahawks logo, the page still filters by NFC and West when the user returns to the Home page.

Since the conference and division information is transferred via the TempData dictionary, it's removed once it's read. That is, once this link is built and the HTML is rendered, the request is completed and MVC removes the TempData properties that have been read. As a result, if the user refreshes this page, the link “forgets” the conference and division that the user selected before viewing this page. If that isn't how you want this page to work, you can use the Peek() method to read from TempData as described in figure 8-15. That way, this data isn't marked as read. Or, you can use a session to store the active conference and division IDs as described in the next chapter.

## The Home/Details view

```
@model TeamViewModel
{
    ViewData["Title"] = "Team Details";
}

<h2>Team Details</h2>

<div class="row">
    <div class="col-sm-6 col-sm-offset-3">
        <ul class="list-group text-center">
            <li class="list-group-item">
                <img src("~/images/@Model.Team.LogoImage" alt="" />
                <h3>@Model.Team.Name</h3>
            </li>
            <li class="list-group-item">
                <h4>Conference: @Model.Team.Conference.Name</h4>
            </li>
            <li class="list-group-item">
                <h4>Division: @Model.Team.Division.Name</h4>
            </li>
            <li class="list-group-item">
                <a asp-action="Index"
                    asp-route-conference="@Model.ActiveConf"
                    asp-route-division="@Model.ActiveDiv">
                    Return to Home Page
                </a>
            </li>
        </ul>
    </div>
</div>
```

### Description

- The Details view receives a TeamViewModel object from the Details() action method of the Home controller.
- The Razor code uses the Team property of the view model to display details about the selected team.
- For the link that sends the user back to the Home page, the Razor code uses the ActiveConf and ActiveDiv properties of the view model to create a URL with the previously selected conference and division.

---

Figure 8-19 The Home/Details view

## Perspective

---

The goal of this chapter is to show how to transfer data from a controller to a view or to another controller. To do that, this chapter reviewed some skills that were presented in chapter 2 such as using the `ViewBag` property. In addition, it presented several new ways to transfer data from a controller to a view including using the `ViewData` property or a view model. Finally, it showed how to transfer data to another controller by using the `TempData` property.

The skills presented in this chapter should provide a solid foundation for transferring data from a controller. However, there are other ways to transfer data within your app, and the next two chapters present the most useful techniques. To start, the next chapter shows how to use sessions and cookies to maintain state in an app.

## Terms

---

dynamic type  
alias  
view model

idempotent  
Post-Redirect-Get (PRG) pattern  
open redirection attack

## Summary

---

- `ViewBag` properties use C#'s *dynamic type*. As a result, the data type of each property is set at runtime.
- Under the hood, the `ViewBag` property uses the `ViewData` dictionary to store its dynamic properties. Thus, you can think of the `ViewBag` property as an *alias* for the `ViewData` dictionary.
- A *view model* is a regular C# class that defines a model of the data that's needed by a view.
- An *idempotent* request has the same effect on the server whether it's made once or multiple times.
- To prevent resubmission of POST data, you can use the *Post-Redirect-Get (PRG) pattern*. With this pattern, a POST action writes data to the server and then redirects to a GET action to read data from the server.
- *Open redirection attacks* attempt to redirect to a malicious external site.

## Exercise 8-1 Update the Guitar Shop app to use a view model and TempData

In this exercise, you'll update the Guitar Shop app presented in chapter 7 so it uses a view model and TempData.

### Run the app and add a Product List view model

1. Open the Ch08Ex1GuitarShop web app in the ex\_starts directory.
2. Run the app and test it. Make sure to test the Product List page in the default area as well as the Product List page in the Admin area.
3. Close the browser and return to Visual Studio.
4. In the Models folder, create a new C# class named ProductListViewModel and enter this code:

```
public class ProductListViewModel
{
    public List<Category> Categories { get; set; }
    public List<Product> Products { get; set; }
    public string SelectedCategory { get; set; }
    public string CheckActiveCategory(string category) =>
        category == SelectedCategory ? "active" : "";
}
```

### Update the Product List controller and view in the default area

5. In the Controllers folder, open the ProductController class. In its List() action method, delete the statements that use the ViewBag.
6. Before the return statement, create and populate the view model like this:

```
var model = new ProductListViewModel
{
    Categories = categories,
    Products = products,
    SelectedCategory = id
};
```

7. Modify the return statement so it passes the view model to the View() method.
8. Open the Product/ List view and change its model from List<Product> to ProductListViewModel.
9. Before the first loop, modify the code that sets the active class for the All category so it uses the CheckActiveCategory() method of the view model.
10. Modify the first loop so it gets its categories from the view model, not the ViewBag.
11. Within the first loop, modify the code that sets the active class for each category so it uses the CheckActiveCategory() method of the view model.
12. Modify the second loop so it gets its products from the new view model (a ProductListViewModel object), not the old view model (a List<Product> object).
13. Run the app and test the Product List page in the default area. It should work the same as before.

## Update the Product List controller and view in the Admin area

14. In Visual Studio's Solution Explorer, view the Admin area by expanding the Areas folder and the Admin folder.
15. In the Admin area, open the ProductController class.
16. In the List() action method, just before the return statement, enter code that creates and populates a Product List view model as described earlier in this exercise. Then, modify the return statement to pass the model to the view.
17. In the Admin area, open the Product/List view and change its view model from List<Product> to ProductListViewModel.
18. Modify the Razor if statement so it uses the view model, not the ViewBag, to get the selected category.
19. Modify the first loop so it gets its categories from the view model, not the ViewBag.
20. Within the first loop, use the CheckActiveCategory() method of the view model to determine whether to set the active class for the category.
21. Modify the second loop so it gets its products from the new view model, not the old one.
22. Run the app and test the Product List page for the Admin area. It should work like it did before.

## Use TempData to display a message

23. In the Admin area, open the ProductController class.
24. In the Update() action method for POST requests, modify the code so adding a product stores a message in TempData that says "You just added the product *ProductName*". When you do that, use a key of "UserMessage".
25. Modify the code so updating a product stores a message in TempData that says "You just updated the product *ProductName*".
26. In the Admin area, open the Product/List view and scroll down to the end of it.
27. After the table but before the Add New Product button, add code that displays the message that's stored in TempData like this:

```
@{  
    string userMessage = TempData?["UserMessage"]?.ToString() ?? "";  
}  
@if (userMessage != "") {  
    <div class="text-success">@userMessage</div>  
}
```

28. Run the app and test it. Make sure that the Product List page in the Admin area displays a message after you add or update a product.

# 9

## How to work with session state and cookies

In the previous chapter, you learned some ways to transfer data from a controller to a view or to another controller. These techniques only allow data to persist across one or more requests. However, you sometimes need data to persist for as long as the browser is open or even longer. In this chapter, you'll learn some techniques for doing that.

<b>How ASP.NET MVC handles state.....</b>	<b>322</b>
Six ways to maintain state .....	322
An introduction to session state.....	322
<b>How to work with session state .....</b>	<b>324</b>
How to configure an app to use session state .....	324
How to work with session state items in a controller .....	326
How to get session state values in a view .....	326
How to use JSON to store objects in session state .....	328
How to extend the ISession interface .....	330
How to use a wrapper class .....	332
<b>The NFL Teams 3.0 app .....</b>	<b>334</b>
The user interface .....	334
The session classes.....	336
The Home controller.....	338
The layout .....	340
The Home/Index view.....	342
The Home/Details view.....	342
The Favorites controller.....	344
The Favorites/Index view .....	346
<b>How to work with cookies.....</b>	<b>348</b>
How to work with session cookies.....	348
How to work with persistent cookies.....	348
<b>The NFL Teams 4.0 app .....</b>	<b>350</b>
The cookies class .....	350
The updated session class .....	350
The updated Home controller.....	352
The updated Favorites controller.....	354
<b>Perspective .....</b>	<b>356</b>

## How ASP.NET MVC handles state

The data that a web app maintains for a user, such as variables, is the *state* of that app. However, HTTP is a *stateless* protocol. This means that HTTP doesn't keep track of an app's state between round trips. Rather, once a browser makes a request and receives a response, the app terminates and its state is lost.

### Six ways to maintain state

Figure 9-1 presents two tables that describe some of the ways that you can maintain state in a web app between HTTP requests. The first table presents four features for maintaining state that are common to all web apps (hidden fields, query strings, session state, and cookies). And the second table presents two features for maintaining state that are specific to ASP.NET Core MVC (routes and TempData).

This book has already shown how to use most of the features in these tables. For example, chapter 6 showed how to use route segments and query strings, and chapter 8 showed how to use TempData. In addition, many of the apps presented so far in this book use hidden fields to post data to the server. Now, this chapter shows how to use session state and cookies to maintain state.

### An introduction to session state

To understand how session state works, you need to know a little about cookies. A *cookie* is a key/value pair that's created on the server and passed to the user's browser in an HTTP response. Then, the browser passes the cookie back to the server with each subsequent HTTP request.

When you use *session state*, the data for a user session is stored in key/value pairs on the server. This data persists across all the HTTP requests that a user makes until the app removes the data or the session ends. A session can end in one of two ways. First, the user can end the session by closing the browser. Second, the session can end when the browser is open but the app is inactive for a specified period of time.

To make it possible for session state to work, the server creates a session ID to identify the user's session. Then, it sends this session ID to the user's browser in a cookie. With each subsequent request, the user's browser sends the cookie with the session ID back to the server, and the server uses that ID to retrieve the session data associated with that user.

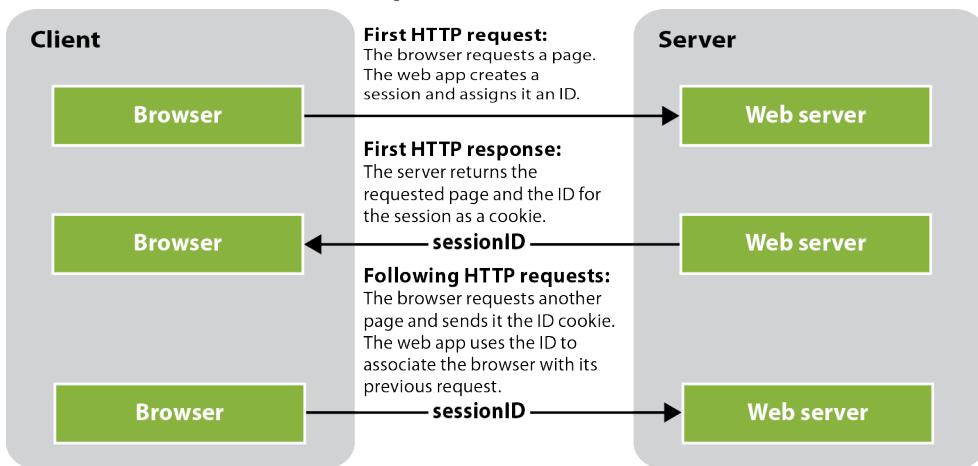
## Common web programming features for maintaining state

Feature	Description
Hidden field	Uses an <input> element to store data in a page that posts back to the server.
Query string	Uses the URL to store data and pass it between requests. See chapter 6.
Cookie	Uses a cookie object to store data in the user's browser or on hard disk. Then, the browser passes that data to the server with every subsequent request.
Session state	Uses a session state object to store data throughout the user's session.

## Two ASP.NET Core MVC features for maintaining state

Feature	Description
Routes	Uses the URL to store data and pass it between requests. See chapter 6.
TempData	Uses a session state object to store data until it is read. See chapter 8.

## How ASP.NET Core MVC keeps track of a session



### Description

- *State* refers to the current status of the properties, variables, and other data maintained by an app for a single user.
- HTTP is a *stateless protocol*. That means that it doesn't keep track of state between round trips. Once a browser makes a request and receives a response, the app terminates and its state is lost.
- ASP.NET Core MVC provides several ways to maintain the state of a web app.
- A *cookie* is a key/value pair passed to the user in an HTTP response and passed back to the server with each subsequent HTTP request.
- *Session state* works by having the server store the session data for a user and by using a cookie to associate the user's web browser with the correct session data.

Figure 9-1 How ASP.NET MVC handles state

## How to work with session state

---

Session state allows you to store data that you want to persist across multiple HTTP requests. This feature is extremely useful and commonly used by many types of web apps such as shopping cart apps.

### How to configure an app to use session state

---

By default, ASP.NET Core MVC doesn't enable session state. Because of that, you need to configure the middleware pipeline for your app so it includes all of the services needed for session state. To do that, open the Startup.cs file and add some statements to the ConfigureServices() and Configure() methods as shown in figure 9-2.

The first code example presents a ConfigureServices() method that adds session state to the services for the app. To do that, it calls the AddMemoryCache() and AddSession() methods that are available from the services parameter. When you call these methods, you must call them before you call the AddControllersWithViews() method.

The second code example presents a Configure() method that identifies the service to use for session state. To do that, it calls the UseSession() method from the app parameter. For this to work, you must call this method before you call the UseEndpoints() method.

If you configure your app as shown in the first two code examples, you're ready to begin working with session state using its default settings. In some cases, however, you may want to change these defaults settings. For instance, you might want to change how long a session can be inactive before it times out.

The last code example shows how to change some of the default settings for a session. To do that, you can pass a SessionOptions object to the AddSession() method that's called in the ConfigureServices() method.

In this example, the first statement uses the IdleTimeout property to change the timeout from 20 minutes to 5 minutes. Reducing the idle timeout can be useful in development when you're testing.

The next two statements use the Cookie property to change how the session cookie works. To start, the second statement sets the HttpOnly property to false. This allows client-side scripts to access the cookie. By default, this isn't allowed to protect from certain scripting attacks.

The third statement sets the IsEssential property to true to indicate that the session cookie is required for the app to function properly. This allows session state to work even if your web app requires users to consent to your cookie policy before the app can use non-essential cookies as specified by the General Data Protection Regulation (GDPR) requirements developed by the European Union (EU). That's because the session cookie is non-essential by default.

Similarly, the TempData cookie is non-essential by default, but you can change it to essential if necessary.

## How to configure an app to use session state

### The ConfigureServices() method in the Startup.cs file

```
public void ConfigureServices(IServiceCollection services)
{
    ...
    // must be called before AddControllersWithViews()
    services.AddMemoryCache();
    services.AddSession();

    services.AddControllersWithViews();
    ...
}
```

### The Configure() method in the Startup.cs file

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env)
{
    ...

    // must be called before UseEndpoints()
    app.UseSession();

    app.UseEndpoints(endpoints =>
    {
        endpoints.MapControllerRoute(
            name: "default",
            pattern: "{controller=Home}/{action=Index}/{id?}");
    });
    ...
}
```

## How to change the default session state settings

```
services.AddSession(options =>
{
    // change idle timeout to 5 minutes - default is 20 minutes
    options.IdleTimeout = TimeSpan.FromSeconds(60 * 5);
    options.Cookie.HttpOnly = false;      // default is true
    options.Cookie.IsEssential = true;   // default is false
});
```

### Description

- By default, session state is not enabled. To enable it, you can add code to the ConfigureServices() and Configure() methods in the Startup.cs file that calls the AddMemoryCache(), AddSession(), and UseSession() methods in the correct sequence.
- To change the default settings for session state, you can use a lambda expression to pass a SessionOptions object to the AddSession() method.

---

Figure 9-2 How to configure an app to use session state

## How to work with session state items in a controller

---

The ISession interface provides methods that allow you to set, get, and remove items from session state. The table in figure 9-3 presents some of these methods. Here, the two set methods allow you to store an int or a string in session state, the two get methods allow you to retrieve an int or a string from session state, and the remove method allows you to remove an item from session state.

All of the methods in this figure accept a key as the first argument. This key is a string that uniquely identifies a key/value pair in session state. The set methods also accept an int or string value as the second argument. This value is stored as the value part of the key/value pair in session state.

Both of the get methods return nullable types. In other words, they return a string or a nullable int. As a result, if no value in session state is associated with the specified key, these methods return null.

The Controller class includes a property named HttpContext that's of the HttpContext type. This class contains information about the current HTTP request. One of its properties, Session, is an implementation of the ISession interface. As a result, you can use this property to work with the items in session state.

To enable all of the features of this Session property, your code must include the using directive shown in the first example. Then, you can use the Session object as shown by the second example. Here, the first statement uses the GetInt32() method of the Session property to retrieve an int value. The second statement increments that value. And the third statement uses the SetInt32() method of the Session property to store the updated int value in session state.

Note that the second code example doesn't pass the int value to the view. That's because the view can retrieve the int value from session state as shown by the later examples in this figure.

## How to get session state values in a view

---

A view provides a property named Context that's of the HttpContext type. To enable all of the features of its Session property, you must specify a using directive like the one shown in the third example.

The next two code examples show how to use the Context property of the view to work with session state. The first example presents a Razor code block that uses the GetInt32() method of the Session property to retrieve the int value from session state. The second code example also retrieves the int value from session state, but it uses an inline Razor expression within a <div> element. As a result, the <div> element displays the int value on the web page.

## Methods of the **ISession** interface that set, get, and remove items

Method	Description
<b>SetInt32(key, value)</b>	Stores the int value in the session object and associates it with the specified key.
<b>SetString(key, value)</b>	Stores the string value in the session object and associates it with the specified key.
<b>GetInt32(key)</b>	Returns the int value associated with the specified key, or null if there's no value.
<b>GetString(key)</b>	Returns the string value associated with the specified key, or null if there's no value.
<b>Remove(key)</b>	Removes the value associated with the specified key if the key is found.

### A using directive for session state in a controller

```
using Microsoft.AspNetCore.Http;
```

### An action method that gets and sets a session state value

```
public ViewResult Index() {
    int num = HttpContext.Session.GetInt32("num");
    num += 1;
    HttpContext.Session.SetInt32("num", num);
    return View();
}
```

### A using directive for session state in a view

```
@using Microsoft.AspNetCore.Http
```

### A Razor code block that gets a session state value

```
@{
    int num = Context.Session.GetInt32("num");
}
```

### A Razor expression that gets a session state value

```
<div>@Context.Session.GetInt32("num")</div>
```

### Description

- The `HttpContext` class has a property named `Session` that implements `ISession` and thus provides methods for setting, getting, and removing items in session state.
- To enable all the functionality of the `Session` property, you must import the `Microsoft.AspNetCore.Http` namespace.
- A controller has an `HttpContext` property that has a data type of `HttpContext`.
- A view has a `Context` property that has a data type of `HttpContext`.
- In ASP.NET Core MVC, session state can only store int and string values. However, you can extend session state so it can store more complex types as shown in the next figure.

---

Figure 9-3 How to work with session state items in controllers and views

## How to use JSON to store objects in session state

---

The ISession interface provides methods for storing string and int values but doesn't provide methods for storing more complex data types. To get around this limitation, you can *serialize* an object of a complex type by converting it to a string and storing that string in session state. Later, you can *deserialize* the object by converting its string back into an object.

*JSON (JavaScript Object Notation)* is a data format that's often used for serialization. Historically, .NET programmers have relied on third-party libraries such as Newtonsoft.Json.NET to work with JSON.

With ASP.NET Core 3.0 and later, the NuGet package for Newtonsoft JSON is not included in your app. As a result, you will need to follow the steps in figure 9-4 to add the Newtonsoft.Json NuGet package to your app. Once the NuGet package is installed, you need to add it to the middleware pipeline by modifying the Startup.cs file as shown by the first code example. Here, the ConfigureServices() method in the Startup.cs file includes a call to the method that adds the Newtonsoft JSON library.

Once Json.NET is added to your app, you can include a using directive for that library like the one in the second code example. Then, you can use the methods of the JsonConvert class to serialize and deserialize objects.

The table in this figure shows two of the static methods of the JsonConvert class. Here, the SerializeObject() method accepts an object of any data type and returns a JSON string, and the DeserializeObject<T>() method accepts a JSON string and returns an object of the specified type.

The first code example below the table shows how to store a Team object in session state as a JSON string. Here, the first statement creates a Team object with values for its TeamID and Name properties. Then, the second statement uses the SerializeObject() method to convert the Team object to a JSON string and store it in a string variable named teamJson. At this point, the teamJson variable contains the following string:

```
{"TeamID": "sea", "Name": "Seattle Seahawks", "Conference": null,  
"Division": null, "LogoImage": null}
```

Finally, the third statement stores the string in session state by passing the teamJson variable to the SetString() method of the Session property.

The last code example shows how to retrieve a JSON string from session state and convert it back to a Team object. Here, the first statement calls the GetString() method of the ISession interface to retrieve the JSON string from session state and store it in a variable named teamJson. Then, the second statement calls the DeserializeObject<Team>() method to convert the JSON string to a Team object.

When you call the SerializeObject() method, you don't need to explicitly specify the data type. That's because SerializeObject() can infer the type based on the type of the argument it receives. However, when you call the DeserializeObject<T>() method, you must specify the data type. That's because this method can't infer the data type from its argument.

## How to add the Newtonsoft JSON NuGet package to your app

1. Use the Tools→NuGet Package Manager→Manage NuGet Packages for Solution command to open the NuGet Package Manager.
2. Click the Browse link.
3. Type “Microsoft.AspNetCore.Mvc.NewtonsoftJson” in the search box.
4. Click on the appropriate package from the list that appears in the left-hand panel.
5. In the right-hand panel, check the project name, select the version that matches the version of .NET Core you’re running, and click Install.
6. Review the Preview Changes dialog that comes up and click OK.
7. Review the License Acceptance dialog that comes up and click I Accept.

## How to configure your app to use the Newtonsoft JSON library

```
public void ConfigureServices(IServiceCollection services) {
    ...
    services.AddMemoryCache();
    services.AddSession();

    services.AddControllersWithViews().AddNewtonsoftJson();
    ...
}
```

### A using directive that imports the Newtonsoft JSON library

```
using Newtonsoft.Json;
```

### Two static methods of the JsonConvert class

Method	Description
<code>SerializeObject(object)</code>	Converts an object to a JSON string.
<code>DeserializeObject&lt;T&gt;(string)</code>	Converts a JSON string to an object of type T.

### Code in an action method that sets a Team object in session state

```
Team team = new Team { TeamID = "sea", Name = "Seattle Seahawks" };
string teamJson = JsonConvert.SerializeObject(team);
HttpContext.Session.SetString("team", teamJSON);
```

### Code in an action method that gets a Team object from session state

```
string teamJson = HttpContext.Session.GetString("team");
Team team = JsonConvert.DeserializeObject<Team>(teamJson);
```

### Description

- *JSON (JavaScript Object Notation)* is a data format that facilitates the transfer of data.
- To *serialize* .NET objects to JSON strings and back again, you can use the open-source Newtonsoft.Json.NET library.
- You can use the `JsonIgnore` attribute of the `Newtonsoft.Json` namespace to mark properties in an object that you don’t want to serialize.

Figure 9-4 How to use JSON to store objects in session state

## How to extend the `ISession` interface

So far, this chapter has shown some low-level techniques for working with session state that are effective but lead to unwieldy code that can be hard to read. That's because you have to include using directives in multiple places, and the method calls for both the `HttpContext` class and the `JsonConvert` class are long. In addition, these techniques require you to use keys to access your session state items. These keys can be hard to remember from one file to the next and are prone to typos that you don't discover until runtime.

That's why the next two figures present two techniques that you can use to make it easier to work with session state. First, figure 9-5 shows how to add extension methods to the `ISession` interface to make it easier to work with JSON. Then, figure 9-6 shows how to create a wrapper class for all of the session state code for your app.

Both of these techniques encapsulate the using directives, method calls, and string keys in one place. As a result, these statements aren't scattered throughout your app. In addition, you can use techniques like these with `TempData` and cookies.

The first code example presents two generic extension methods for the `ISession` interface named `SetObject<T>()` and `GetObject<T>()`. The file that contains these extension methods starts with using directives for the namespaces that contain the `JsonConvert` class and the `ISession` interface. Then, the example uses an ellipsis (...) to indicate that the namespace statement for this class isn't shown. Within the class, the set method uses the `SerializeObject()` method to store an object of type T in the session state object, and the get method uses the `DeserializeObject<T>()` method to retrieve an object of type T from the session state object.

The second code example starts by showing how to use these extension methods in a controller or a view. In the controller, the first statement uses a key of "team" to get a `Team` object from session state. If no `Team` object exists for this key, the code creates a new `Team` object. Then, the second statement changes the name that's stored in the `Team` object, and the third statement sets the `Team` object in the session state object with a key of "team".

In the view, the statement in the Razor code block gets the `Team` object that's stored with a key of "team". This works similarly to the controller code, except that it uses the `Context` property of the view to access the `Session` property, not the `HttpContext` property of the controller.

The third code example also shows how to use these extension methods in a controller or a view, but it works with a list of `Team` objects instead of a single `Team` object. The main difference here is that the second statement in the controller adds a new `Team` object to the list instead of changing the name for the `Team` object.

When working with the extension methods, the call to the `SetObject<T>()` method doesn't explicitly specify the data type. That's because the compiler can infer the object type based on the type of the second argument. However, the compiler can't infer the object type for the `GetObject<T>()` method because it always receives a string argument.

## Two extension methods for the ISession interface

```
using Microsoft.AspNetCore.Http;
using Newtonsoft.Json;
...
public static class SessionExtensions
{
    public static void SetObject<T>(this ISession session,
                                    string key, T value)
    {
        session.SetString(key, JsonConvert.SerializeObject(value));
    }

    public static T GetObject<T>(this ISession session, string key)
    {
        var valueJson = session.GetString(key);
        if (string.IsNullOrEmpty(value)) {
            return default(T);
        }
        else {
            return JsonConvert.DeserializeObject<T>(valueJson);
        }
    }
}
```

## Code that uses the extension methods to work with a single team

### In a controller

```
var team = HttpContext.Session.GetObject<Team>("team") ?? new Team();
team.Name = "Seattle Seahawks";
HttpContext.Session.SetObject("team", team);
```

### In a view

```
@{
    var team = Context.Session.GetObject<Team>("team");
}
```

## Code that uses the extension methods to work with a list of teams

### In a controller

```
var teams = HttpContext.Session.GetObject<List<Team>>("teams") ??
           new List<Team>();
teams.Add(new Team { TeamID = "gb", Name = "Green Bay Packers" });
HttpContext.Session.SetObject("teams", teams);
```

### In a view

```
@{
    var teams = Context.Session.GetObject<List<Team>>("teams");
}
```

## Description

- To make it easier to store objects in session state, you can add extension methods to the ISession interface.

---

Figure 9-5 How to extend the ISession interface

## How to use a wrapper class

---

Although the code in the previous figure was an improvement over the code in figure 9-4, it's still unwieldy. That's why figure 9-6 shows how to create a wrapper class for all the session state code for your app. A *wrapper class* is any class that "wraps up" or encapsulates the functionality of another class or component. For example, the MySession class encapsulates the using directives, method calls, and string keys necessary to work with an ISession object and its extension methods. As a result, these statements aren't scattered throughout your app. In addition, you can use a similar technique to work with TempData and cookies.

The first code example presents a wrapper class named MySession. The file that contains this class starts with a using directive for the namespace that stores the ISession interface. Then, the class defines a constant value for the "teams" key. Next, it defines a constructor that receives an ISession argument and stores it in a private property named session. Finally, it defines two methods named GetTeams() and SetTeams() that use the ISession extension methods to store and retrieve a list of Team objects in session state.

The second code example shows how to use the MySession class in a controller or a view. In the controller, this code starts by passing the Session property of the HttpContext property to the constructor to create a MySession class. After that, it calls the GetTeams() method to get a list of Team objects from session state, adds a new Team object to the list, and calls the SetTeams() method to store the new list in session state. This code is a big improvement over the third example of the previous figure.

In the view, the first statement in the Razor code block creates the MySession object. This works similarly to the controller code except that it uses the Context property of the view, not the HttpContext property of the controller. Then, the second statement calls the GetTeams() method of the MySession object to get a list of Team objects.

## A wrapper class that encapsulates the code for working with session state

```
using Microsoft.AspNetCore.Http;
...
public class MySession
{
    private const string TeamsKey = "teams";

    private ISession session { get; set; }
    public MySession(ISession sess) {
        session = sess;
    }

    public List<Team> GetTeams() =>
        session.GetObject<List<Team>>(TeamsKey) ?? new List<Team>();

    public void SetTeams(List<Team> teams) =>
        session.SetObject(TeamsKey, teams);
}
```

## Code that uses the wrapper class to work with a list of teams

### In a controller

```
var session = new MySession(HttpContext.Session);
var teams = session.GetTeams();
teams.Add(new Team { TeamID = "gb", Name = "Green Bay Packers" });
session.SetTeams(teams);
```

### In a view

```
@{
    var session = new MySession(Context.Session);
    var teams = session.GetTeams();
}
```

## Description

- To make it easier to work with session state in your app, you can create a wrapper class that encapsulates the using directives, method calls, and string keys in one place.
- A wrapper class can call extension methods from the ISession interface like the ones shown in the previous figure.

---

Figure 9-6 How to use a wrapper class

## The NFL Teams 3.0 app

---

The next few topics present a version of the NFL Teams app that keeps track of a user's favorite teams in session state. Since this app updates the NFL Teams app that was presented in chapter 8, this chapter only presents code that is new or changed. If you haven't read chapter 8 yet, you should at least review that chapter before you continue.

### The user interface

---

The first screen in figure 9-7 shows the "Return to Home Page" and "Add to Favorites" buttons that this app displays at the bottom of the Details page presented in chapter 8. The second screen shows the Home page after a user has clicked the "Add to Favorites" button from the Details page for two teams to add them to the Favorites page.

There are two things to notice about this Home page. First, it displays the message that indicates a new team has been added to your favorites only immediately after the team is added. If the user takes any other action on the Home page, such as selecting a new conference, this message disappears. That's because this message is transferred via TempData, so it's automatically removed after it's read.

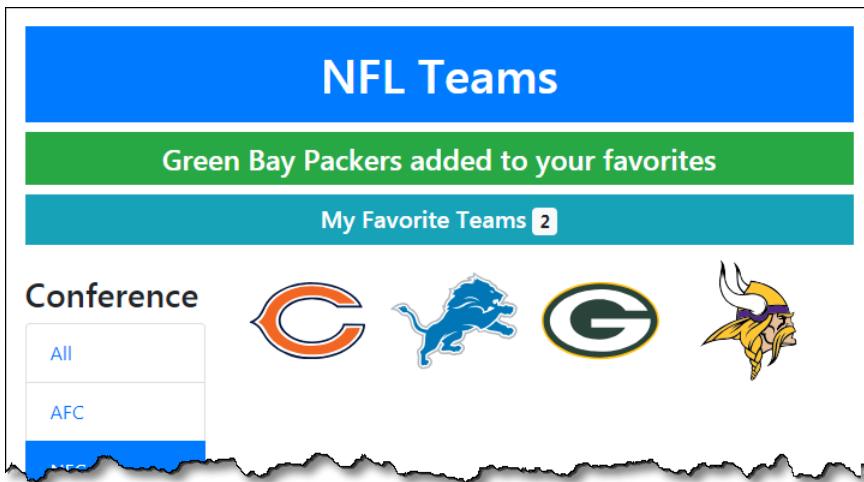
Second, the Home page displays a "My Favorite Teams" link that the user can click to view the teams that have been added to their favorites. To the right of this link, a Bootstrap badge displays a count of the user's favorite teams. This badge is displayed until the user closes the browser, the session times out, or the user clears the favorites. That's because the user's favorite teams are stored in session state.

The third screen shows the Favorites page. This page is displayed when the user clicks on the "My Favorite Teams" link on the Home page. This page retrieves the user's favorite teams from session state and displays them. It also includes a button to return to the Home page, and a button to clear all the teams from the Favorites page. When the user clicks this button, all the Team objects are removed from session state.

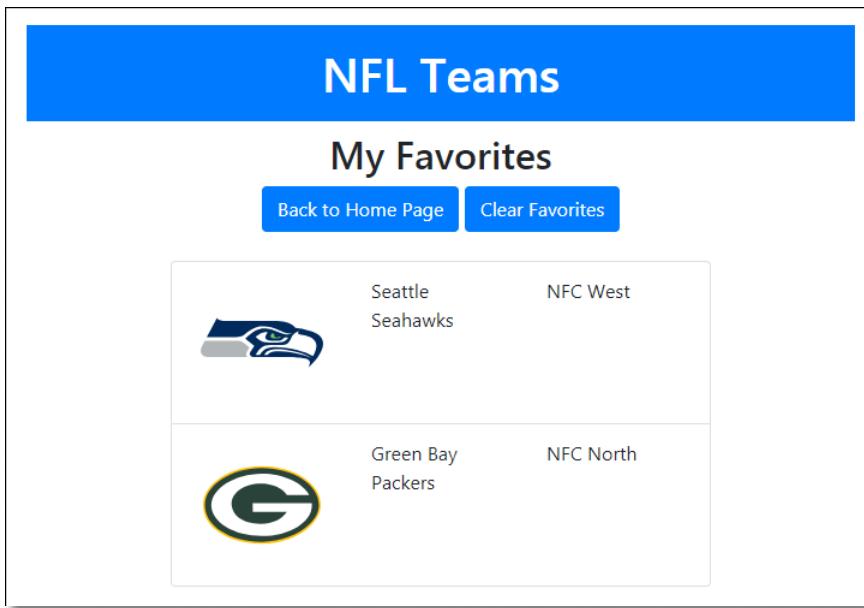
## Two buttons on the Details page

[Return to Home Page](#) [Add to Favorites](#)

## The Home page after a team has been added to favorites



## The Favorites page



## Description

- This version of the NFL Teams app enhances the apps presented in chapter 8. It allows you to add a team to your favorites and view your favorite teams.

Figure 9-7 The user interface of the NFL Teams 3.0 app

## The session classes

---

Figure 9-8 presents the two classes that have been added to the model layer for this version of the NFL app. The SessionExtensions class starts with using directives for the namespaces that contain the ISession interface and the Newtonsoft JSON library. Then, this class defines two generic extension methods for the ISession interface. This code is similar to the code presented earlier in this chapter, except that it uses a conditional operator instead of an if statement in the GetObject<T>() method.

The NFLSession class starts with a using directive that makes it easy to work with session state. Then, this class encapsulates all the session state calls that the NFL Teams app needs to make.

To start, it defines four private constants for key names. The use of constants like this decreases the chance of typos and errors when working with session state keys. It also lets you use IntelliSense when you're typing keys.

After defining the constants, this class defines a constructor that accepts an argument of the ISession type. Then, it stores the value it receives in a private property named session.

After the constructor, this class defines several methods that store and retrieve session state items. The SetMyTeams() method accepts a list of Team objects. Then, it uses the SetObject() extension method of the SessionExtensions class to store the list of teams in session state. In addition, it uses the SetInt32() method of the ISession interface to store a count of the teams in session state. This class stores the count of teams separately so the app doesn't have to retrieve and deserialize the entire list when it only needs a count of the teams.

The GetMyTeams() method and GetMyTeamCount() methods retrieve the data stored by the SetMyTeams() method. Here, GetMyTeams() gets a list of the teams, and GetMyTeamCount() retrieves the count of teams. Both of these methods return a default value if there's no value in session state. This is necessary for the GetMyTeamCount() method because the GetInt32() method of the ISession interface returns a nullable int.

The SetActiveConf() method uses the SetString() method of the ISession interface to store a string in session state, and the GetActiveConf() method uses the GetString() method to retrieve it. The SetActiveDiv() and GetActiveDiv() methods work similarly.

Finally, the RemoveMyTeams() method uses the Remove() method of the ISession interface to remove all data about favorite teams from session state. To do that, it removes both the list and count of the favorite teams.

## The SessionExtensions class

```
using Microsoft.AspNetCore.Http;
using Newtonsoft.Json;
...
public static class SessionExtensions
{
    public static void SetObject<T>(this ISession session,
        string key, T value)
    {
        session.SetString(key, JsonConvert.SerializeObject(value));
    }

    public static T GetObject<T>(this ISession session, string key)
    {
        var value = session.GetString(key);
        return (value == null) ? default(T) :
            JsonConvert.DeserializeObject<T>(value);
    }
}
```

## The NFLSession class

```
using Microsoft.AspNetCore.Http;
...
public class NFLSession
{
    private const string TeamsKey = "myteams";
    private const string CountKey = "teamcount";
    private const string ConfKey = "conf";
    private const string DivKey = "div";

    private ISession session { get; set; }
    public NFLSession(ISession session)
    {
        this.session = session;
    }

    public void SetMyTeams(List<Team> teams) {
        session.SetObject(TeamsKey, teams);
        session.SetInt32(CountKey, teams.Count);
    }
    public List<Team> GetMyTeams() =>
        session.GetObject<List<Team>>(TeamsKey) ?? new List<Team>();
    public int GetMyTeamCount() => session.GetInt32(CountKey) ?? 0;

    public void SetActiveConf(string activeConf) =>
        session.SetString(ConfKey, activeConf);
    public string GetActiveConf() => session.GetString(ConfKey);

    public void SetActiveDiv(string activeDiv) =>
        session.SetString(DivKey, activeDiv);
    public string GetActiveDiv() => session.GetString(DivKey);

    public void RemoveMyTeams() {
        session.Remove(TeamsKey);
        session.Remove(CountKey);
    }
}
```

---

Figure 9-8 The session classes

## The Home controller

---

Figure 9-9 presents some of the action methods for the Home controller. In this version of the NFL Teams app, the Home controller uses session state to store and retrieve the active conference and division IDs.

The updated Index() action method starts by creating an NFLSession object. To do that, it passes the Session property of the controller's HttpContext property to the constructor of the NFLSession class. Then, the code calls methods from the session state object to store the IDs of the active conference and division in session state. The rest of the code is the same as in figure 8-11. To review, that code initializes a new TeamListViewModel object with data and transfers it to the view via the View() method.

The updated Details() action method now uses session state instead of TempData. That way, users can refresh the Details page as many times as they want and it will still “remember” the active conference and division when users return to the Home page. That’s because the data in session state persists across multiple requests.

The Add() action method handles the POST request that’s run when users click the “Add to Favorites” button on the Details page. This method receives a TeamViewModel object as its parameter. However, when users click the “Add to Favorites” button, the app only posts the TeamID. Because of that, this code starts by using the TeamID it receives to get the rest of the data for the selected team from the database.

After getting data from the database, this code creates a new NFLSession object and calls the GetMyTeams() method from it. This retrieves the list of favorite teams from session state. Then, it adds the team to the list and calls the SetMyTeams() method to store the updated list in session state. Remember, SetMyTeams() also updates the team count value that’s stored in session state.

After updating session state, the code stores a message in TempData about the team that was just added to favorites. This message displays on the Home page. Since it’s stored in TempData, it only persists until the app reads it once.

Finally, the code redirects the user back to the Home page. To do that, it gets the ID values of the active conference and division from session state. Then, it builds an anonymous object to pass those IDs to the activeConf and activeDiv route parameters of the URL.

### The updated Index() action method of the Home controller

```
public ViewResult Index(string activeConf = "all",
                        string activeDiv = "all")
{
    var session = new NFLSession(HttpContext.Session);
    session.SetActiveConf(activeConf);
    session.SetActiveDiv(activeDiv);

    // rest of code same as figure 8-11
}
```

### The updated Details() action methods of the Home controller

```
public ViewResult Details(string id)
{
    var session = new NFLSession(HttpContext.Session);
    var model = new TeamViewModel
    {
        Team = context.Teams
            .Include(t => t.Conference)
            .Include(t => t.Division)
            .FirstOrDefault(t => t.TeamID == id),
        ActiveDiv = session.GetActiveDiv(),
        ActiveConf = session.GetActiveConf()
    };
    return View(model);
}
```

### The Add() action method of the Home controller

```
[HttpPost]
public RedirectToActionResult Add(TeamViewModel model)
{
    model.Team = context.Teams
        .Include(t => t.Conference)
        .Include(t => t.Division)
        .Where(t => t.TeamID == model.Team.TeamID)
        .FirstOrDefault();

    var session = new NFLSession(HttpContext.Session);
    var teams = session.GetMyTeams();
    teams.Add(model.Team);
    session.SetMyTeams(teams);

    TempData["message"] = $"{model.Team.Name} added to your favorites";

    return RedirectToAction("Index",
        new {
            ActiveConf = session.GetActiveConf(),
            ActiveDiv = session.GetActiveDiv()
        });
}
```

---

Figure 9-9 The Home controller

## The layout

---

Figure 9-10 presents the code for the Razor layout for this new version of the NFL Teams app. This view works much like the layout presented in the previous chapter, but it adds two elements.

First, the layout displays any message that's in TempData. To do that, it uses a Razor if statement to check whether there's an item in TempData with a key of "message". If there is, the layout displays it in an `<h4>` element that's styled with Bootstrap.

Second, the layout displays a link to the Favorites page and the number of teams that have been added to favorites. However, it doesn't display this link and team count if the app is already displaying the Favorites page. To determine which page is currently displayed, the layout uses a Razor if statement to inspect the ViewContext property described in chapter 7.

If the current page isn't the Favorites page, the Razor code block starts by creating a new NFLSession object and passing it the Session property of the Context property. After that, the layout uses an `<h5>` element and styles it with Bootstrap.

Within the `<h5>` element, the layout begins by constructing an `<a>` element that redirects to the Favorites controller. Since the `<a>` element doesn't include an asp-action tag helper, MVC generates a URL that redirects to the Index() action method of the Favorites controller.

After the `<a>` element, the view constructs a `<span>` element that's styled as a Bootstrap badge. Within the `<span>` element, the layout uses a Razor inline expression to get the number of favorite teams from session state.

## The layout

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <link href="~/lib/bootstrap/dist/css/bootstrap.min.css" rel="stylesheet" />
    <link href="~/css/site.css" rel="stylesheet" />
    <title>@ViewBag.Title</title>
</head>
<body>
    <div class="container">
        <header class="text-center text-white">
            <h1 class="bg-primary mt-3 p-3">NFL Teams</h1>

            @* show any message in TempData *@
            @if (TempData["message"] != null)
            {
                <h4 class="bg-success p-2">@TempData["message"]</h4>
            }

            @* show link to Favorites page unless on Favorites page *@
            @if (!ViewContext.View.Path.Contains("Favorites"))
            {
                var session = new NFLSession(Context.Session);
                <h5 class="bg-info p-2">
                    <a asp-controller="Favorites"
                        class="text-white">My Favorite Teams</a>

                    @* display number of fav teams in badge *@
                    <span class="badge badge-light">
                        @(session.GetMyTeamCount())</span>
                </h5>
            }
        </header>
        <main>
            @RenderBody()
        </main>
    </div>
</body>
</html>
```

## Description

- The layout checks whether TempData stores a message. If so, it displays that message.
- The layout checks whether the current page is the Favorites page. If not, it displays a link to the Favorites page, gets the number of favorite teams from session state, and displays that number in a Bootstrap badge.

---

Figure 9-10 The layout

## The Home/Index view

---

The first example in figure 9-11 presents the code for the section of the Home/Index view that builds a link that contains an image for each team. Each of these links displays a team logo and uses a GET request to pass the team ID to the Details() action method when a user clicks the logo.

## The Home/Details view

---

The second example presents the code for the section of the Home/Details view that builds the “Return to Home Page” and “Add to Favorites” buttons shown at the top of figure 9-7.

Here, the `<a>` element that creates a link for the “Return to Home Page” button uses Bootstrap styles to style the link as a button. Also, the version in the link is coded within a `<form>` element that posts to the Add() action method. This displays the “Return to Home Page” link that’s styled as a button side-by-side with the “Add to Favorites” button. If you moved this link outside the `<form>` element, it would still function properly, but it wouldn’t be displayed side-by-side with the `<button>` element.

Since the “Add to Favorites” button is a submit button, it must be coded inside the `<form>` element. When a user clicks this button, the app makes a POST request to the Add() action method. Below this submit button, the code includes a hidden field whose value is the TeamID of the currently selected team. This posts the TeamID to the Add() action method and binds it to the TeamViewModel object that the Add() method accepts as an argument.

### Some code from the Home/Index view

```
<div class="col-sm-9">
    <ul class="list-inline">
        @foreach (Team team in Model.Teams)
        {
            <li class="list-inline-item">
                <a asp-action="Details" asp-route-id="@team.TeamID">
                    <img src("~/images/@team.LogoImage" alt="@team.Name"
                        title="@team.Name | @team.Conference.Name
                        @team.Division.Name" />
                </a>
            </li>
        }
    </ul>
</div>
```

### Some code from the Home/Details view

```
<li class="list-group-item">
    <form asp-action="Add" method="post">
        <a asp-action="Index" class="btn btn-primary"
            asp-route-activeConf="@Model.ActiveConf"
            asp-route-activeDiv="@Model.ActiveDiv">
            Return to Home Page</a>
        <button type="submit" class="btn btn-primary">
            Add to Favorites
        </button>
        <input type="hidden" asp-for="Team.TeamID" />
    </form>
</li>
```

### Description

- The Home/Index view displays the images for the teams as links.
- The Home/Details view includes an “Add to Favorites” button that uses a `<form>` element to post the team ID to the `Add()` action method.
- The Home/Details view includes a “Return to Home Page” link that’s styled as a button. This link returns to the Home page with the active conference and division selected.

---

Figure 9-11 The Home/Index and Home/Details views

## The Favorites controller

---

Figure 9-12 presents the Favorites controller of the NFL Teams app. This controller has an Index() action method that handles GET requests and a Delete() action method that handles POST requests.

The app calls the Index() action method when the user clicks on the “My Favorite Teams” link. This action method starts by creating a new NFLSession object and passing it the Session property of the controller’s HttpContext property. Then, it creates a new TeamListViewModel object and uses the session state object to load it with data from session state. Specifically, it gets the active conference and division IDs and the list of favorite teams. Finally, it transfers this view model to the view by passing it to the View() method.

The app calls the Delete() action method when the user clicks on the “Clear Favorites” button on the Favorites page. Because it doesn’t need any data from the page to do its work, this action method doesn’t accept any arguments.

The Delete() action method starts by creating a new NFLSession object and passing it the Session property of the controller’s HttpContext property. Then, it calls the RemoveMyTeams() method of the NFLSession object. This causes both the list of favorite teams and the team count value to be removed from session state.

After removing this data from session state, the code stores a message in TempData that tells the user that their favorite teams have been cleared. The layout displays this message as described in figure 9-10. Then, after the layout reads the message, it doesn’t persist in TempData. That’s why TempData is ideal for displaying temporary messages that you don’t want to store in session state.

Finally, the code redirects the user back to the Home page. To do that, it gets the ID values of the active conference and division that are stored in session state and uses them to build the route parameters of the URL.

## The Favorites controller

```
public class FavoritesController : Controller
{
    [HttpGet]
    public ViewResult Index()
    {
        var session = new NFLSession(HttpContext.Session);
        var model = new TeamListViewModel
        {
            ActiveConf = session.GetActiveConf(),
            ActiveDiv = session.GetActiveDiv(),
            Teams = session.GetMyTeams()
        };
        return View(model);
    }

    [HttpPost]
    public RedirectToActionResult Delete()
    {
        var session = new NFLSession(HttpContext.Session);
        session.RemoveMyTeams();

        TempData["message"] = "Favorite teams cleared";

        return RedirectToAction("Index", "Home",
            new {
                ActiveConf = session.GetActiveConf(),
                ActiveDiv = session.GetActiveDiv()
            });
    }
}
```

## Description

- The Favorites controller has two action methods.
- The Index() action method is called when a user clicks the My Favorite Teams link. This action method retrieves data from session state and transfers it to the view for display.
- The Delete() action method is called when a user clicks the Clear Favorites button on the Favorites page. This action method removes the favorite teams from session state and redirects the user to the Home page using data from session state.

---

Figure 9-12 The Favorites controller

## The Favorites/Index view

---

Figure 9-13 presents the code for the Favorites/Index view for this version of the NFL Teams app. The model object for this view is a TeamListViewModel object. As you may recall, that's what the Index() action method of the Favorites controller transfers to it.

In this view, the HTML consists of two `<div>` elements. The first uses the Bootstrap `text-center` class to center the `<a>` and `<button>` elements within the `<div>` element. It also includes a `<form>` element that posts to the `Delete()` action method of the current controller, which is the Favorites controller. Since the `<a>` and `<button>` elements are both within the `<form>` element, they display side-by-side.

In the first `<div>` element, the `<a>` element uses Bootstrap classes to style the link like a button. In addition, it links back to the Home page. To do that, it uses the properties of the model object to add the route parameters for the active conference and division IDs.

Remember that when the controller builds the `TeamListViewModel` object that it transfers to this view, it gets the active conference and division IDs from session state. As a result, a user can refresh this page multiple times, and this link always “remembers” the active conference and division. That's because the data in session state persists across multiple requests.

In the first `<div>` element, the submit button starts a POST request to the `Delete()` action method. This code doesn't include a hidden field or any other element with data that posts to the server. For this app, that isn't necessary because the `Delete()` action method doesn't need any data from the page.

The second `<div>` element uses Bootstrap classes to create a row with one indented column. The column displays the favorite teams in a `<ul>` element styled as a Bootstrap list-group. Within the `<ul>` element, a Razor `foreach` statement loops through all the `Team` objects in the `Teams` property of the model. For each team, it adds an item to the list group where each item displays that team's logo, name, conference, and division.

## The Favorites/Index view

```
@model TeamListViewModel
@{
    ViewData["Title"] = "Favorites";
}
<div class="text-center">
    <h2>My Favorites</h2>
    <form asp-action="Delete" method="post">
        <a asp-action="Index" asp-controller="Home" class="btn btn-primary"
            asp-route-activeConf="@Model.ActiveConf"
            asp-route-activeDiv="@Model.ActiveDiv">
            Back to Home Page</a>
        <button type="submit" class="btn btn-primary">
            Clear Favorites
        </button>
    </form>
    <br />
</div>

<div class="row">
    <div class="col-8 offset-2">
        <ul class="list-group">
            @foreach (Team team in Model.Teams)
            {
                <li class="list-group-item">
                    <div class="row">
                        <div class="col-sm-4">
                            
                        </div>
                        <div class="col-sm-4">
                            @team.Name
                        </div>
                        <div class="col-sm-4">
                            @team.Conference.Name @team.Division.Name
                        </div>
                    </div>
                </li>
            }
        </ul>
    </div>
</div>
```

## Description

- The Favorites/Index view displays information about the teams that users have added to their favorites. In addition, it includes a button and a link that has been styled as a button.
- The “Clear Favorites” button uses a `<form>` element to call the `Delete()` action method of the Favorites controller with a POST request.
- The “Return to Home Page” link gets the Home page with the active conference and division selected.

---

Figure 9-13 The Favorites/Index view

## How to work with cookies

A *cookie* is a key/value pair that's stored in the user's browser or on the user's disk. A web app sends a cookie to a browser via an HTTP response. Then, each time the browser sends an HTTP request to the server, it sends that cookie back.

A *session cookie* is stored in the browser's memory and exists only for the duration of the browser session. A *persistent cookie*, on the other hand, is stored on the user's disk and is retained until the cookie's expiration date, or until the user clears the cookie.

### How to work with session cookies

The first table in figure 9-14 presents two properties of the Controller class that you can use to work with cookies. The Request property represents the current HTTP request sent from the browser to the server, and the Response property represents the current HTTP response sent from the server to the browser. Each of these properties has a Cookies collection.

Below the table, the first two code examples show how to use the Response property to set and delete a cookie. The first example sets a cookie by calling the Append() method of the Cookies property and passing it two string values. The first string specifies the unique key, and the second string specifies the value to be stored.

The second example deletes a cookie by calling the Delete() method of the Cookies property and passing it a single string value. This string value specifies the key of the cookie to delete.

The third example shows how to use the Request property to retrieve, or get, a cookie. This code retrieves a value from the Cookies collection by specifying the associated key for the value. To do that, it uses brackets to specify the key. In other words, it doesn't use parentheses to call a method.

### How to work with persistent cookies

The second table presents some of the properties of the CookieOptions class. This class allows you to change some of the settings of a cookie and is stored in the Microsoft.AspNetCore.Http namespace. As a result, to use it, you need to include a using directive like the one shown in the first code example below this table.

To set a persistent cookie, you need to set the Expires property of the CookieOptions class. To do that, you create a CookieOptions object, set its Expires property to a DateTime value, and include the CookieOptions object when you create a cookie. This is shown by the last example in this figure. To start, it creates a new CookieOptions object with an Expires property of 30 days from the current date and time. Then, it passes that object to the Append() method as the third argument.

## Two properties of the Controller class

Property	Description
<b>Request</b>	Represents the HTTP request sent from the browser to the server.
<b>Response</b>	Represents the HTTP response sent from the server to the browser.

### Code that sets a session cookie

```
Response.Cookies.Append("username", "Grace");
```

### Code that deletes a cookie

```
Response.Cookies.Delete("username");
```

### Code that gets a cookie

```
string value = Request.Cookies["username"]; // brackets, not parentheses
```

## Some of the properties of the CookieOptions class

Property	Description
<b>Domain</b>	The domain to associate the cookie with. The default value is null.
<b>Expires</b>	The cookie's expiration date and time. The default value is null.
<b>Path</b>	The cookie's path. The default path is "/".
<b>MaxAge</b>	The maximum age for the cookie. The default value is null.
<b>SameSite</b>	The value for the cookie's SameSite attribute. The values can be Lax or Strict. The default value is Lax.
<b>Secure</b>	Indicates whether the cookie can be transmitted over HTTPS only. The default value is false.

## A using directive that's necessary to work with the CookieOptions class

```
using Microsoft.AspNetCore.Http;
```

### Code that sets a persistent cookie

```
var options = new CookieOptions { Expires = DateTime.Now.AddDays(30) };
Response.Cookies.Append("username", "Grace", options);
```

### Description

- A *cookie* is a key/value pair that's stored in the user's browser or on the user's disk.
- A web app sends a cookie to a browser via an HTTP response. Then, each time the browser sends an HTTP request to the server, it sends that cookie back.
- A *session cookie* is stored in the browser's memory and exists only for the duration of the browser session.
- A *persistent cookie* is stored on the user's disk and is retained until the cookie's expiration date or until the user clears the cookie.
- To set or delete a cookie, use the Cookies property of the controller's Response property.
- To get a cookie, use the Cookies property of the controller's Request property.
- To set a persistent cookie, use a CookieOptions object that expires in the future.

Figure 9-14 How to work with cookies

## The NFL Teams 4.0 app

---

Now that you know how to work with cookies, you're ready to see a version of the NFL Teams app after it has been updated to store the IDs of the user's favorite teams in a persistent cookie. That way, the app "remembers" its users' favorite teams, even when its users close their browsers and return to this app later.

### The cookies class

---

Figure 9-15 presents the NFLCookies class. This class encapsulates the code that the NFL Teams app needs to work with cookies. It starts with a using directive for the namespace that stores the interfaces for working with cookies. Then, after the class declaration, it declares two private constants. The first stores the key for accessing the team IDs, and the second stores the delimiter value (in this case, a dash) that's used to separate the team IDs.

The NFLCookies class has two constructors. The first accepts an object that stores a collection of request cookies, and the second accepts an object that stores a collection of response cookies. As a result, to create an NFLCookies object, you can pass it the cookie collection from a Request or Response object. After the constructors, this class defines a method that sets a persistent cookie, a method that gets that cookie, and a method that removes the cookie from the user's browser.

The SetMyTeamIds() method accepts a list of Team objects. Then, it uses the LINQ Select() method to get a list of team ID values. Next, it uses the Delimiter constant to join the list of strings into a single string where each string is separated by a dash. After that, the code initializes a CookieOptions object with an Expires property that's set to 30 days in the future. Finally, it deletes any previous cookie and stores the new one.

The GetMyTeamIds() method retrieves the persistent cookie that was set by the SetMyTeamIds() method and assigns it to a string variable. Then, it checks if that string is null or empty. If so, it returns an empty string array. Otherwise, it uses the Delimiter constant to split the string variable and return a string array of team ID values.

The RemoveMyTeamIds() method uses the Delete() method of the collection of response cookies to remove the persistent cookie. To do that, this method uses the responseCookies private object.

### The updated session class

---

Figure 9-15 shows the NFLSession class. This class works like the NFLSession class shown earlier in this chapter, but its GetMyTeamCount() method returns a nullable int type, not a regular int type. This makes it possible to check whether a session state object already exists or whether you should initialize a new object by getting data from cookies.

## The NFLCookies class

```
using Microsoft.AspNetCore.Http;
...
public class NFLCookies
{
    private const string MyTeams = "myteams";
    private const string Delimiter = "-";

    private IRequestCookieCollection requestCookies { get; set; }
    private IResponseCookies responseCookies { get; set; }

    public NFLCookies(IRequestCookieCollection cookies) {
        requestCookies = cookies;
    }
    public NFLCookies(IResponseCookies cookies) {
        responseCookies = cookies;
    }

    public void SetMyTeamIds(List<Team> myteams) {
        List<string> ids = myteams.Select(t => t.TeamID).ToList();
        string idsString = String.Join(Delimiter, ids);
        CookieOptions options = new CookieOptions {
            Expires = DateTime.Now.AddDays(30)
        };
        RemoveMyTeamIds(); // delete old cookie first
        responseCookies.Append(MyTeams, idsString, options);
    }

    public string[] GetMyTeamsIds() {
        string cookie = requestCookies[MyTeams];
        if (string.IsNullOrEmpty(cookie))
            return new string[] { }; // empty string array
        else
            return cookie.Split(Delimiter);
    }

    public void RemoveMyTeamIds(){
        responseCookies.Delete(MyTeams);
    }
}
```

## The updated NFLSession class

```
public class NFLSession
{
    ...
    public int? GetMyTeamCount() => session.GetInt32(CountKey);
    ...
}
```

### Description

- This app stores the IDs of the user's favorite teams in a persistent cookie.
- The NFLCookies class has an overloaded constructor that can accept the Cookies collection from the Request object or from the Response object.
- The GetMyTeamCount() method of the NFLSession class now returns a nullable int.

---

Figure 9-15 The cookies and session classes

## The updated Home controller

---

Figure 9-16 shows two action methods of the Home controller that have been updated to work with cookies. The Index() action method now retrieves the IDs of the user's saved favorite teams from a persistent cookie. And the Add() action method now updates the favorite team data in the persistent cookie after it updates session state.

The updated Index() action method starts as it did in figure 8-11. First, it creates a new NFLSession object and stores the IDs of the active conference and division in session state. Then, it calls the GetMyTeamCount() method of the NFLSession object. This assigns the return value to a nullable int variable named count. This works because the GetMyTeamCount() method now returns a nullable int.

After getting the count of teams, the code checks whether the count variable is null. If it is, session state doesn't contain any data. This typically means that the web browser is loading the app for the first time. In this case, the code creates a new NFLCookies object and passes it the Cookies collection of the controller's Request object.

After creating the NFLCookies object, the code calls its GetMyTeamIds() method. This gets the persistent cookie and converts it to a string array of team IDs. Then, this code initializes an empty list of Team objects and checks the length of the string array. If there are any team IDs in the string array, this code uses the Where() method of a LINQ query to get data about the user's favorite teams from the database. To do that, the Where() method uses the LINQ Contains() method to determine what team IDs are contained in the string array. Finally, this code stores the list of Team objects, either empty or with the user's favorite teams, in session state. So, for the rest of the session, the team count will be zero or more. As a result, this code block won't run again.

The rest of the Index() action method is the same as in figure 8-11. To review, that code initializes a new TeamListViewModel object with conference, division, and team data, and transfers it to the view via the View() method.

The Add() action method works much like the Add() method shown in figure 9-9. Now, however, it updates the favorite team data in the persistent cookie after it adds the new favorite team to session state.

To update the cookie, the code creates a new NFLCookies object by passing it the Cookies collection of the controller's Response object. Then, this code calls the SetMyTeamIds() method of the NFLCookies object and passes it the updated list of Team objects.

## The Index() action method of the Home controller

```
public ViewResult Index(string activeConf = "all",
                        string activeDiv = "all")
{
    var session = new NFLSession(HttpContext.Session);
    session.SetActiveConf(activeConf);
    session.SetActiveDiv(activeDiv);

    // if no count in session, get cookie and restore fave teams in session
    int? count = session.GetMyTeamCount();
    if (count == null) {
        var cookies = new NFLCookies(Request.Cookies);
        string[] ids = cookies.GetMyTeamIds();

        List<Team> myteams = new List<Team>();
        if (ids.Length > 0) {
            myteams = context.Teams.Include(t => t.Conference)
                .Include(t => t.Division)
                .Where(t => ids.Contains(t.TeamID)).ToList();
        }
        session.SetMyTeams(mytteams);
    }
    // rest of code same as figure 8-11
}
```

## The Add() action method of the Home controller

```
[HttpPost]
public RedirectToActionResult Add(TeamViewModel model)
{
    model.Team = context.Teams
        .Include(t => t.Conference).Include(t => t.Division)
        .Where(t => t.TeamID == model.Team.TeamID)
        .FirstOrDefault();

    var session = new NFLSession(HttpContext.Session);
    var teams = session.GetMyTeams();
    teams.Add(model.Team);
    session.SetMyTeams(teams);

    var cookies = new NFLCookies(Response.Cookies);
    cookies.SetMyTeamIds(teams);

    TempData["message"] = $"{model.Team.Name} added to your favorites";

    return RedirectToAction("Index",
        new { ActiveConf = session.GetActiveConf(),
              ActiveDiv = session.GetActiveDiv() });
}
```

## Description

- The Index() action method uses the cookie collection of the Request object to get the user's favorite teams from a persistent cookie. Then, it stores them in session state.
- The Add() action method uses the cookie collection of the Response object to update the data in the persistent cookie. But first, it updates session state.

---

Figure 9-16 The Home controller

## The updated Favorites controller

---

Figure 9-17 shows the Delete() action method of the Favorites controller. This action method works like the Delete() action method presented in figure 9-12. However, it also removes the persistent cookie that holds the IDs of the user's favorite teams after it removes those teams from session state.

To delete the cookie, the code creates a new NFLCookies object and passes it the Cookies collection of the controller's Response object. Then, this code calls the RemoveMyTeamIds() method of the NFLCookies object.

## The Delete() action method of the Favorites controller

```
[HttpPost]
public RedirectToActionResult Delete()
{
    var session = new NFLSession(HttpContext.Session);
    var cookies = new NFLCookies(Response.Cookies);

    session.RemoveMyTeams();
    cookies.RemoveMyTeamIds();

    TempData["message"] = "Favorite teams cleared";

    return RedirectToAction("Index", "Home",
        new {
            ActiveConf = session.GetActiveConf(),
            ActiveDiv = session.GetActiveDiv()
        }
    );
}
```

### Description

- The Delete() action method of the Favorites controller removes the persistent cookie that stores the IDs of the favorite teams. But first, it removes the favorite teams from session state.

---

Figure 9-17 The Favorites controller

## Perspective

---

The goal of this chapter is to show you how to use ASP.NET Core MVC to work with session state and cookies. If you understand the NFL Teams apps presented at the end of this chapter, you're ready to develop web apps that use session state and cookies. In addition, you should have a solid understanding of when to use TempData, which was presented in the previous chapter, and when to use session state or cookies.

If you develop a production website, it's a good idea to meet the General Data Protection Regulation (GDPR) requirements developed by the European Union (EU). To do that, the users of your website need to consent to your privacy and cookie policy before your website can send non-essential cookies to the browser. Typically, a website gets consent from a user by providing an alert that allows the user to click on an Accept button.

Fortunately, ASP.NET Core provides APIs to help you meet the EU's GDPR requirements. The official documentation for ASP.NET Core explains this in detail and provides sample code that you can use. In short, you need to modify the code in the Startup.cs file for an app to configure cookie policy options so consent is required for non-essential cookies. Then, you need to modify the layout file so it contains HTML that displays the alert and JavaScript that allows users to accept your privacy and cookie policy. If they accept this policy, your website can use non-essential cookies. Otherwise, your website can only use essential cookies.

## Terms

---

state	wrapper class
stateless protocol	serialize
cookie	deserialize
session state	session cookie
JSON (JavaScript Object Notation)	persistent cookie

## Summary

---

- *State* refers to the current status of the properties, variables, and other data maintained by an app for a single user.
- HTTP is a *stateless protocol*. That means that it doesn't keep track of state between round trips. Once a browser makes a request and receives a response, the app terminates and its state is lost.
- A *cookie* is a key/value pair that's passed to the browser in an HTTP response and stored in the browser or on disk. Then, the cookie is passed back to the server with each subsequent HTTP request.
- *Session state* works by having the server store the session data for a user and by using a cookie to associate the user's browser with the correct session data.

- *JSON (JavaScript Object Notation)* is a data format that facilitates the transfer of data.
- To *serialize* a .NET object to a JSON string, you can use the open-source Newtonsoft.Json.NET library. Later, you can *deserialize* the object by using this library to convert its JSON string back into a .NET object.
- A *session cookie* is stored in the browser's memory and exists only for the duration of the browser session.
- A *persistent cookie* is stored on the user's disk and is retained until the cookie's expiration date or until the user clears the cookie.

## Exercise 9-1 Store the user's name in session state

In this exercise, you'll review the NFLTeams app presented in this chapter and update it so it can store the user's name in session state.

### Review the NFLTeams app

1. Open the Ch09Ex1NFLTeams web app in the ex\_starts directory.
2. Run the app and make sure that you can add and clear favorites and that all pages except the Favorites page display the number of favorites in the badge.

### Update the models and add a controller

3. In the Models folder, open the NFLSession class and add methods that can get and set the user's name.
4. Open the TeamListViewModel class and update it so it includes a UserName property that can get and set the user's name.
5. In the Controllers folder, add a new controller class named NameController.
6. In the Name controller, add an Index() action method that displays the Name/Index view. This action method should work like the Index() action method of the Favorites controller.
7. In the Name controller, add a Change() action method that gets the user's name from the TeamListViewModel object, sets that name in session state, and redirects to the Home page like this:

```
[HttpPost]
public RedirectToActionResult Change(TeamListViewModel model)
{
    var session = new NFLSession(HttpContext.Session);
    session.SetName(model.UserName);
    return RedirectToAction("Index", "Home", new {
        ActiveConf = session.GetActiveConf(),
        ActiveDiv = session.GetActiveDiv()
    });
}
```

## Create the Name/Index view

8. In the Views folder, create a Name folder.
9. In the Name folder, create a view named Index that defines a page that can change a user's name. This view should use a TeamListViewModel object as its model, and it should include a form that calls the Change() action method of the Name controller like this:

```
<form asp-action="Change" method="post">
    <input asp-for="@Model.UserName" placeholder="Type name here"
        class="mb-3" /><br />
    <!-- you can code a link to the Home page here -->
    <button type="submit" class="btn btn-primary">
        Change Name
    </button>
</form>
```

10. Note that the asp-for tag helper binds the UserName property to the view model. This makes it easy for the Change() action method to get the user's name from the model.

## Update the Favorites view and the layout

11. Open the Favorite/Index view. Before the Clear Favorites button, add some code that displays a link that calls the Index() action of the Name controller:

```
<a asp-action="Index" asp-controller="Name" class="btn btn-primary">
    Change Name
</a>
```

12. In the Razor code block, add a statement that creates an NFLSession object.
13. Modify the `<h2>` element so it displays "My Favorites" if the user's name is not stored in session state or "Mary's Favorites" if a name of Mary is stored in session state. To do that, you can use a Razor expression within the `<h2>` element like this:

```
<h2>@(session.GetName() == null ? "My" : session.GetName() + "'s")
Favorites</h2>
```

14. Open the layout for this app (Views/Shared/\_Layout).
15. Modify the `<h5>` element so it displays "My Favorite Teams" if a name is stored in session state or "Mary's Favorite Teams" if a name of Mary is stored in session state.
16. Run the app. When it displays the Home page, the second heading should say "My Favorite Teams".
17. Navigate to the Favorites page and use the Change Name button to change the user's name to John. When you do, you should be redirected to the Home page and the second heading should say "John's Favorite Teams". In addition, if you navigate back to the Favorites page, its heading should say "John's Favorites".

# 10

## How to work with model binding

In chapter 2, you learned that if you post a strongly-typed view to an action method with a parameter of the same type, MVC automatically populates the parameter object with values from the view. This is known as *model binding*, and this chapter describes how it works in more detail.

<b>An introduction to MVC model binding.....</b>	<b>360</b>
How to use controller properties to retrieve GET and POST data.....	360
How to use model binding to retrieve GET and POST data .....	362
How to use model binding with complex types .....	364
<b>Model binding in the NFL Teams app.....</b>	<b>366</b>
An action method that binds to primitive types .....	366
An action method that binds to complex types .....	366
<b>More skills for binding data.....</b>	<b>368</b>
Two ways to code a submit button.....	368
How to use a submit button to post a name/value pair.....	370
How to post an array to an action method.....	372
How to control the source of bound values .....	374
How to control which values are bound .....	376
<b>The ToDo List app .....</b>	<b>378</b>
The user interface .....	378
The entity classes .....	380
The database context class.....	382
A utility class for filtering.....	382
The Home controller.....	384
The layout .....	388
The Home/Index view .....	388
The Home/Add view.....	392
<b>Perspective .....</b>	<b>394</b>

## An introduction to MVC model binding

---

In a web app, you often need to retrieve values that have been sent by the browser to the server in either a GET or a POST request. To start, you'll learn how to do that manually by using some of the properties of the Controller class. Then, you'll learn how to use model binding to do the same thing. This leads to simpler code that's easier to maintain. Finally, you'll learn how to use model binding with complex types.

### **How to use controller properties to retrieve GET and POST data**

---

The first table in figure 10-1 presents two properties of the Controller class. The Request property represents the HTTP request that's been sent from the browser to the server, and the RouteData property represents the MVC route for the current request.

The second table presents two properties of the Request property. The Query property is a dictionary that holds all the values in the URL's query string. The Form property is a dictionary that holds all the values in the body of a POST request.

The third table presents one of the properties of the RouteData property. The Values property is a dictionary that holds all the route data for the current request. The first two values in this dictionary are the name of the current controller and the name of the current action method. After that, it contains any named route parameters.

Below these tables, the code examples show how to use these properties to retrieve data. The first example shows a URL with a query string parameter with a name of page and a value of 2.

The second example shows an action method that uses the Query property of the Request property to retrieve that value from the query string parameter. Here, the name of the parameter (page) is the key that's used to get the associated value from the dictionary.

The third example shows form data from a POST request that has a parameter with a name of firstname and a value of "Grace". This data looks similar to query string data, but it's appended to the body of the HTTP request, not to the URL.

The fourth example has an action method that uses the Form property of the Request property to retrieve that value from the form data. Once again, the name of the parameter (firstname) is the key that's used to get the associated value from the dictionary.

The fifth example has a URL with a value of "all" for its third segment. With the default route, this segment is a route parameter with a name of id.

The sixth example has an action method that uses the Values property of the RouteData property to retrieve the value from that route parameter. Once again, the code uses the name of the route parameter (id) as the key to get the associated value from the dictionary.

## Two properties of the Controller class

Property	Description
<b>Request</b>	Represents the HTTP request sent from the browser to the server.
<b>RouteData</b>	Represents the MVC route for the current request.

## Two properties of the Request property

Property	Description
<b>Query</b>	A dictionary that holds the query string parameters in the URL.
<b>Form</b>	A dictionary that holds the form values in the body of a POST request.

## One property of the RouteData property

Property	Description
<b>Values</b>	A dictionary that holds the route data for the current request, including the current controller, action method, and named route parameters.

### A URL with a query string parameter

`https://localhost:5001/Home/Index?page=2`

### An action method that retrieves the value of the query string parameter

```
public IActionResult Index() {
    ViewBag.Page = Request.Query["page"];
    return View();
}
```

### Form data in the body of a POST request

`firstname=Grace`

### An action method that retrieves the form data

```
public IActionResult Index() {
    ViewBag.FirstName = Request.Form["firstname"];
    return View();
}
```

### A URL with a value for the id parameter of the default route

`https://localhost:5001/Home/Index/all`

### An action method that retrieves the value of the route parameter named id

```
public IActionResult Index() {
    ViewBag.Id = RouteData.Values["id"];
    return View();
}
```

## Description

- An HTTP request can pass data from the browser to the server in the query string of the URL or in the body of a POST request.
- MVC can pass data from the browser to the server in the route segments of the URL.
- The Controller class provides several properties that allow you to access this data.

---

Figure 10-1 How to use controller properties to retrieve GET and POST data

## How to use model binding to retrieve GET and POST data

---

Although you can access form, route, and query string data directly as shown in the previous figure, there are many drawbacks to this approach. First, the references to the Request and RouteData properties can get repetitive. Second, you have to manually type case-sensitive key names that aren't checked by IntelliSense or the compiler and that must match the parameter and route names. As a result, they are prone to typos that you won't discover until runtime.

Third, the values from the form, route, or query string are strings. As a result, you must cast them to the type that's needed by the action method. Fourth, you need to know where you're getting your values from, and if that changes, you need to change your code. For example, if you change from a query string parameter to a route parameter, you need to update the code in the action method that retrieves that value.

Fortunately, MVC *model binding* fixes these problems. Model binding works by automatically mapping named request parameters to the names of an action method's parameters. As a result, you don't need to explicitly retrieve the data. This is shown by the first example in figure 10-2.

In addition, you don't need to specify whether the data is in the form, route, or query string. That's because MVC checks for the three types of data that can be passed to an action method. So, if you change how you pass data to the method, you don't have to change your code. For example, you can update your code to use a route parameter instead of a query string parameter.

When using model binding, MVC looks for a parameter name that matches the name of an action method parameter in a specific order. First, it looks in the body of the POST request. If it isn't there, it looks in the route values provided by MVC routing. If it isn't there, it looks in the URL's query string. When it finds the name it's looking for, it binds the associated value to the action method parameter and skips any remaining checks.

MVC model binding is case insensitive. This means that an action method parameter named ID or Id matches a route parameter named id.

If MVC doesn't find a name that matches the parameter name, it sets the value of the action method parameter to the default value for its data type. For instance, it sets a string parameter to null and an int parameter to 0.

A final benefit is that MVC automatically casts a value to the type of the action method parameter. Sometimes, though, the value can't be cast. For instance, MVC can't cast the value "three" to an int. When this happens, MVC doesn't throw an exception. Instead, it adds an error message to the ModelStateDictionary property of the Controller class. As a result, you can check the IsValid property of the controller's ModelState property to make sure the model binding completed successfully. You'll learn more about that in the next chapter.

## An action method that uses model binding to get a string value

```
public IActionResult Index(string id)
{
    ViewBag.Id = id;
    return View();
}
```

## Three types of data this action method can retrieve

### A form parameter in the body of a POST request

```
id=2
```

### A route parameter

```
https://localhost:5001/Home/Index/2
```

### A query string parameter

```
https://localhost:5001/Home/Index?id=2
```

## The order in which MVC looks for data to bind to a parameter

1. The body of the POST request.
2. The route values in the URL.
3. The query string parameters in the URL.

## The benefits of model binding

- You don't have to write repetitive code to retrieve values.
- MVC automatically casts the value to match the data type of the action method parameter.
- Model binding is not case sensitive.
- You can change how you pass data to an action without having to change its code.

## Description

- *Model binding* in MVC automatically maps data in the HTTP request to an action method's parameters by name.
- MVC looks for parameter names in the order listed above. When it finds a name, it binds its associated value and moves on to the next parameter, if any. In other words, it doesn't continue looking once it finds a value that matches a parameter name.
- If there's no request parameter name that matches the name of an action method parameter, MVC binds the default value of the data type.
- If a value can't be cast to the correct type, MVC doesn't throw an error. Instead, it adds an error message to the ModelState property of the Controller class as described in chapter 11.

---

Figure 10-2 How to use model binding to retrieve GET and POST data

## How to use model binding with complex types

The last figure showed how to use model binding with a single primitive type, a string parameter named id. Often, that's all you need to do. For example, it's common to have an action method parameter named id bind to the default route parameter named id and for this parameter to be a primitive type like a string or an int.

However, an action method can also accept a complex type like the ToDo class shown in the first example in figure 10-3, and you can use that type for model binding. To help illustrate why it makes sense to bind complex types, the second example in this figure shows how to use model binding with two primitive types that are used to create a complex type. Here, the action method receives a string parameter that contains a description and a DateTime parameter that contains a due date. Then, this action method creates a ToDo object and uses the values it receives to initialize that object.

This code is an improvement over accessing the Form property of the controller's Request property directly. However, you must be sure that the names of the action method parameters match the names posted by the view. Because model binding isn't case sensitive, though, these values don't need to match exactly. For example, the <input> element with a name attribute of duedate in the view in this example is successfully bound to the action method parameter named dueDate.

Also, this action method needs to explicitly initialize the complex ToDo object. In this case, that's not a big problem since the code only initializes two properties of this object. However, if your complex object has dozens of properties that need to be initialized, doing that manually would be tedious and lead to code that's unwieldy and difficult to maintain.

Fortunately, MVC allows you to bind to complex types as shown by the third example. Here, the action method has a single parameter of the ToDo type. As a result, this method no longer needs to create and initialize a ToDo object. That's because MVC automatically initializes the object when you bind to a complex type. To do that, it looks for request or route parameters with the same names as the object properties.

For a complex type to work with model binding, it must have a default constructor, and the properties to be bound must be public and writeable. For example, the ToDo class shown in this figure meets these criteria. Although this class doesn't include a default constructor, which is a constructor without any parameters, the C# compiler will generate one for you if a class doesn't include any constructors.

In this example, the view is strongly typed and uses the asp-for tag helper in the <input> elements to specify the property names of the model object. This provides for IntelliSense and compiler checking, which makes it easier and less error prone to enter these property names.

Note here that the type="text" attribute is included on the <input> element for the DueDate property. That way, a normal text field will be displayed rather than a field that requires the user to enter a date and time.

## The class for a complex type

```
public class ToDo
{
    public int Id { get; set; }
    public string Description { get; set; }
    public DateTime? DueDate { get; set; }
}
```

## An action method and view that bind to primitive types

### The action method

```
[HttpPost]
public IActionResult Add(string description, DateTime dueDate) {
    ToDo task = new ToDo {
        Description = description,
        DueDate = dueDate
    };
    // rest of code
}
```

### The view

```
<form asp-action="Add" method="post">
    <label for="description">Description:</label>
    <input type="text" name="description">
    <label for="duedate">Due Date:</label>
    <input type="text" name="duedate">
    <button type="submit">Add</button>
</form>
```

## An action method and view that bind to a complex type

### The action method

```
[HttpPost]
public IActionResult Add(ToDo task) {
    // rest of code
}
```

### The view

```
@model ToDo
...
<form asp-action="Add" method="post">
    <label asp-for="Description">Description:</label>
    <input asp-for="Description">
    <label asp-for="DueDate">Due Date:</label>
    <input type="text" asp-for="DueDate">
    <button type="submit">Add</button>
</form>
```

## Description

- MVC allows you to bind complex types.
- You can use the asp-for tag helper to set the name attribute of an HTML element.
- MVC automatically initializes an action method parameter that's a complex type. To do that, it looks for values with the same names as the property names of the parameter.

---

Figure 10-3 How to use model binding with complex types

## Model binding in the NFL Teams app

Chapter 9 presented an NFL Teams app that uses MVC routing, TempData, sessions, cookies, and view models to transfer data. Now, you'll take a closer look at how that app uses model binding and how that model binding could be improved.

### An action method that binds to primitive types

Figure 10-4 starts by reviewing the custom route that the NFL Teams app uses. By now, you should be familiar with how this custom route works.

After the custom route, this figure shows the Index() action method of the Home controller. This action method defines two string parameters with default values of “all” whose names match the route parameters of the custom route. Then, it creates a new TeamListViewModel object and initializes it with the parameter values as well as data from the database.

This is an example of an action method that would benefit from binding to a complex type rather than primitive types. That's because this data is assigned directly to the properties of the model object. In addition, if a more significant amount of data was passed to this action method, each parameter would have to be bound individually.

### An action method that binds to complex types

To update the Index() action method to use a complex type, you need to move the default values for the string parameters to the class that defines the complex type. That's why the third example shows the TeamViewModel class after it has been updated so the ActiveConf and ActiveDiv properties have a default value of “all”.

In addition, you need to make sure the property names match the parameter names in the custom route. In this figure, for example, the TeamViewModel base class of the TeamListViewModel class has property names that match the parameter names in the custom route. This makes it easy to update this action method so it binds to the model type.

Once the complex class sets the default values correctly, you can update the Index() action method as shown by the fourth example. Here, the action method has a single TeamListViewModel parameter. As a result, MVC creates a new TeamListViewModel object and populates its ActiveConf and ActiveDiv properties with the values in the activeConf and activeDiv route parameters. If there are no route parameters, those properties are populated with the default values of “all”. After that, the code populates more of the properties of the model object and sends this model to the view.

If you compare the code in the second and fourth examples, you can see that the fourth example is shorter and simpler than the second example. That's because the fourth example lets MVC model binding do the work. In general, you should use this approach unless you have a good reason not to.

## The custom route in the NFL Teams app

```
endpoints.MapControllerRoute(
    name: "custom",
    pattern: "{controller=Home}/{action=Index}/conf/{activeConf}/div/{activeDiv}");
```

## A Home/Index() action method that binds to primitive types

```
public IActionResult Index(string activeConf = "all",
                           string activeDiv = "all")
{
    var model = new TeamListViewModel
    {
        ActiveConf = activeConf,
        ActiveDiv = activeDiv,
        Conferences = context.Conferences.ToList(),
        Divisions = context.Divisions.ToList()
    };

    IQueryable<Team> query = context.Teams;
    // conditional where clauses based on active conference and division
    model.Teams = query.ToList();

    return View(model);
}
```

## The updated base class for the TeamListViewModel class

```
public class TeamViewModel
{
    public Team Team { get; set; }
    public string ActiveConf { get; set; } = "all";      // default value
    public string ActiveDiv { get; set; } = "all";      // default value
}
```

## A Home/Index() action method that binds to a complex type

```
public IActionResult Index(TeamListViewModel model)
{
    model.Conferences = context.Conferences.ToList();
    model.Divisions = context.Divisions.ToList();

    IQueryable<Team> query = context.Teams;
    // conditional where clauses based on active conference and division
    model.Teams = query.ToList();

    return View(model);
}
```

## Description

- The original Home/Index() action method binds to two string parameters that have a default value of “all”. Within this method, the code creates a view model object and uses its parameters to set its active conference and division.
- The updated Home/Index() action method binds to a view model object that specifies default values of “all” for the active conference and division. For this method, MVC automatically creates the model object and sets its active division and conference.

---

Figure 10-4 The model binding for the Home/Index() action method

## More skills for binding data

---

So far, you've learned the basics of model binding in MVC. Now, you'll learn more skills for working with model binding.

### Two ways to code a submit button

---

In addition to using HTML text boxes, drop-down lists, and hidden fields to post data to the server, you can use submit buttons to post data to the server. Figure 10-5 shows how this works.

The first example in this figure shows an action method named Add() that has a single parameter of the Team type. Then, the second and third examples present two strongly-typed views, each with a `<form>` element that posts to the Add() method. Each form contains a single submit button that posts a team ID to the server and binds it to the TeamID property of the Team parameter defined by the Add() action method.

The second example uses an `<input>` element to define the submit button. Since this element has a type attribute of "submit", it appears as a button and submits the form when clicked. Because it's an `<input>` element, it can use the `asp-for` tag helper. This works just as it does for an `<input>` element for a text box. In other words, it sets the name attribute to "TeamID" and the value attribute to the team ID.

The problem with using an `<input>` element as a submit button is that it displays the value attribute as the text of the button. Here, for example, the button displays the TeamID value as the text of the button. So, if the Team model object contains data for the Seattle Seahawks, the button displays "sea". In this case, that's not what you want.

The third example uses a `<button>` element to define the submit button. Since this element has a type attribute of "submit", it submits the form when clicked. However, unlike an `<input>` element, a `<button>` element can't use the `asp-for` tag helper. As a result, you need to set the name and value attributes manually. In other words, you can't count on MVC to automatically make sure all your names match. Because of this, you need to make sure the name attribute matches the name of the action method parameter or parameter property.

The advantage of using a `<button>` element as a submit button is that it doesn't display the value attribute as the text of the button. Instead, the button displays whatever you specify for the content of the `<button>` element. This gives you control over the text that's displayed by the button. In this figure, the button displays the team name. However, you can also use a `<button>` element to display an image such as a logo as shown in the next figure.

## An action method

```
[HttpPost]  
public IActionResult Add(Team team)  
{  
    // action method code  
}
```

## An <input> element for a button that posts a team ID to the action method

```
@model Team;  
...  
<form asp-action="Add" method="post">  
    <input type="submit" asp-for="TeamID" class="btn btn-primary" />  
</form>
```

### How the button looks in the browser



sea

## A <button> element for a button that posts a team ID to the action method

```
@model Team;  
...  
<form asp-action="Add" method="post">  
    <button type="submit" name="TeamID" value="@Model.TeamID"  
            class="btn btn-primary">@Model.Name</button>  
</form>
```

### How the button looks in the browser



Seattle Seahawks

## The <input> element...

- Allows you to use the asp-for tag helper to generate the name and value attributes.
- Automatically displays the value attribute as the text for the button.

## The <button> element...

- Gives you control over the text (or image) that's displayed on the button.
- Requires you to manually code the name and value attributes.
- Requires you to make sure the value in the name attribute matches the name of the action method parameter or property.

## Description

- You can use the name and value attributes of a submit button to POST data.
- To create a submit button, you can use an <input> element or a <button> element.

---

Figure 10-5 Two ways to code a submit button

## How to use a submit button to post a name/value pair

---

Using a submit button to post data to the server can sometimes improve some clunky code. For example, let's say the NFL Teams app used an action method like the one shown in the first example of figure 10-6 and view code like the code shown in the second example. Here, the Razor code loops through all the Team objects in the Teams property of the model object, which is of the TeamListViewModel type.

Within the loop, the code creates a list item that contains a form. Within the form, it creates a button that displays an image for the current team. Below this button, it creates three hidden fields and uses the `asp-for` tag helper to bind each one to a property of the model object.

Unfortunately, this code generates a fair amount of duplicate HTML. For instance, a separate form is used for each team. In addition, the form for each team contains the hidden fields that store the IDs of the active conference and division. However, those values are the same for every team. As a result, if all 32 teams are displayed, this code generates 32 forms containing 64 identical hidden fields.

The third code example fixes these issues by binding the `TeamID` value to a submit button rather than a hidden field. This allows the view to use a single form for all teams, instead of using one form for each team. That in turn allows the hidden fields for the active conference and division IDs to be moved outside the loop. As a result, these two hidden fields are only created once, which is what you want.

Because of these changes, the code in the loop only creates the image buttons for the teams. Here, each image button has a `name` attribute whose value is “`Team.TeamID`”, and a `value` attribute whose value is the ID of the current team. This button name tells MVC to bind to the `TeamID` property that’s nested within the `Team` property.

So, why does the second example need to have a separate form for each team? Well, for model binding to work, the element that holds the team ID needs to be named “`Team.TeamID`”. But, if you have multiple hidden fields with that name in one form, their values all post to the server as an array. That’s a useful feature that you’ll learn more about in the next figure. But that’s not what you want here.

By contrast, the submit buttons in the third example only post the value of the button that’s clicked to the server. As a result, you can code multiple submit buttons in one form even though they all have the same name.

## An action method that binds to a model

```
[HttpPost]
public IActionResult Details(TeamViewModel model)
{
    TempData["ActiveConf"] = model.ActiveConf;
    TempData["ActiveDiv"] = model.ActiveDiv;
    return RedirectToAction("Details", new { ID = model.Team.TeamID });
}
```

## Part of a view that uses a hidden field to post the team ID

```
<ul class="list-inline">
    @foreach (Team team in Model.Teams)
    {
        <li class="list-inline-item">
            <form asp-action="Details" method="post">
                <button type="submit">
                    
                </button>
                <input type="hidden" asp-for="@team.TeamID" />
                <input type="hidden" asp-for="ActiveConf" />
                <input type="hidden" asp-for="ActiveDiv" />
            </form>
        </li>
    }
</ul>
```

## Part of a view that uses a submit button to post the team ID

```
<form asp-action="Details" method="post">
    <input type="hidden" asp-for="ActiveConf" />
    <input type="hidden" asp-for="ActiveDiv" />
    <ul class="list-inline">
        @foreach (Team team in Model.Teams)
        {
            <li class="list-inline-item">
                <button type="submit"
                    name="Team.TeamID" value="@team.TeamID">
                    
                </button>
            </li>
        }
    </ul>
</form>
```

## Description

- The first view needs a separate `<form>` element for each team logo button and it duplicates its hidden fields in every form.
- The second view only needs one `<form>` element for the page and it doesn't duplicate any of its hidden fields.

---

Figure 10-6 How to use a submit button to post a name/value pair

## How to post an array to an action method

When elements in a form have the same name, the values of the elements post to the server as an array. One scenario where this is useful is when you want to filter data by multiple criteria.

The first example in figure 10-7 shows an action method named Filter() that handles POST requests. It has a parameter named filter that's an array of strings.

Below the action method, this figure presents a view with a form that posts to the Filter() action method. This form contains two <select> elements. The first contains several price options, and the second contains several color options.

Crucially, both of these <select> elements have the same name. As a result, when the form is posted, MVC collects the selected value of each <select> element in a string array.

Below the view, an image shows the string array that the Filter() action method receives when the user selects the “Under \$10” and “Blue” options and clicks the Filter button. For this to work, of course, the names of the <select> elements need to match the name of a parameter or one of its properties in the action method signature. In addition, the action method parameter or property needs to be of the IEnumerable type, such as an array or a List<T> type.

## An action method that accepts a string array

```
[HttpPost]
public IActionResult Filter(string[] filter)
{
    // code that does the filtering
    return View();
}
```

## A view that posts a string array to the action method

```
<h3>Filter By:</h3>
<form asp-action="Filter" method="post">
    <!-- make sure each select element has the same name and that it
        matches the action method parameter name -->
    <label>Price</label>
    <select name="filter">
        <option value="all">All</option>
        <option value="lt10">Under $10</option>
        <option value="10to50">$10 to $50</option>
        <option value="gt50">Over $50</option>
    </select>
    <label>Color</label>
    <select name="filter">
        <option>All</option>
        <option>Red</option>
        <option>Blue</option>
        <option>Yellow</option>
        <option>Green</option>
        <option>Purple</option>
    </select>
    <button type="submit">Filter</button>
</form>
```

## The string array the action method receives



## Description

- You can pass an array to an action method by coding multiple HTML elements that have the same name within a form.
- The name of the HTML elements needs to match the name of an action method parameter or property.
- The action method parameter or property needs to be of the IEnumerable type, such as an array or a List<T> type.

---

Figure 10-7 How to post an array to an action method

## How to control the source of bound values

---

As you learned earlier, MVC model binding automatically checks for names in multiple places and in a specific sequence. To start, it checks posted data, then route data, and then query string data. However, in some situations you might want more control over this sequence. For example, you might want MVC to look only in the route. Or, you might want to bind to data that's sent in other ways, such as the headers of the HTTP request or via dependency injection.

You can use the `From` attributes presented in the table in figure 10-8 to control where MVC looks for the values an action method binds to. The first three attributes direct MVC to look in one of the three places it already looks. However, if you use one of these attributes, MVC only looks in that one place and doesn't check the others. For example, if you use the `FromRoute` attribute, MVC doesn't check the posted data first.

The `FromBody` attribute is typically used when a client-side script, such as a JavaScript script, posts JSON data to an action method of a controller. This is common in Web API apps. When you use this attribute, you can only decorate one parameter per action method. However, when you use the other `From` attributes, you can decorate as many parameters per action method as you like.

Below the table, the figure presents some examples of working with the `From` attributes. In the first example, an action method tells MVC to get the value for the `id` parameter from the URL's route segments, and it tells MVC to get the value for the `pagenum` parameter from the URL's query string.

In the second example, an action method tells MVC to get the value for the `agent` parameter from the headers of the HTTP request. However, since an HTTP request doesn't include a header that matches the name of the parameter, this attribute includes a `Name` argument that specifies that MVC should bind to the request header named `User-Agent`.

The third example shows that you can apply `From` attributes to class properties as well as action method parameters. Here, a class named `Browser` has a property named `UserAgent`. This property is decorated with a `FromHeader` attribute that uses a `Name` argument to specify that the header has a name of `User-Agent`. As a result, when you use model binding with the `Browser` class, MVC always looks in the `User-Agent` request header for the value for this property, and doesn't look anywhere else.

## Some of the attributes that specify the source of the value to be bound

Attribute	Tells MVC to retrieve the value...
[FromForm]	from the form parameters in the body of a POST request.
[FromRoute]	from the route parameters of the URL.
[FromQuery]	from the query string parameters of the URL.
[FromHeader]	from the HTTP request header.
[FromServices]	from services that are injected into the app as described in section 3.
[FromBody]	from the body of the HTTP request. This is often used when a client-side script sends JSON data to an action method. This attribute can only decorate one parameter per action method.

## An action method that specifies the source of its parameters

```
public IActionResult Index([FromRoute] string id, [FromQuery] int pagenum)
{
    ViewBag.Id = id;
    ViewBag.Page = pagenum;
    return View();
}
```

## An action method that passes an argument to an attribute

```
public IActionResult Index([FromHeader(Name = "User-Agent")] string agent)
{
    ViewBag.UserAgent = agent;
    return View();
}
```

## A class that applies an attribute to a property

```
public class Browser
{
    [FromHeader(Name = "User-Agent")]
    public string UserAgent { get; set; }
    ...
}
```

## Description

- You can use attributes to control how binding works.
- You can apply attributes to parameters of action methods in a controller or to properties of model classes.
- You can pass arguments to an attribute to further refine its behavior.
- If you specify a From attribute, MVC only looks for the specified value from that source and doesn't check other sources.

---

Figure 10-8 How to control the source of bound values

## How to control which values are bound

In the previous figure, you learned how to control *where* MVC looks when it binds values to the parameters in an action method. In this figure, you'll learn how to control *which* properties of a complex type are bound. To do that, you can use the Bind and BindNever attributes presented in the first table in figure 10-9. Curiously, these attributes are stored in different namespaces as shown by the second table.

The Bind attribute accepts a list of strings. This list tells MVC the names of the properties that can be set during model binding. MVC doesn't bind any properties that aren't in this list, even if values for them are sent to the server. When you code a Bind attribute, you can apply it to a parameter of an action method or to the model itself.

By contrast, the BindNever attribute tells MVC that a property can't be set during model binding. As a result, MVC doesn't bind the specified property, even if a value for it has been sent to the server. When you code a BindNever attribute, you can only apply it to a property of a model, not to a parameter of an action method.

An important thing to know about both of these attributes is that they only prevent MVC model binding from setting the value of a property. You can still write code that manually sets the property, even if you use attributes to tell MVC not to automatically bind a value to that property.

Controlling which properties are set during model binding protects against *mass assignment attacks*, also called *over posting attacks*. In this kind of attack, a malicious user manipulates the data sent to the server to set properties that the app isn't expecting. For example, the Employee model shown in the first example has an IsManager property that the app might use to decide which pages users can view or what actions they can take. If a user manages to set that property to true, the security of your web app could be compromised.

The code below the Employee model presents three ways that you can use the Bind and BindNever attributes to make sure that a malicious user can't set the value of the IsManager property. First, you can decorate the employee parameter in the action method with a Bind attribute. Here, only the Name and JobTitle properties of the Employee object can be set during model binding. However, the IsManager property can still be set in code.

Second, you can decorate the Employee class itself with the Bind attribute. Again, only the Name and JobTitle properties of the Employee object can be set during model binding. If you're sure you never want the unlisted properties to be bound, this technique is optimal. However, if you may want the unlisted properties to be bound occasionally, you're better off using the Bind attribute to decorate the parameter in the action method.

Third, you can decorate the IsManager property itself with the BindNever attribute. This technique is optimal when your model has many properties, and you only want to prevent a few of them from being bound. In that case, listing all the properties in the Bind attribute would be tedious.

## Two attributes that determine which values are bound

Attribute	Description
<code>[Bind(names)]</code>	Allows you to list the names of the properties that can be set during model binding. This attribute can be applied to a model or to a parameter of an action method.
<code>[BindNever]</code>	Indicates that a property should never be set during model binding. This attribute can only be applied to model properties.

## The namespaces of the two attributes

Attribute	Namespace
<code>[Bind]</code>	<code>Microsoft.AspNetCore.Mvc</code>
<code>[BindNever]</code>	<code>Microsoft.AspNetCore.Mvc.ModelBinding</code>

## The Employee class

```
public class Employee {
    public string Name { get; set; }
    public string JobTitle { get; set; }
    public bool IsManager { get; set; }
}
```

## Three ways to make sure the IsManager property is not bound

### With the Bind attribute in the parameter list of an action method

```
[HttpPost]
public IActionResult Index([Bind("Name", "JobTitle")] Employee employee) {
    if (ModelState.IsValid) {
        if (employee.JobTitle == "Boss")
            employee.IsManager = true; // can be set in code
    }
    return View(employee);
}
```

### With the Bind attribute on the class

```
[Bind("Name", "JobTitle")]
public class Employee {
    public string Name { get; set; }
    public string JobTitle { get; set; }
    public bool IsManager { get; set; }
}
```

### With the BindNever attribute on the IsManager property

```
public class Employee {
    public string Name { get; set; }
    public string JobTitle { get; set; }

    [BindNever]
    public bool IsManager { get; set; }
}
```

## Description

- You can use attributes to tell MVC which properties to set during model binding.

Figure 10-9 How to control which values are bound

## The ToDo List app

---

To put the model binding skills you just learned into context, this chapter finishes by presenting a ToDo List app. This app allows users to store a list of tasks in a database. If you haven't already done so, you'll need to run the Update-Database command from the Package Manage Console to create the database before running this app.

### The user interface

---

The ToDo List app consists of the two pages shown in figure 10-10. Here, the first two screens show the Home page, and the third screen shows the Add page.

On first load, the Home page displays all the tasks in the database, ordered by due date. In addition, it highlights any task that's past due and open.

The left column of the Home page provides three drop-down lists that allow users to filter the tasks by category, by due date, and by status. All of the drop-down lists that provide filtering default to the value "All" as shown in the first screen. However, if the user selects one or more filter options and clicks the Filter button, the app only displays the filtered tasks as shown in the second screen. Here, the app filters the tasks to display only those with a category of "Work" and a status of "Open".

The ToDo List app uses the default id route parameter to pass the filter values. To do that, it uses a dash-separated value for the id parameter. For instance, the URL for the second screen is:

```
https://localhost:5001/home/index/work-all-open/
```

This provides filter values of "work" for category, "all" for due date, and "open" for status. Note that this app makes its URL lowercase with a trailing slash by setting the routing options in the Startup.cs file as described in chapter 4.

The Home page provides two buttons for each task. The Completed button allows a user to mark a task as completed, and the Delete button allows a user to delete a task.

The Add page allows a user to add a new task to the database. In this figure, this page displays a validation message that says, "The value 'tomorrow' is not valid for DueDate". That's because the user entered a string value of "tomorrow" for the due date, which can't be converted to the DateTime type that the app expects. When you review the code for this app, you'll see that it doesn't explicitly add data validation for this. Instead, MVC adds this message to the validation messages when it tries to cast the due date value during model binding.

### The Home page with no filtering

## My Tasks

Category:

Due:

Status:

**Add new task**

Description	Category	Due Date	Status	
File home warranty claim	Home	3/1/2019	Open	<input type="button" value="Completed"/> <input type="button" value="Delete"/>
Go to Freddy's	Shopping	7/2/2019	Completed	<input type="button" value="Completed"/> <input type="button" value="Delete"/>
Send thank you cards	Contact	7/31/2019	Open	<input type="button" value="Completed"/> <input type="button" value="Delete"/>
Switch web host	Work	8/20/2019	Open	<input type="button" value="Completed"/> <input type="button" value="Delete"/>

### The Home page after it has been filtered to show open work tasks

Category:

Due:

Status:

**Add new task**

Description	Category	Due Date	Status	
Switch web host	Work	8/20/2019	Open	<input type="button" value="Completed"/> <input type="button" value="Delete"/>

### The Add page with a validation message

## New Task

- The value 'tomorrow' is not valid for DueDate.

Description:

Category:

Due Date:

Status:

---

Figure 10-10 The user interface

## The entity classes

---

Figure 10-11 presents the code for the entity classes of the ToDo List app. Here, the Category class stores data about the category for a task, and the Status class stores data about the status for a task. These classes are used for the Category and Status filters.

The ToDo class holds data about the task the user wants to add, update, or delete. By the way, this class is called ToDo rather than Task to avoid conflicts with the Task class in the System.Threading.Tasks namespace.

The ToDo class defines several properties. The first property it defines is an Id property that's set by the database. The last property it defines is a read-only property named Overdue that indicates whether a task is overdue.

Between these properties, this class defines four public properties that hold data entered by a user. Each of these properties also has a data validation attribute indicating that the field is required. As a result, if the user doesn't enter data for one of these properties, the app displays a validation message. In addition, this class defines two navigation properties that make it easy to get the Category and Status objects that correspond to the current task.

Note that there's no validation attribute for the DueDate property that specifies that it must be a valid date that can be converted to the DateTime type. That's because MVC data binding handles that for you.

Unlike the Category and Status filters, there's no entity class for the Due filter. Instead, the code for the Filters class defines a dictionary with the values that will be displayed in the Due list. You'll see the code for this class in the next figure.

### The Category class

```
public class Category
{
    public string CategoryId { get; set; }
    public string Name { get; set; }
}
```

### The Status class

```
public class Status
{
    public string StatusId { get; set; }
    public string Name { get; set; }
}
```

### The ToDo class

```
public class ToDo
{
    public int Id { get; set; }

    [Required(ErrorMessage = "Please enter a description.")]
    public string Description { get; set; }

    [Required(ErrorMessage = "Please enter a due date.")]
    public DateTime? DueDate { get; set; }

    [Required(ErrorMessage = "Please select a category.")]
    public string CategoryId { get; set; }
    public Category Category { get; set; }

    [Required(ErrorMessage = "Please select a status.")]
    public string StatusId { get; set; }
    public Status Status { get; set; }

    public bool Overdue =>
        Status?.ToLower() == "open" && DueDate < DateTime.Today;
}
```

---

Figure 10-11 The entity classes

## The database context class

---

The first example in figure 10-12 presents the context class that the ToDo List app uses to communicate with the database. This class provides the DbSet properties for working with the ToDo, Category, and Status objects. In addition, its OnModelCreating() method provides the initial values for the Category and Status tables. In short, this context class works similarly to the one presented in chapter 4 for the Movie List app. As a result, you shouldn't have much trouble understanding how it works.

Although it isn't shown here, this app also stores its database connection string in an appsettings.json file. In addition, it uses the Startup.cs file to configure the database context for the app. Again, this works similarly to the Movie List app presented in chapter 4, so you shouldn't have much trouble understanding how it works.

## A utility class for filtering

---

The second example presents a utility class named Filters. The app uses this class to help filter tasks by category, due date, and status.

The constructor for the Filters class accepts a dash-separated string that contains the values for the three filters. To start, the constructor stores this value in a property named FilterString. However, if the string is null, the constructor stores a default value that displays all the tasks in the database. Next, the constructor splits the string at the dashes and loads the values of the resulting array into the CategoryId, Due, and StatusId properties. All of these properties are read-only. As a result, they can only be set by this constructor.

After these four properties, the Filters class defines three read-only Has properties that indicate whether the user wants to filter by one of the filter criteria. These properties work by checking whether a filter value is "all". If not, the user has selected a value to filter by, and the property is set to true. Otherwise, it's set to false.

After the three Has properties, the static Filters class provides four more properties to handle filtering by due date. To start, the DueFilterValues property defines a dictionary whose keys and values are strings. This dictionary holds the values that appear in the Due drop-down list. This code defines this property as a dictionary instead of a list because it provides an easy way to capitalize values for display to the user, and use lowercase for the filter value that MVC retrieves from the id route parameter. That's why the keys here are lowercase and the values are uppercase.

After the DueFilterValues property, the next three properties are read-only Boolean properties that indicate whether the user wants to return due dates in the past, the future, or today. These properties work by checking the value in the Due property.

## The ToDoContext class

```
public class ToDoContext : DbContext
{
    public ToDoContext(DbContextOptions<ToDoContext> options)
        : base(options) { }

    public DbSet<ToDo> ToDos { get; set; }
    public DbSet<Category> Categories { get; set; }
    public DbSet<Status> Statuses { get; set; }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        modelBuilder.Entity<Category>().HasData(
            new Category { CategoryId = "work", Name = "Work" },
            new Category { CategoryId = "home", Name = "Home" },
            new Category { CategoryId = "ex", Name = "Exercise" },
            new Category { CategoryId = "shop", Name = "Shopping" },
            new Category { CategoryId = "call", Name = "Contact" }
        );
        modelBuilder.Entity<Status>().HasData(
            new Status { StatusId = "open", Name = "Open" },
            new Status { StatusId = "closed", Name = "Completed" }
        );
    }
}
```

## The Filters class

```
public class Filters
{
    public Filters(string filterstring)
    {
        FilterString = filterstring ?? "all-all-all";
        string[] filters = FilterString.Split('-');
        CategoryId = filters[0];
        Due = filters[1];
        StatusId = filters[2];
    }

    public string FilterString { get; }
    public string CategoryId { get; }
    public string Due { get; }
    public string StatusId { get; }

    public bool HasCategory => CategoryId.ToLower() != "all";
    public bool HasDue => Due.ToLower() != "all";
    public bool HasStatus => StatusId.ToLower() != "all";

    public static Dictionary<string, string> DueFilterValues =>
        new Dictionary<string, string> {
            { "future", "Future" },
            { "past", "Past" },
            { "today", "Today" }
        };
    public bool IsPast => Due.ToLower() == "past";
    public bool IsFuture => Due.ToLower() == "future";
    public bool IsToday => Due.ToLower() == "today";
}
```

---

Figure 10-12 The database context class and a utility class for filtering

## The Home controller

---

Figure 10-13 presents the Home controller. To start, part 1 presents the private variable for the database context, the constructor, and the first three action methods.

The Index() action method has a single parameter named id of the string type. As you know, MVC can bind this parameter to a POST request parameter named id, to a route parameter named id, or to a query string parameter named id. In practice, though, this action method always binds to the id parameter of the default route that's defined in the Startup.cs file.

To start, the Index() action method creates a new Filters object by passing the id parameter to its constructor. Next, it gets lists of Category and Status objects from the database and the due date filter dictionary from the Filters object. In addition, this action method assigns these objects to the ViewBag. That way, the view can use them to create drop-down lists for filtering.

After storing data in the ViewBag, the Index() action method retrieves the tasks to display from the database. To start, it builds a query for ToDo objects and their related Category and Status objects. Then, it uses the Has properties of the Filters object to add WHERE clauses to the query if the user has selected filter criteria. For the due date filtering, it also uses the Is properties to add the correct WHERE clause for the due date filter. Next, the code completes the query by sorting the results by due date and calling the ToList() method. Finally, it passes the collection of tasks to the view.

The first Add() action method handles GET requests. To do that, it gets lists of Category and Status objects from the database and assigns these collections to the ViewBag. That way, the view can use these collections to create the drop-down lists the user needs to add a new task to the database.

The second Add() action method handles POST requests. This method has a single parameter named task of the ToDo type. Since MVC model binding starts by searching the body of a POST request, and since this action method handles POST requests, the values for this task argument are always posted from the Add page. As a result, the view for the Add page must make sure that the names of its elements match the names of this object.

To start, this Add() action method makes sure there aren't any validation errors, either from the validation attributes in the ToDo class or from MVC's model binding. To do that, it checks the IsValid property of the ModelState property of the Controller class.

If there aren't any validation errors, the code adds the new ToDo object to the Todos context collection and calls the SaveChanges() method to update the database. Then, it redirects the user back to the Home page, where the new task now displays in the list of tasks.

If there are validation errors, it sets the collections for the drop-down lists in the ViewBag and sends the task to the view. This causes the view to display the data the user entered again with validation messages for the invalid entries.

## The Home controller

```
public class HomeController : Controller
{
    private ToDoContext context;
    public HomeController(ToDoContext ctx) => context = ctx;

    public IActionResult Index(string id)
    {
        var filters = new Filters(id);
        ViewBag.Filters = filters;
        ViewBag.Categories = context.Categories.ToList();
        ViewBag.Statuses = context.Statuses.ToList();
        ViewBag.DueFilters = Filters.DueFilterValues();

        IQueryable<ToDo> query = context.Todos
            .Include(t => t.Category).Include(t => t.Status);
        if (filters.HasCategory) {
            query = query.Where(t => t.CategoryId == filters.CategoryId);
        }
        if (filters.HasStatus) {
            query = query.Where(t => t.StatusId == filters.StatusId);
        }
        if (filters.HasDue) {
            var today = DateTime.Today;
            if (filters.IsPast)
                query = query.Where(t => t.DueDate < today);
            else if (filters.IsFuture)
                query = query.Where(t => t.DueDate > today);
            else if (filters.IsToday)
                query = query.Where(t => t.DueDate == today);
        }
        var tasks = query.OrderBy(t => t.DueDate).ToList();
        return View(tasks);
    }

    [HttpGet]
    public IActionResult Add()
    {
        ViewBag.Categories = context.Categories.ToList();
        ViewBag.Statuses = context.Statuses.ToList();
        return View();
    }

    [HttpPost]
    public IActionResult Add(ToDo task)
    {
        if (ModelState.IsValid){
            context.Todos.Add(task);
            context.SaveChanges();
            return RedirectToAction("Index");
        }
        else {
            ViewBag.Categories = context.Categories.ToList();
            ViewBag.Statuses = context.Statuses.ToList();
            return View(task);
        }
    }
}
```

---

Figure 10-13 The Home controller (part 1)

Part 2 of figure 10-13 presents the last two action methods of the Home controller. Both of these methods handle POST requests.

The Filter() action method has a parameter named filter that's an array of strings. MVC binds this parameter to an array of strings that it gets from the POST request. The next figure shows that this array comes from three <select> elements that have a name of filter. The action method takes this array and converts it in to a single dash-separated string. Then, it redirects to the Index() action method and passes the dash-separated string as the value for the id parameter of the default route.

The Edit() action method has a string parameter named id and a ToDo parameter named selected. Here, the id parameter includes the FromRoute attribute that tells MVC to look for its value in a route parameter. This attribute is necessary because the ToDo class also has a property named Id that's sent to this action method in the POST request. Without the FromRoute attribute, MVC would look first in the POST request, find a parameter named Id, bind its value to this parameter, and skip any further checks. With this attribute, though, MVC skips the POST request and looks for the value in the route parameter named id. By contrast, the ToDo parameter doesn't need a FromForm attribute because MVC looks for the property values in the POST request first by default.

The form posts to the Edit() action method when the user clicks the Completed or Delete button. Then, it sends a route parameter named id that contains the filter string that's bound to the id parameter. It also sends a hidden field named Id that contains the task ID value that's bound to the Id property of the ToDo object. These two values are passed regardless of which button the user clicks. However, the form only sends a StatusId that's bound to the StatusID property of the ToDo object when the user clicks the Completed button. As a result, the code uses the StatusId property to determine which action to take.

If the StatusId property is null, the code in the Edit() action method deletes the task. To do that, it passes the ToDo object to the Remove() method. Although this object only contains the Id property, that's all that's needed to identify the task to be removed.

Otherwise, if the StatusId property has a value, the code marks the task as completed. To do that, the Edit() action method begins by storing the statusId value in a variable named newStatusId. Then, it queries the database to populate all properties in the ToDo object. Otherwise, any of the ToDo properties that are null would replace the current database values when the code calls the Update() method. After that, this code replaces the StatusId value from the database with the new one from the form. Finally, it passes the ToDo object to the Update() method.

After the if statement, the Edit() action method calls the SaveChanges() method. This deletes or updates the task in the database. Then, it redirects back to the Index() action method, passing the id parameter of the action method to the id parameter of the route. This preserves any filtering the user did before clicking the Completed or Delete button.

## The Home controller (continued)

```
[HttpPost]
public IActionResult Filter(string[] filter)
{
    string id = string.Join('-', filter);
    return RedirectToAction("Index", new { ID = id });
}

[HttpPost]
public IActionResult Edit([FromRoute] string id, ToDo selected)
{
    if (selected.StatusId == null) {
        context.ToDos.Remove(selected);
    }
    else {
        string newStatusId = selected.StatusId;
        selected = context.ToDos.Find(selected.Id);
        selected.StatusId = newStatusId;
        context.ToDos.Update(selected);
    }
    context.SaveChanges();

    return RedirectToAction("Index", new { ID = id });
}
```

---

Figure 10-13 The Home controller (part 2)

## The layout

---

Figure 10-14 presents the code for the Razor layout of the ToDo List app. This layout works much like other layouts presented in this book. As a result, you shouldn't have much trouble understanding how it works.

## The Home/Index view

---

Figure 10-14 also presents the code for the Home/Index view. To start, part 1 shows the @model directive that binds the view to a collection of ToDo objects, a Razor code block, and a Bootstrap column with the three drop-down lists that let a user filter by category, due date, and status.

The Razor code block contains a helper method named Overdue(). This method accepts a ToDo object and checks its Overdue property. If that property returns true, the method returns the name of a Bootstrap class that's appropriate for changing the background for an overdue warning. Otherwise, it returns an empty string. Later in this view, Razor code uses this method to highlight tasks that are overdue.

The Bootstrap column contains a form that posts to the Filter() action method presented in the last figure. That form contains three <select> elements that all share the same name, filter. This name is also the name of the parameter in the Filter() action method presented in the previous figure. As a result, this form posts the values of these three <select> elements to the Filter() action method as an array of strings.

The Razor code for each <select> element passes four arguments to the constructor of the SelectList class described in chapter 7. The first argument is the collection that the controller transferred in the ViewBag. The SelectList class uses the items in this collection to create the <option> elements for the drop-down list.

The second argument is the name of the property to bind to the value attribute of the option tag, and the third is the name of the property to display to the user. For the <select> element for the due date, these names are "Key" and "Value". That's because that <select> element is bound to a dictionary of key/value pairs, not a list of objects.

The fourth argument comes from the Filters object that the controller transferred in the ViewBag, and it identifies the currently selected option. This is how the <select> elements "remember" what the user selected.

After the Razor code that generates most of the <option> elements is a final <option> element that displays the "All" item. Although this element is coded after the Razor code, the option will be displayed at the top of the list. That's the case with any hardcoded <option> elements.

The Bootstrap column ends with two buttons. The Filter button submits the values of the drop-down lists to the server as an array. The Clear button resets the drop-down lists by reloading the Home page with no id route parameter. This causes the default values of "all" to load in the Filters object, which resets the drop-down lists to "All".

## The layout

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <link href="~/lib/bootstrap/dist/css/bootstrap.min.css" rel="stylesheet" />
    <title>My Tasks</title>
</head>
<body>
    <div class="container">
        <header class="bg-primary text-white text-center">
            <h1 class="m-3 p-3">My Tasks</h1>
        </header>
        <main>
            @RenderBody()
        </main>
    </div>
</body>
</html>
```

## The Home/Index view

```
@model IEnumerable<ToDo>
@{
    string Overdue(ToDo task) => task.Overdue ? "bg-warning" : "";
}
<div class="row">
    <div class="col-sm-2">
        <form asp-action="Filter" method="post">
            <div class="form-group">
                <label>Category:</label>
                <select name="filter" class="form-control"
                    asp-items="@new SelectList(ViewBag.Categories,
                    "CategoryId", "Name", ViewBag.Filters.CategoryId))">
                    <option value="all">All</option>
                </select></div>
            <div class="form-group">
                <label>Due:</label>
                <select name="filter" class="form-control"
                    asp-items="@new SelectList(ViewBag.DueFilters,
                    "Key", "Value", ViewBag.Filters.Due))">
                    <option value="all">All</option>
                </select></div>
            <div class="form-group">
                <label>Status:</label>
                <select name="filter" class="form-control"
                    asp-items="@new SelectList(ViewBag.Statuses,
                    "StatusId", "Name", ViewBag.Filters.StatusId))">
                    <option value="all">All</option>
                </select></div>
            <button type="submit" class="btn btn-primary">Filter</button>
            <a asp-action="Index" asp-route-id=""
                class="btn btn-primary">Clear</a>
        </form>
    </div>
```

---

Figure 10-14 The layout and the Home/Index view (part 1)

Part 2 of figure 10-14 presents the code for the rest of the Home/Index view. Specifically, it shows a Bootstrap column that contains a link to the Add page and a table that displays information about the tasks. Within this table, each task has a Completed button that the user can click to mark the task as completed and a Delete button that the user can click to delete the task.

The body of the table uses a Razor foreach statement to loop through the collection of ToDo objects that is the model for this view. Then, it builds one row for each ToDo object. To start, it passes the current ToDo object to the Overdue() method and stores the result in a string variable named `overdue`. This returns either a Bootstrap class name for marking overdue tasks or an empty string.

After setting the `overdue` variable, the code creates a row with five data columns. The first four columns display information about the tasks such as the description, category, due date, and status. Here, the third and fourth columns use the `overdue` variable to assign a Bootstrap class if the task is overdue.

The fifth column displays the Completed and Delete buttons. This column starts with a `<form>` element that posts to the `Edit()` action method presented in figure 10-13. However, this form uses a tag helper to set the route parameter named `id` to the `FilterString` property of the `Filters` object that's stored in the `ViewBag`. This property contains the dash-separated string that holds the filter values. As a result, this string is sent to the `Edit()` action method as a route parameter.

Within the `<form>` element, the code creates a hidden field to store the `Id` value of the current `ToDo` object. To do that, this code manually sets the `name` and `value` attributes instead of using the `asp-for` tag helper. That's because that tag helper doesn't always work properly when the model object is a collection rather than a single object. To reduce the chances of error, the code uses the C# `nameof` operator.

After the hidden field, the code creates the Completed and Delete buttons. The Completed button works a little like the hidden field in that it sets the `name` and `value` attributes. However, the `name` and `value` are for the `StatusId` of the `ToDo` object. The Delete button, by contrast, doesn't set these attributes. As a result, when a user clicks the Completed button, this form posts both the `Id` and `StatusId` values of the `ToDo` object to the `Edit()` action method. But when a user clicks the Delete button, the form only posts the `Id` value.

So, why does this code use a separate `<form>` element for the Completed and Delete buttons for each task? Couldn't the table that contains the `<button>` elements be coded within a single form instead as shown in figure 10-6? Unfortunately, that won't work if you need to post more than one value to the controller. In this case, the Completed button needs to post the `StatusId` value as well as the `Id` value. On a good note, though, this code example only duplicates the `<form>` element and not any posted values. That's because the value in the hidden field is the `Id` value for a task, which is different for each task.

## The Home/Index view (continued)

```
<div class="col-sm-10">

    <a asp-action="Add"><b>Add new task</b></a>

    <table class="table table-bordered table-striped mt-2">
        <thead>
            <tr>
                <th>Description</th>
                <th>Category</th>
                <th>Due Date</th>
                <th>Status</th>
                <th class="w-25"></th>
            </tr>
        </thead>
        <tbody>
            @foreach (ToDo task in Model)
            {
                string overdue = Overdue(task);
                <tr>
                    <td>@task.Description</td>
                    <td>@task.Category.Name</td>
                    <td class="@overdue">
                        @task.DueDate?.ToShortDateString()</td>
                    <td class="@overdue">@task.Status.Name</td>
                    <td>
                        <form asp-action="Edit" method="post"
                            asp-route-id="@ViewBag.Filters.FilterString"
                            class="mr-2">

                            <input type="hidden"
                                name="@nameof(ToDo.Id)" value="@task.Id" />

                            <button type="submit"
                                name="@nameof(ToDo.StatusId)" value="closed"
                                class="btn btn-primary btn-sm">Completed
                            </button>

                            <button type="submit"
                                class="btn btn-primary btn-sm">Delete
                            </button>

                        </form>
                    </td>
                </tr>
            }
        </tbody>
    </table>
</div>
</div>
```

---

Figure 10-14 The layout and the Home/Index view (part 2)

## The Home/Add view

---

Figure 10-15 presents the code for the Home/Add view. It starts with the @model directive that indicates this view is strongly typed and bound to a ToDo object. After displaying a heading for the view, the code uses a <div> element with a tag helper to display a summary of any validation errors it receives from the controller.

After the summary of validation errors, this view includes a form that posts to the Add() action method presented in figure 10-13. This form contains two <input> elements for text boxes and two <select> elements for drop-down lists that the user can use to enter the data for a task.

All four of these elements use the asp-for tag helper to bind the element to a property of the model. This accomplishes two goals. First, it makes sure the name attributes of the tags match the property names of the Add() action method parameter. Second, if there's a validation error, it redisplays the values the user entered.

In this view, the Razor code for the <select> elements only passes three arguments to the constructor of the SelectList class. The fourth argument that identifies the currently selected option isn't necessary here because that's handled by the asp-for tag helper.

Like the <select> elements in the Home/Index view, each <select> element in this view ends with a hardcoded <option> element. In this case, though, the value of this element is an empty string and the element doesn't contain any content. Since this element appears first in the drop-down list, that means the user will need to select a different option before adding a task.

This view ends by displaying two buttons. The Add button submits the values of the <input> and <select> elements to the server. And the Cancel button, which is a link styled as a button, returns the user to the Home page.

## The Home/Add view

```
@model ToDo

<h2>New Task</h2>

<div asp-validation-summary="All" class="text-danger"></div>

<form asp-action="Add" method="post">
    <div class="form-group">
        <label asp-for="Description">Description:</label>
        <input asp-for="Description" class="form-control">
    </div>

    <div class="form-group">
        <label asp-for="CategoryId">Category:</label>
        <select asp-for="CategoryId" class="form-control"
            asp-items="@new SelectList(ViewBag.Categories,
                "CategoryId", "Name"))">
            <option value=""></option>
        </select>
    </div>

    <div class="form-group">
        <label asp-for="DueDate">Due Date:</label>
        <input type="text" asp-for="DueDate" class="form-control">
    </div>

    <div class="form-group">
        <label asp-for="StatusId">Status:</label>
        <select asp-for="StatusId" class="form-control"
            asp-items="@new SelectList(ViewBag.Statuses,
                "StatusId", "Name"))">
            <option value=""></option>
        </select>
    </div>

    <button type="submit" class="btn btn-primary">Add</button>
    <a asp-action="Index" class="btn btn-primary">Cancel</a>
</form>
```

---

Figure 10-15 The Home/Add view

## Perspective

---

The goal of this chapter is to teach you the details of model binding so you can benefit from it in your ASP.NET MVC apps. Now, if this chapter has succeeded, you should be able to benefit from model binding and control how it works when you need to. As usual, there's always more to learn. Still, this chapter should give you a good foundation for working with model binding.

## Terms

---

model binding  
mass assignment attack  
over posting attack

## Summary

---

- *Model binding* in MVC automatically maps data in the HTTP request to an action method's parameters.
- In a *mass assignment attack*, also known as an *over posting attack*, a malicious user manipulates the data sent to the server to set properties that the app isn't expecting.

## Exercise 10-1 Update the model binding for the ToDo List app

In this exercise, you'll update the ToDo List app presented in this chapter so it binds its views to a view model instead of binding them to entity models.

### Create a view model

1. Open the Ch10Ex1ToDoList app in the ex\_starts directory.
2. In Visual Studio, run the app and review how it works. Make sure to add one or more tasks.
3. In the Models directory, create a new class named ToDoViewModel with the following code:

```
public class ToDoViewModel
{
    public ToDoViewModel()
    {
        CurrentTask = new ToDo();
    }
    public Filters Filters { get; set; }
    public List<Status> Statuses { get; set; }
    public List<Category> Categories { get; set; }
    public Dictionary<string, string> DueFilters { get; set; }
    public List<ToDo> Tasks { get; set; }
    public ToDo CurrentTask { get; set; } // used for Add
}
```

4. Note that the view model provides all the properties necessary to store the data that the app currently stores in the ViewBag.

### Modify the Home/Index() action method so it uses a view model

5. Open the HomeController class.
6. In the Index() action method, create a ToDoViewModel object. Then, modify the code that stores data in the ViewBag so it stores that data in this view model object.
7. Modify the return statement so it passes the view model to the view.

### Bind the view model to the Home/Index view

8. Open the Home/Index view.
9. At the top of the file, change its model from an IEnumerable<ToDo> object to a ToDoViewModel object.
10. Modify the code so it uses the new view model instead of the ViewBag. For example, you can change all instances of ViewBag.Categories to Model.Categories.
11. Modify the code so it uses the new view model instead of the old view model. For example, in the loop, make sure that the tasks come from the Tasks property of the new view model like this:  
`@foreach (ToDo task in Model.Tasks)`
12. Run the app and make sure that the Home page works as it did before.

**Modify the Home/Add() action methods so they work with a view model**

13. Open the HomeController class.
14. In the Add() action method that doesn't accept any arguments, add a statement that creates a ToDoViewModel object.
15. Modify the code that stores data in the ViewBag so it stores that same data in the ToDoViewModel object.
16. Modify the return statement so it passes the view model to the view.
17. In the Add() action method that accepts a ToDo parameter, change the parameter so it uses the ToDoViewModel type and has a name of model.
18. In the if block, modify the code that adds the task so it uses the CurrentTask property of the view model parameter like this:  
`context.Todos.Add(model.CurrentTask);`
19. In the else block, modify the code that stores data in the ViewBag so it stores data in the view model parameter.
20. Modify the return statement so it passes the view model to the view.

**Bind the view model to the Home/Add view**

21. Open the Home/Add view.
22. At the top of the file, change its view model from a ToDo object to a ToDoViewModel object.
23. Modify the code so it uses properties of the ToDoViewModel object instead of the old ToDo model. For example, you can change  
`<label asp-for="Description">`  
to  
`<label asp-for="CurrentTask.Description">`
24. Modify the code so it uses properties of the ToDoViewModel object instead of the ViewBag. For example, you can change all instances of ViewBag.Categories to Model.Categories.
25. Run the app and make sure that the Add page works as it did before.

# 11

## How to validate data

In chapter 2, you learned how to validate data for simple scenarios like making sure a user has entered a value for a field and making sure the value falls within a specified range. Now, this chapter reviews those skills and presents some more advanced skills for validating data, including how to create custom data attributes, and how to make custom validation work on both the client and the server.

<b>How data validation works in MVC .....</b>	<b>398</b>
The default data validation provided by model binding .....	398
How to use data attributes for validation.....	400
A Registration page with data validation .....	402
<b>How to control the user experience .....</b>	<b>404</b>
How to format validation messages with CSS .....	404
How to check validation state and use code to set validation messages ....	406
How to display model-level and property-level validation messages.....	408
How to enable client-side validation.....	410
<b>How to customize server-side validation.....</b>	<b>412</b>
How to create a custom data attribute .....	412
How to pass values to a custom data attribute.....	414
How to check multiple properties.....	416
<b>How to customize client-side validation .....</b>	<b>418</b>
How to add data attributes to the generated HTML .....	418
How to add a validation method to the jQuery validation library .....	420
How to work with remote validation .....	422
<b>The Registration app.....</b>	<b>424</b>
The user interface and CSS .....	424
The Customer and RegistrationContext classes .....	426
The MinimumAgeAttribute class.....	428
The minimum-age JavaScript file.....	428
The Validation and Register controllers.....	430
The layout .....	432
The Register/Index view.....	434
<b>Perspective .....</b>	<b>436</b>

## How data validation works in MVC

The next few figures review some of the skills for working with data validation that were presented in earlier chapters. In addition, they present some new skills for using attributes for data validation.

### The default data validation provided by model binding

Figure 11-1 begins by presenting a model class for a Movie object, a strongly-typed view that's bound to this model, and the action method that this view posts to. The view includes `<input>` tags that use the `asp-for` tag helper to bind them to properties of the model. In addition, it includes a `<div>` tag that uses the `asp-validation-summary` tag helper to display validation messages. This `<div>` tag is assigned to Bootstrap's `text-danger` class so the validation messages appear in red.

The action method accepts a Movie object named `movie`. When the view posts to this method, MVC binds the values it receives from the view to the Movie object. If MVC can't bind a value, it adds a validation message to the `ModelState` property of the Controller class. That's why this action method starts by checking whether the `ModelState` is valid. If it is, no errors occurred during model binding. In this case, the action method redirects to the page that lists the movies.

Otherwise, if errors occurred during model binding, the action method returns the original view. In addition, it passes the Movie object parameter back to the view. This returns the values that the user posted. As a result, the view can redisplay them to the user. Also, since the `ModelState` property contains validation messages, the validation summary `<div>` tag displays those messages in red.

In this figure, the Movie class doesn't decorate its properties with any data attributes. As a result, the code in this figure doesn't check for required fields or check whether a value is in range as described in the next figure.

However, as described in the previous chapter, when MVC performs model binding, it casts the data posted by the view to the parameter type specified by the action method. If this data conversion fails, MVC adds a message to the `ModelState` property. As a result, this code notifies the user about data conversions that fail, even though there are no data attributes in the model.

For example, the screen in this figure shows the view after the user enters a string of "four" in the Rating field. Since MVC can't cast that string to the `int` type, the data conversion fails and MVC adds a message to the `ModelState` property and sets its `IsValid` property to false. As a result, when the action method checks the `IsValid` property, its value is false. This causes the view to display the validation messages and the values the user entered.

You can also avoid this error by removing the `type="text"` attribute from the `<input>` element. Then, the `asp-for` tag helper will generate a number field instead of a text field since the `Rating` property has a data type of `int`.

## The Movie class

```
public class Movie
{
    public string Name { get; set; }
    public int Rating { get; set; }
}
```

## A strongly-typed view that posts a Movie object to an action method

```
@model Movie
...
<div asp-validation-summary="All" class="text-danger"></div>

<form asp-action="Add" method="post" class="form-inline">
    <div class="form-group">
        <label>Name:</label>&nbsp;
        <input asp-for="Name" class="form-control" />
    </div>&nbsp;
    <div class="form-group">
        <label>Rating (1 - 5):</label>&nbsp;
        <input type="text" asp-for="Rating" class="form-control" />
    </div>&nbsp;
    <button type="submit" class="btn btn-primary">Submit</button>
</form>
...
```

## An action method that receives a Movie object from the view

```
[HttpPost]
public IActionResult Add(Movie movie)
{
    if (ModelState.IsValid) {
        /* code to add movie goes here */
        return RedirectToAction("List", "Movie");
    }
    else {
        return View(movie);
    }
}
```

## How it looks in the browser when the rating can't be cast to an int

• The value 'four' is not valid for Rating.

Name:	<input type="text"/>	Rating (1 - 5):	<input type="text" value="four"/>	<input type="button" value="Submit"/>
-------	----------------------	-----------------	-----------------------------------	---------------------------------------

## Description

- When you use model binding and MVC can't cast an HTTP request value to the data type specified by the action method parameter, it adds an error message to the ModelState property of the Controller class.

---

Figure 11-1 The default data validation provided by model binding

## How to use data attributes for validation

Chapter 2 showed how to validate user data by decorating model properties with *data attributes*. This is called *property-level validation*. Now, figure 11-2 reviews these data validation skills and presents some new ones. To start, the first table presents some of the most common data attributes that are stored in the System.ComponentModel.DataAnnotations namespace.

The Required attribute checks that the value of a property is not null. The property that this attribute decorates must be nullable. As a result, for numeric values, the property must use a nullable type like int? or double?.

The Range attribute checks that the value of a property is within a specified range. The property that this attribute decorates must implement the IComparable interface. Fortunately, most C# types (int, double, decimal, string, DateTime, etc.) implement this interface.

The two-argument Range attribute accepts int or double values that specify the minimum and maximum values of the range. However, the three-argument Range attribute also accepts a type argument that indicates the type of the values to compare. For example, the next figure shows how to use this type argument to compare DateTime values. When you use the type argument, the minimum and maximum values must be strings.

The StringLength attribute checks that the value of a string property doesn't exceed a specified number of characters. This attribute accepts an int value that specifies the maximum number of characters.

The RegularExpression attribute checks that the value of a property matches a regular expression (regex) pattern. It accepts a string value that's the pattern to match. Regular expression patterns are beyond the scope of this book, but you can find many good resources and regex libraries online.

The Compare attribute checks that the value of the property matches the value of another property in the model. It accepts a string value that contains the name of the other property.

The Display attribute doesn't check any values. Instead, it specifies how the name of the property should be displayed to the user. That includes not only how it's displayed in a validation message, but how it's displayed in a label that's bound to the property.

The first code example shows the properties of the Movie class decorated with some data attributes. Below that, the screen shows the validation messages that these attributes produce in the browser. This works because each attribute has a default validation message as shown in the second table.

However, sometimes the default message isn't user friendly. For example, the default message for the RegularExpression attribute displays a cryptic regex pattern. This regex pattern might look something like “[a-zA-Z0-9]+”, which is not user friendly. Fortunately, to create user-friendly messages, you can use the ErrorMessage property to add your own validation message to a data attribute as shown by the second code example.

## Common data attributes for validation

Attribute	Description
<b>Required</b>	Makes sure the property value is not an empty string or blank spaces. Numeric data types like int or double must be nullable.
<b>Range(min, max)</b>	Makes sure the property value is within the specified range of values. The property must implement the IComparable interface.
<b>Range(type, min, max)</b>	Makes sure a value of a type other than int or double is within a range of values. The min and max arguments must be strings.
<b>StringLength(length)</b>	Makes sure the string property value doesn't exceed a specific length.
<b>RegularExpression(ex)</b>	Makes sure a property value matches a regex pattern.
<b>Compare(other)</b>	Makes sure the property has the same value as another property.
<b>Display(Name = "n")</b>	Specifies how the field name should be displayed to the user.

## The Movie entity class decorated with data attributes

```
using System.ComponentModel.DataAnnotations;
...
public class Movie
{
    [Required]
    [StringLength(30)]
    public string Name { get; set; }

    [Range(1, 5)]
    public int Rating { get; set; }
}
```

## The view after the user submits invalid data

- The field Name must be a string with a maximum length of 30.
- The field Rating must be between 1 and 5.

Name:  Rating (1 - 5):

## The default validation messages

Attribute	Message
<b>Required</b>	The field [Name] is required.
<b>Range</b>	The field [Name] must be between [minimum] and [maximum].
<b>StringLength</b>	The field [Name] must be a string with a maximum length of [length].
<b>RegularExpression</b>	The field [Name] must match the regular expression '[pattern]'.
<b>Compare</b>	'[Name]' and '[OtherName]' do not match.

## How to replace the default message with a custom message

```
[Required(ErrorMessage = "Please enter a name")]
[StringLength(30, ErrorMessage = "Name must be 30 characters or less.")]
public string Name { get; set; }

[Range(1, 5, ErrorMessage = "Please enter a rating between 1 and 5.")]
public int Rating { get; set; }
```

Figure 11-2 How to use data attributes for validation

## A Registration page with data validation

Figure 11-3 shows how to code a Registration page that validates user input in a variety of ways. The first code example presents the model class for a Customer object that has four properties. This class decorates all four properties with a Required attribute that specifies a custom error message.

Each property also has one or more additional attributes. The Name property has a RegularExpression attribute with a regex pattern that only allows uppercase letters, lowercase letters, and numbers. This attribute includes a user-friendly validation message that describes the requirements of the regex pattern.

The DOB property has a Range attribute that includes a type argument to indicate that the range is of the DateTime type. Then, it passes two string arguments that indicate the minimum and maximum dates for the range. This code sets the maximum value to a date that's in the distant future to make sure the current date is always before the maximum value. Later, this chapter presents some techniques for using code to compare a date to the current date.

Note that the DOB property is of the DateTime? type, not the DateTime type. This is necessary for the Required attribute to work properly.

The Password property has a StringLength attribute that only allows string values of 25 characters or less. This attribute doesn't include a custom validation message because the default message is adequate. In addition, text fields and password fields often prevent the error message from being displayed by not allowing the user to enter more than the specified number of characters.

The Password property also has a Compare attribute that tells MVC to check that the value of this property matches the value of the ConfirmPassword property. This attribute doesn't include a custom validation message. That's because the default message for the Compare attribute is adequate here.

The ConfirmPassword property has a Display attribute that adds a space to the property name. As a result, this property displays as “Confirm Password” in the default validation message of the Compare attribute that decorates the Password property.

The second code example presents the validation <div> and <input> tags in a strongly-typed view that posts a Customer object to an action method. Here, the <input> tags for the passwords are of the password type. This displays bullets for the characters.

The last code example presents the action method that receives the Customer object from the view. To start, this action method checks the IsValid property of the controller's ModelState property. If it's false, the code passes the Customer object to the view. That way, the view can redisplay the data to the user as well as the validation messages as shown by the screen at the bottom of this figure.

## The Customer class

```
public class Customer {
    [Required(ErrorMessage = "Please enter a name.")]
    [RegularExpression("^[a-zA-Z0-9]+$",
        ErrorMessage = "Name may not contain special characters.")]
    public string Name { get; set; }

    [Required(ErrorMessage = "Please enter a date of birth.")]
    [Range(typeof(DateTime), "1/1/1900", "12/31/9999",
        ErrorMessage = "Date of birth must be after 1/1/1900.")]
    public DateTime? DOB { get; set; }

    [Required(ErrorMessage = "Please enter a password.")]
    [StringLength(25)]
    [Compare("ConfirmPassword")]
    public string Password { get; set; }

    [Required(ErrorMessage = "Please confirm your password.")]
    [Display(Name = "Confirm Password")]
    public string ConfirmPassword { get; set; }
}
```

## A validation tag in a strongly-typed view that posts a Customer object

```
<div asp-validation-summary="All" class="text-danger"></div>
...
<input asp-for="Name" class="form-control" />
<input type="text" asp-for="DOB" class="form-control" />
<input type="password" asp-for="Password" class="form-control" />
<input type="password" asp-for="ConfirmPassword" class="form-control" />
```

## An action method that receives a Customer object from the view

```
public IActionResult Index(Customer customer) {
    if (ModelState.IsValid) {
        // code that adds customer to database
        return RedirectToAction("Welcome");
    } else {
        return View(customer);
    }
}
```

## The view after the user submits invalid data

- Name may not contain special characters.
- Date of birth must be after 1/1/1900.
- 'Password' and 'Confirm Password' do not match.
- Please confirm your password.

Name:	<input style="width: 80%;" type="text" value="Mary D!"/>
DOB:	<input style="width: 80%;" type="text" value="1/1/1872"/>
Password:	<input style="width: 80%;" type="password" value="*****"/>
Confirm Password:	<input style="width: 80%;" type="password"/>

Submit

Figure 11-3 A Registration page with data validation

## How to control the user experience

So far, you've learned how to use Bootstrap CSS classes to make messages in a validation summary `<div>` tag display in red. In addition, you've learned how to add your own validation messages to a data attribute. In the next few figures, you'll learn more skills for controlling how MVC displays validation messages to the user.

### How to format validation messages with CSS

When you use data attributes and model binding as described in the previous two figures, MVC automatically adds `data-*` attributes to the `<input>` tags. These `data-*` attributes are an HTML5 feature that allows you to embed data that's specific to an app in your HTML.

The first example in figure 11-4 shows the HTML that MVC emits for an `<input>` tag that's bound to the `Name` property. This HTML includes a `data-val` attribute with a value of "true". In addition, it includes several `data-val-*` attributes that contain information MVC emits for the `Required` and `RegularExpression` attributes that decorate the `Name` property. When data validation fails, MVC also assigns a CSS class named `input-validation-error` to the `<input>` tag as shown in the second example.

The `<div>` tags that use the `asp-validation-summary` tag helper work similarly. For instance, the third example shows that MVC adds a `data-valmsg-summary` attribute to the `<div>` tag. In addition, if validation succeeds, it assigns the `<div>` tag to a CSS class named `validation-summary-valid`. However, if validation fails, MVC assigns the `<div>` tag to a CSS class named `validation-summary-errors` as shown by the fourth example.

You can define style rules for these CSS classes to control how these tags appear to the user. For instance, the last example presents style rules for all three CSS classes. For `<input>` tags that have failed validation, the CSS adds a border that's the same color as the Bootstrap `text-danger` CSS class. In addition, it adds a background color. The screen at the bottom of this figure shows how such an `<input>` tag appears on the page.

The last two style rules are for validation summary `<div>` tags. When validation succeeds (or when the page first loads), the first style rule sets the display to none. This hides the `<div>` tag and prevents it from taking up any space on the page. Then, if you want, you can include a generic message like this and it won't be displayed unless validation fails:

```
<div asp-validation-summary="All" class="text-danger">
    <h4>Please correct the following errors:</h4>
</div>
```

When validation fails, the last style rule removes the bullets from the unordered list items. The screen at the bottom of the figure shows how such a `<div>` tag appears on a page.

### The HTML that MVC emits for an <input> tag bound to the Name property

```
<input type="text" class="form-control"
    data-val="true"
    data-val-regex="Name may not contain special characters."
    data-val-regex-pattern="^[a-zA-Z0-9 ]$"
    data-val-required="Please enter a name."
    id="Name" name="Name" value="" />
```

### The HTML that MVC emits for that <input> tag when data validation fails

```
<input type="text" class="form-control input-validation-error"
    data-val="true"
    data-val-regex="Name may not contain special characters."
    data-val-regex-pattern="^[a-zA-Z0-9 ]$"
    data-val-required="Please enter a name."
    id="Name" name="Name" value="" />
```

### The HTML that MVC emits for a summary of valid data

```
<div class="text-danger validation-summary-valid"
    data-valmsg-summary="true">
    <ul><li style="display: none;"></li></ul>
</div>
```

### The HTML that MVC emits for a summary of invalid data

```
<div class="text-danger validation-summary-errors"
    data-valmsg-summary="true">
    <ul>
        <li>Please enter a name.</li>
        <li>Please enter a date of birth.</li>
    </ul>
</div>
```

### Some CSS styles in the site.css file

```
.input-validation-error {
    border: 2px solid #dc3545;      /* same red as text-danger */
    background-color: #faebd7;      /* antique white */
}

.validation-summary-valid { display: none; }

.validation-summary-errors ul { list-style: none; }
```

### The view after the user submits invalid data

Please enter a name.  
Please enter a date of birth.

Name:

DOB:

### Description

- When performing validation, MVC emits HTML that has CSS classes that you can style to control the appearance of the validation.

Figure 11-4 How to format validation messages with CSS

## How to check validation state and use code to set validation messages

---

The ModelState property of the Controller class has a data type of ModelStateDictionary. This dictionary stores the data that a form posts to the controller. It stores this data as key/value pairs. It also stores any validation messages produced during model binding.

The first table in figure 11-5 shows some of the properties of the ModelStateDictionary class. The Count property returns the number of key/value pairs sent to the server, and the ErrorCount property returns the number of validation messages. For instance, if a form posts three values and one of them produces a validation message, the Count property is 3 and the ErrorCount property is 1.

The IsValid property returns false if there are any validation messages. The Keys property is a collection that contains the names of the form data parameters that were posted. And the Values property is a collection that contains the values of those parameters.

The second table in this figure shows two of the methods of the ModelStateDictionary class. The AddModelError() method adds a validation message. It accepts two string values. The first argument specifies the name of the property that's associated with the validation message. However, if you want to associate the message with the overall model, you can pass an empty string to this argument. The second argument specifies the validation message itself.

The GetValidationState() method returns the validation state of the specified property. It accepts a string value that's the name of the property to check and returns a value of the ModelValidationState enum.

Below the tables, the code example uses these properties and methods to check the DOB property against the current date. Here, the action method accepts a Customer object and starts by checking whether the DOB value passed MVC validation during model binding.

To perform this check, the code uses the nameof operator to get the name of the DOB property and store it in a variable named key. This is less error prone than coding a string literal of “DOB”. Then, the code calls the GetValidationState() method, passes it the name of the DOB property, and compares its return value to the Valid value of the ModelValidationState enum. If they match, the DOB value passed MVC validation and is a date that's later than 1/1/1900 and before 12/31/9999.

If the DOB value passes MVC validation, the code checks if the date is a future date. If so, the code calls the AddModelError() method and adds an error message for the DOB field to the ModelState property. Finally, the code checks the IsValid property and proceeds based on the value of that property.

## Some of the properties of the ModelStateDictionary class

Property	Description
<b>Count</b>	The number of key/value pairs sent to the server and stored in the dictionary.
<b>ErrorCount</b>	The number of validation messages stored in the dictionary.
<b>IsValid</b>	A Boolean value indicating whether any of the key/value pairs are invalid or not validated.
<b>Keys</b>	A collection of the keys of the form data parameters.
<b>Values</b>	A collection of the values of the form data parameters.

## Two of the methods of the ModelStateDictionary class

Method	Description
<b>AddModelError(key, msg)</b>	Adds a validation message to the dictionary and associates it with the property that matches the specified key. To associate a validation message with the overall model, rather than a specific property, specify an empty string as the key.
<b>GetValidationState(key)</b>	Gets the ModelValidationState enum value for the specified property name. Possible values are Valid, Invalid, Unvalidated, and Skipped.

## Code that adds a validation message to the ModelState property

```
using Microsoft.AspNetCore.Mvc.ModelBinding;
...
[HttpPost]
public IActionResult Index(Customer customer)
{
    string key = nameof(Customer.DOB);

    if (ModelState.GetValidationState(key) == ModelValidationState.Valid) {
        if (customer.DOB > DateTime.Today) {
            ModelState.AddModelError(
                key, "Date of birth must not be a future date.");
        }
    }

    if (ModelState.IsValid) {
        // code that adds customer to database
        return RedirectToAction("Welcome");
    }
    else {
        return View(customer);
    }
}
```

### Description

- The Controller class has a property named ModelState of the ModelStateDictionary type. This dictionary stores key/value pairs that represent the form parameters sent to the server and the validation messages associated with those parameters.
- The ModelValidationState enum is in the Microsoft.AspNetCore.Mvc.ModelBinding namespace.

Figure 11-5 How to check validation state and use code to set validation messages

## How to display model-level and property-level validation messages

In the last figure, you learned how to associate a validation message with either a specific property or the overall domain model. But what difference does that make? Well, MVC has features that let you display validation messages differently based on this association.

The first table in figure 11-6 shows two MVC tag helpers that you can use with data validation. So far, the examples in this book have used the `asp-validation-summary` helper. This helper targets the `<div>` tag and displays one or more validation messages for one or more properties in an unordered list (a `<ul>` tag). It accepts a value of the `ValidateSummary` enum that's described in the second table.

The `asp-validation-for` tag helper, by contrast, targets the `<span>` tag and displays a single validation message for a specific property. If there's more than one validation message for a property, it displays the first one. It accepts a string value that's the name of the property.

The second table presents the values of the `ValidateSummary` enum. So far, the examples in this book have used the `All` value. This value tells MVC to display all the validation messages in the `ModelState` dictionary. By contrast, the `ModelOnly` value tells MVC to display only those messages that are associated with the model. That is, it tells MVC to display only messages that have a key value of an empty string, not any messages that have a key value of a property name.

The code below the tables shows how to display model-level messages in one place and property-level messages in another. This action method accepts a `Customer` object as an argument. As usual, it uses the controller's `ModelState` property to check whether validation has succeeded. If not, the code adds another message and associates it with the model.

The view has a `<div>` tag that uses the `asp-validation-summary` tag helper. This time, though, the code sets the tag helper to "ModelOnly" rather than "All". This means that the `<div>` tag only displays model-level messages like the one added by the action method in this figure.

Below the `<div>` tag, the view includes the `<input>` tags for the form. Here, each `<input>` tag is followed by a `<span>` tag that uses the `asp-validation-for` tag helper. This HTML sets the value of each tag helper to the same property name that's bound to the corresponding `<input>` tag. This means that each `<span>` tag displays the first property-level message for the specified property.

The screen at the bottom of the figure shows what the browser displays when validation fails. Here, the page displays the model-level message that notifies the user that the form contains errors above the form. Then, the property-level messages that describe how to fix the invalid data for each entry display to the right of each text box.

## Two tag helpers used with data validation

Tag Helper	Description
<code>asp-validation-summary</code>	Displays the validation messages in the ModelState property. Accepts a value of the ValidateSummary enum to determine which messages to display. Targets the <div> tag.
<code>asp-validation-for</code>	Displays the first validation message in the ModelState property for the specified property. Targets the <span> tag.

## The values of the ValidateSummary enum

Value	Description
<code>All</code>	Displays all the validation messages in the ModelState property.
<code>ModelOnly</code>	Displays only those messages in the ModelState property that are associated with the model. In other words, only those messages whose key is an empty string rather than a property name.
<code>None</code>	Displays no messages.

## An action method that adds a model-level validation message

```
[HttpPost]
public IActionResult Index(Customer customer) {
    if (ModelState.IsValid) {
        // code that adds customer to database
        return RedirectToAction("Welcome");
    } else {
        ModelState.AddModelError("", "There are errors in the form.");
        return View(customer);
    }
}
```

## Part of a view that displays both model-level and property-level messages

```
<div asp-validation-summary="ModelOnly" class="text-danger"></div>

<form asp-action="Index" method="post">
    <div class="form-group row">
        <div class="col-sm-2"><label>Name:</label></div>
        <div class="col-sm-4">
            <input asp-for="Name" class="form-control" />
        </div>
        <div class="col-sm-6">
            <span asp-validation-for="Name" class="text-danger"></span>
        </div>
    </div>
    ...

```

## The form in the browser after validation fails

There are errors in the form.

Name:  Please enter a name.

DOB:  Please enter a date of birth.

Figure 11-6 How to display model-level and property-level validation messages

## How to enable client-side validation

---

So far, this book has shown how to validate data by posting it to the server, checking it on the server, and returning validation results to the client. However, it's generally considered a good practice to validate data on the client before sending it to the server. This can improve the responsiveness of your app significantly since it doesn't require an unnecessary round trip to the server.

Fortunately, it's easy to enable client-side validation in an MVC app. All you need to do is to add the jQuery libraries shown in figure 11-7 as described in chapter 3. These libraries use `data-val-*` attributes to perform data validation without making a round trip to the server. This works well with MVC since it automatically emits `data-val-*` attributes when you use server-side validation as described in this chapter.

After you add these libraries, you can include them with your view using `<script>` tags. When you code these tags, you must code them in the order shown in this figure. This example adds these `<script>` tags to the end of the `<head>` tag. However, many programmers prefer to add the jQuery libraries at the end of the `<body>` tag as shown later in this chapter.

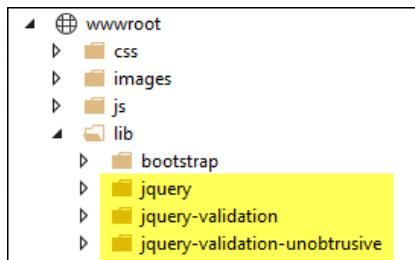
There are some caveats to using client-side validation in your MVC app. First, client-side validation only works correctly with property-level validation, not with model-level validation. That's because there's no round trip to the server with client-side validation, so the `ModelState` dictionary doesn't contain any model-level validation messages. As a result, if the user doesn't enter valid data, the form doesn't submit and the app doesn't display any model-level messages explaining why the form didn't submit. This leaves the user wondering why the form didn't submit. Because of that, it's best to use property-level validation with client-side validation.

Second, if there's custom validation on the server, such as the DOB check shown in figure 11-5, validation can run twice. In other words, after the client-side validation passes, the form posts to the server, and the server-side validation runs. This can lead to a process where the user gets validation messages in two different steps, which some users might not like. Whenever possible, it's best to let your users know in one step everything they need to do to fix their data entry.

Third, not all of the MVC data attributes work properly with the jQuery validation libraries. In particular, the `Range` attribute doesn't work well with dates. For example, in the screen at the bottom of this figure, the user has entered a date of 1/1/1971, which is after 1/1/1900, but the client-side validation is telling the user otherwise. In a case like this, you need to use custom validation as shown later in this chapter.

When working with validation, keep in mind that client-side validation *is not* a substitute for server-side validation. That's because a user's browser might have JavaScript turned off, or a malicious user might tamper with the JavaScript. As a result, you should think of client-side validation as a convenience for your users. You should always validate user input on the server as well.

## The jQuery libraries that download by default with the MVC template



## The jQuery libraries in the head section of a Layout view

```

<head>
  <meta name="viewport" content="width=device-width" />
  <link href="~/lib/bootstrap/dist/css/bootstrap.min.css"
        rel="stylesheet" />
  <link href("~/css/site.css" rel="stylesheet" />
  <script src="~/lib/jquery/dist/jquery.min.js"></script>
  <script src="~/lib/jquery-validation/dist/jquery.validate.min.js">
  </script>
  <script src="~/lib/jquery-validation-unobtrusive/
                jquery.validate.unobtrusive.min.js">
  </script>
  <title>@ViewBag.Title</title>
</head>
  
```

## Some caveats to client-side validation

- It only works correctly with property-level validation, not model-level validation.
- Any server-side validation doesn't run until all client-side validation passes. This can lead to a 2-step process that may annoy some users.
- Not all data annotations work properly on the client. In the example below, for instance, the Range annotation for the DOB field isn't working as it should.

## The form in the browser after validation fails

Name:  Please enter a name.

DOB:  1/1/1971 Date of birth must be after 1/1/1900.

## Description

- To enable client-side validation, add the jQuery, jQuery validation, and jQuery unobtrusive validation libraries to your app in the order shown above.
- Client-side validation is optional and should be thought of as a convenience for the user. For security reasons, you should always perform server-side validation too.

---

Figure 11-7 How to enable client-side validation

## How to customize server-side validation

The built-in data attributes that MVC provides are often all you need to validate the data for an app. However, you may occasionally need to create custom validation for an app as shown in the next few figures.

### How to create a custom data attribute

One way to customize validation is to create a custom data attribute. Then, you can use it just like the built-in data attributes.

The first table in figure 11-8 presents three of the classes that you use to create a custom data attribute. The `ValidationAttribute` class is the base class for data attributes. The `ValidationContext` class provides information about the context in which the validation takes place, including information about the property being validated. And the `ValidationResult` class contains information about the result of the validation.

The second table presents the virtual `IsValid()` method of the `ValidationAttribute` class. This method accepts two arguments. The first is an object that's the value to check, and the second is a `ValidationContext` object. This method returns a `ValidationResult` object.

The third table presents a constructor and a static field of the `ValidationResult` object. The constructor accepts a string value that specifies the validation message to associate with the property being checked. The static `Success` field indicates that the validation was successful.

To create a custom data attribute, you code a class that inherits the `ValidationAttribute` class and overrides its `IsValid()` method as shown by the `PastDateAttribute` class presented in this figure. It's a convention to include a suffix of *Attribute* at the end of the class name. However, you don't include that suffix when you decorate a model property.

The `IsValid()` method of the `PastDateAttribute` class starts by checking whether the value it receives is a date. If so, it casts that value to a `DateTime` object. Then, it checks whether the date is in the past. If so, the `IsValid()` method returns the `Success` field of the `ValidationResult` class.

If the date isn't in the past, the `IsValid()` method returns a new `ValidationResult` object that contains an error message. But first, it builds the message to pass to the constructor of the `ValidationResult` object. To do that, it checks the `ErrorMessage` property of the base class. If it's null, no custom validation message has been set. In that case, the code creates a default validation message. To do that, the code uses the `ValidationContext` parameter to get the display name of the property being checked.

The last two code examples in this figure show how to use the `PastDate` data attribute with a default or custom validation message. This works just like the built-in data attributes that you learned about earlier in this chapter.

### Three classes used to create a custom data attribute

Class	Description
<code>ValidationAttribute</code>	The base class for a custom data attribute.
<code>ValidationContext</code>	Describes the context for the validation.
<code>ValidationResult</code>	Contains data that represents the validation results.

### A virtual method of the ValidationAttribute class

Virtual method	Description
<code>IsValid(object, context)</code>	Checks whether a value is valid. Accepts the value to check and a ValidationContext object. Returns a ValidationResult object.

### A constructor and a field of the ValidationResult class

Constructor	Description
<code>ValidationResult(string)</code>	Creates an object with the validation message.
Static field	Description
<code>Success</code>	Indicates that validation was successful.

### A custom data attribute that checks if a date is in the past

```
public class PastDateAttribute : ValidationAttribute
{
    protected override ValidationResult IsValid(object value,
        ValidationContext ctx)
    {
        if (value is DateTime) {
            DateTime dateToCheck = (DateTime)value;
            if (dateToCheck < DateTime.Today) {
                return ValidationResult.Success;
            }
        }

        string msg = base.ErrorMessage ??
                    $"{ctx.DisplayName} must be a valid past date.";
        return new ValidationResult(msg);
    }
}
```

### Code that uses the PastDate attribute with the default validation message

```
[PastDate]
public DateTime? DOB { get; set; }
```

#### How it looks in the browser

DOB:	<input type="text" value="1/1/2050"/>	DOB must be a valid past date.
------	---------------------------------------	--------------------------------

### Code that uses the PastDate attribute with a custom validation message

```
[PastDate(ErrorMessage = "Please enter a date of birth in the past.")]
public DateTime? DOB { get; set; }
```

Figure 11-8 How to create a custom data attribute

## How to pass values to a custom data attribute

Sometimes a data attribute needs additional information to validate the value of a property. For instance, the Compare attribute needs the name of the property to compare, and the Range attribute needs the minimum and maximum values in the range. To enable this in your own custom data attribute class, you can use the constructor and public properties.

Figure 11-9 presents a class for a custom data attribute named YearsFromNow. This attribute checks that the value of a property is a valid past or future date that's within a specified number of years from the current date.

Like the class in the last figure, the YearsFromNowAttribute class inherits the ValidationAttribute base class and overrides its IsValid() method. However, it also has a constructor that accepts an int value and assigns it to a private variable named numYears. In addition, it has a property named IsPast that accepts a Boolean value. This property has a default value of false.

The IsValid() method in this figure starts by making sure that the value it receives is a date. If so, it casts this value to a DateTime object. After that, it assigns the current date to a DateTime variable named now and declares another DateTime variable named from. These from and now variables define the range of valid dates.

The code then uses the IsPast property to calculate the from variable. If the date is in the past, the code creates a new DateTime value for January 1 of the current year and assigns it to the from variable. Then, this code subtracts the number of years in the numYears variable. So, if the current year is 2020 and the value in numYears is 25, the from date would be 1/1/1995.

Conversely, if the date is in the future, the code creates a new DateTime value for December 31 of the current year and assigns it to the from variable. Then, this code adds the number of years in the numYears variable. So, if the current year is 2020 and the value in numYears is 25, the from date will be 12/31/2045.

Once the from date is calculated, the code checks whether the date is valid. For a past date, the code checks if the parameter value is greater than or equal to the from date and less than the current date. Conversely, for a future date, the code checks to see if the property value is greater than the current date and less than or equal to the from date. If the validation succeeds, the code returns the Success field of the ValidationResult class. Otherwise, the code gets either the custom validation message or the default validation message, passes it to the constructor of a new ValidationResult object, and returns that object.

The second code example shows how to use the YearsFromNow attribute. This shows that you can pass the values of parameters defined by the constructor, as well as the values of properties defined by the class. When you do that, you should code the parameter values first, followed by the named property values.

## A custom attribute that accept values via a constructor and a property

```

public class YearsFromNowAttribute : ValidationAttribute
{
    private int numYears;
    public YearsFromNowAttribute(int years) { // constructor
        numYears = years;
    }
    public bool IsPast { get; set; } = false; // property with default

    protected override ValidationResult IsValid(object value,
        ValidationContext ctx)
    {
        if (value is DateTime) {
            // cast value to DateTime
            DateTime dateToCheck = (DateTime)value;

            // calculate date range
            DateTime now = DateTime.Today;
            DateTime from;

            if (IsPast) {
                from = new DateTime(now.Year, 1, 1);
                from = from.AddYears(-numYears);
            }
            else {
                from = new DateTime(now.Year, 12, 31);
                from = from.AddYears(numYears);
            }

            // check date
            if (IsPast) {
                if (dateToCheck >= from && dateToCheck < now) {
                    return ValidationResult.Success;
                }
            }
            else {
                if (dateToCheck > now && dateToCheck <= from) {
                    return ValidationResult.Success;
                }
            }
        }
        string msg = base.ErrorMessage ??
            ctx.DisplayName + " must be a " + (IsPast ? "past" : "future") +
            " date within " + numYears + " years of now.";
        return new ValidationResult(msg);
    }
}

```

## A DOB property that requires a past date no more than 100 years ago

```

[YearsFromNow(100, IsPast = true)] // constructor parameter first
public DateTime? DOB { get; set; }

```

### Description

- You can use the constructor and named properties to accept data for a data attribute.
- When you use an attribute, you must code the values for the constructor parameters first, followed by the values for the properties.

Figure 11-9 How to pass values to a custom data attribute

## How to check multiple properties

The two custom data attributes you've seen so far only check the value of the property they decorate. However, it's possible to create a data attribute that checks more than one property in the model object. For instance, the RequiredContactInfo data attribute at the top of figure 11-10 checks that a user has entered either a phone number or an email address.

To do that, the code in the `IsValid()` method uses the `ObjectInstance` property of the `ValidationContext` class to return the object that contains the properties being checked. However, it returns it as the object type, so you need to cast it to the correct type. In this example, the code casts the object to the `Customer` type. To use an attribute like this, you only need to decorate one of the properties it validates with the attribute.

Because this custom data attribute class only works with the `Customer` model class, it may be hard to reuse. As a result, it sometimes can make more sense to simply validate the class rather than create several data attributes that you may not be able to use anywhere else. This is called *class-level validation*, and the rest of this figure shows how it works.

To start, the two tables present the `IValidatableObject` interface and its `Validate()` method. To code a class that can validate itself, you need to implement this interface and method. Like the `IsValid()` method, the `Validate()` method accepts a `ValidationContext` object. However, the `Validate()` method returns a collection of `ValidationResult` objects rather than a single `ValidationResult` object. MVC calls the `Validate()` method during model binding.

The third table presents another constructor for the `ValidationResult` class. Like the constructor presented in figure 11-8, this constructor accepts a string value for the validation message. However, it also accepts a list of strings for the properties associated with the validation message.

Below the tables, the example shows some of the code in a `Customers` class that's been updated to validate itself. This class implements the `IValidatableObject` interface and its `Validate()` method. Within the `Validate()` method, the code returns a validation error if the `DOB` value is in the future. Similarly, it returns a validation error if there isn't a value for either the phone number or the email address.

When you use class-level validation, you can still use regular data attributes to perform property-level validation. In this example, for instance, the class decorates the `DOB` property with the `Required` attribute. However, class-level validation only runs if all the property-level validation has passed. This can create a 2-step validation process that users might not like.

One way to address this issue is to only use class-level validation. To do that, you avoid data attributes and manually code all validation in the `Validate()` method. Of course, this isn't as convenient as using data attributes. And, depending on how much validation there is, this can result in unwieldy code. So, whether you code a class that validates itself or individual data attributes that are specific to that class depends on your needs and preferences.

## A custom attribute that checks more than one property in a class

```
public class RequiredContactInfoAttribute : ValidationAttribute
{
    protected override ValidationResult IsValid(object v,
                                                ValidationContext c){
        var cust = (Customer)c.ObjectInstance;
        if (string.IsNullOrEmpty(cust.PhoneNumber) &&
            string.IsNullOrEmpty(cust.EmailAddress))
        {
            string msg = base.ErrorMessage ?? "Enter phone number or email.";
            return new ValidationResult(msg);
        }
        else {
            return ValidationResult.Success;
        }
    }
}
```

## The single method of the IValidatableObject interface

Method	Description
Validate(context)	Checks whether an object is valid. Accepts a ValidationContext object. Returns an IEnumerable of ValidationResult objects.

## A constructor of the ValidationResult class

Constructor	Description
ValidationResult(m, list)	Creates an object with the validation message and list of properties.

## A custom validation class that checks more than one field

```
public class Customer : IValidatableObject {
    ...
    [Required(ErrorMessage = "Please enter a date of birth.")]
    public DateTime? DOB { get; set; }

    public string PhoneNumber { get; set; }
    public string EmailAddress { get; set; }

    public IEnumerable<ValidationResult> Validate(ValidationContext ctx) {
        if (DOB > DateTime.Now)
        {
            yield return new ValidationResult(
                "Date of birth can't be in the future.",
                new[] { nameof(DOB) });
        }
        if (string.IsNullOrEmpty(PhoneNumber) &&
            string.IsNullOrEmpty(EmailAddress))
        {
            yield return new ValidationResult(
                "Please enter a phone number or email address.",
                new[] { nameof(PhoneNumber), nameof(EmailAddress) });
        }
    }
}
```

Figure 11-10 How to check multiple properties

## How to customize client-side validation

The previous figures have shown how to add custom server-side data validation to an app. Now, the next few figures show how to add custom client-side data validation to an app.

### How to add data attributes to the generated HTML

When you create a custom data attribute, it's a good practice to enable it to perform validation on the client too. That way, the attribute can participate in client-side validation as described in figure 11-7.

To add client-side validation, you can start by updating your data attribute class to emit the `data-val-*` attributes that the jQuery validation libraries use as shown in figure 11-11. Then, you can add jQuery code to do the actual validation as shown in the next figure.

To specify the HTML that MVC emits for the tag that's being validated, a data attribute must implement the `IClientModelValidator` interface and its `AddValidation()` method shown in the table at the top of this figure. This is illustrated by the `PastDate` attribute in the first code example. This version accepts an optional constructor argument that allows you to limit valid past dates to a given time span.

This code starts by including the namespace that contains the `IClientModelValidator` interface. Then, the class includes a constructor that accepts an int value and assigns it to the `yearsLimit` variable. If there's no value, it assigns a default of -1.

The `IsValid()` method contains the validation that runs on the server. For instance, if the user has JavaScript disabled, MVC runs this code. This method is similar to the method presented in figure 11-8. However, this version checks if the number of years in the past is limited.

The `AddValidation()` method accepts a `ClientModelValidationContext` object that describes the context in which the validation is taking place. Here, the code uses the `Attributes` property of this object to add `data-val-*` attributes to the generated HTML. To start, it checks whether the `data-val` attribute exists before adding it. That's necessary because an error occurs if another custom data attribute has already added a `data-val` attribute and your code attempts to add it again. Then, it adds the attributes that store the years limit and the validation message.

To get the validation message, this code calls the `GetMsg()` method and passes it the `DisplayName` attribute for the property if it has one or the `Name` attribute if it doesn't. To get these attributes, it uses the `ModelMetadata` property, which allows you to access data about the model property. Then, it uses a conditional operator to determine if a max number of years in the past is included in the message.

The second code example shows the `DOB` property decorated with the `PastDate` attribute. This example passes a value of 100 to the constructor. As a result, this validation fails if the date of birth is in the future or more than 100 years in the past.

## A method of the IClientModelValidator interface

Method	Description
AddValidation(context)	Adds data-val-* attributes to the generated HTML.

## The updated PastDate attribute with client-side validation

```
using Microsoft.AspNetCore.Mvc.ModelBinding.Validation;
...
public class PastDateAttribute : ValidationAttribute, IClientModelValidator
{
    private int numYears;
    public PastDateAttribute(int years = -1) => numYears = years;

    protected override ValidationResult IsValid(object val,
        ValidationContext ctx)
    {
        if (val is DateTime) {
            DateTime dateToCheck = (DateTime)val;
            if (numYears == -1) { // no limit on past date
                if (dateToCheck < DateTime.Today)
                    return ValidationResult.Success;
            } else {
                DateTime minDate = DateTime.Today.AddYears(-numYears);
                if (dateToCheck >= minDate && dateToCheck < DateTime.Today)
                    return ValidationResult.Success;
            }
        }
        return new ValidationResult(GetMsg(ctx.DisplayName));
    }

    public void AddValidation(ClientModelValidationContext c)
    {
        if (!c.Attributes.ContainsKey("data-val"))
            c.Attributes.Add("data-val", "true");
        c.Attributes.Add("data-val-pastdate-numyears", numYears.ToString());
        c.Attributes.Add("data-val-pastdate",
            GetMsg(c.ModelMetadata.DisplayName ?? c.ModelMetadata.Name));
    }

    private string GetMsg(string name) =>
        base.ErrorMessage ?? name + " must be a valid past date" +
        (numYears == -1 ? "." : " (max " + numYears + " years ago).");
}
}
```

## A model property decorated with the PastDate attribute

```
[PastDate(100)]
public DateTime? DOB { get; set; }
```

## The HTML that the PastDate attribute emits

```
<input type="text" class="form-control" data-val="true"
    data-val-pastdate="DOB must be a valid past date (max 100 years ago)."
    data-val-pastdate-numyears="100" id="DOB" name="DOB" value="" />
```

## Description

- To add the data-val-\* attributes jQuery uses for validation to an HTML element, implement the IClientModelValidator interface and its AddValidation() method.

Figure 11-11 How to add data attributes to the generated HTML

The last example in figure 11-11 shows the HTML that MVC generates for an `<input>` tag bound to this DOB property. This HTML includes a `data-val` attribute set to true, a `data-val-pastdate` attribute set to the default error message for 100 years, and a `data-val-pastdate-numyears` attribute set to 100.

## How to add a validation method to the jQuery validation library

---

In the previous figure, you learned how to update your custom data attribute to generate `data-val-*` attributes. Now, figure 11-12 shows how to tell the jQuery validation libraries to use these attributes. To do that, you need to add a JavaScript function that performs the validation to the jQuery validation library. In addition, you need to add the name of the validation function and its parameters to the adapters collection of the jQuery unobtrusive validation library.

This figure begins by presenting a JavaScript file named `pastdate.js`. This file starts by calling the `addMethod()` function of the jQuery validation library. The two arguments of this function specify the name of the function you're adding and the code for the function.

The anonymous function that's passed to the `addMethod()` function defines three parameters. The first parameter specifies the value of the HTML element that's being validated. By default, this parameter is a string, but you can cast it to another type if necessary. The second parameter specifies the HTML element. You can use this parameter to get information about the HTML element such as the values of its attributes. And the third parameter specifies any other parameters the function needs to do its work. In this figure, it's mapped to the `numyears` parameter.

The body of the anonymous function starts by checking whether the user entered a value. If so, it checks whether the value is a valid date. If either of these checks fail, the function returns false.

After those preliminary checks, the code converts the third parameter to a number and stores it in the `numYears` variable. This variable stores the number of years that's passed to the constructor of the `PastDate` attribute. Or, if the code didn't pass any years, it stores the default value of -1.

Next, the code gets the current date, and it checks the date entered by the user. If the `numYears` variable is -1, there's no limit to how far in the past the date can be. In that case, the code just checks whether the date is before the current date. However, if the `numYears` variable stores a positive value, the code calculates the minimum date and checks whether the date is after the minimum date and in the past.

The last statement in this file adds the name of the validation function to the adapters collection of the jQuery unobtrusive validation library. To do that, it uses the `addSingleVal()` function because the `pastdate()` function accepts a single parameter value named `numyears`. If a validation function doesn't accept any parameters, you can use the `addBool()` function to add it to the adapters collection. Or, if a validation function accepts multiple parameters, you can use the `add()` function. However, the details of using these functions are beyond the scope of this book.

## The pastdate.js file

```

jQuery.validator.addMethod("pastdate", function (value, element, param) {
    // get date entered by user, confirm it's a date
    if (value === '') return false;
    var dateToCheck = new Date(value);
    if (dateToCheck === "Invalid Date") return false;

    // get the number of years
    var numYears = Number(param);

    // get the current date
    var now = new Date();

    // check date
    if (numYears == -1) {
        if (dateToCheck < now) return true;
    } else {
        // calculate limit
        var minDate = new Date();
        var minYear = now.getFullYear() - numYears;
        minDate.setFullYear(minYear);

        if (dateToCheck >= minDate && dateToCheck < now) return true;
    }
    return false;
});

jQuery.validator.unobtrusive.adapters.addSingleVal("pastdate", "numyears");

```

## The header section of the layout

```

<head>
    <meta name="viewport" content="width=device-width" />
    <link href="~/lib/bootstrap/dist/css/bootstrap.min.css"
          rel="stylesheet" />

    <script src="~/lib/jquery/jquery.min.js"></script>
    <script src="~/lib/jquery-validation/jquery.validate.min.js"></script>
    <script src="~/lib/jquery-validation-unobtrusive/
              jquery.validate.unobtrusive.min.js"></script>
    <script src="~/js/pastdate.js"></script>
    <title>@ViewBag.Title</title>
</head>

```

## Description

- To implement client-side validation, you need to use the addMethod() function to add a JavaScript function that performs the validation to the jQuery validation library.
- To map a JavaScript validation function to its HTML5 data-val-\* attributes, you need to use a JavaScript function to add the validation function to the adapters collection of the jQuery unobtrusive validation library.
- To map a JavaScript validation function that accepts one argument, you can use the addSingleVal() function. For validation functions that accept zero arguments, you can use the addBool() function. And for validation functions that accept multiple arguments, you can use the add() function.

---

Figure 11-12 How to add a validation function to the jQuery validation library

To use the code in the `pastdate.js` file, you need to include a `<script>` element for that file in addition to the required jQuery libraries. When you do that, be sure to code your custom file after the jQuery libraries.

## How to work with remote validation

---

In addition to enabling custom attributes so the validation runs on the client, you can write code on the server that's called by the client without reloading the page. This is called *remote validation*, and you can use it when you need to perform a task on the server like accessing a file or a database.

To enable remote validation, you can use the `Remote` attribute. The table at the top of figure 11-13 presents two constructors of the `RemoteAttribute` class. Both accept arguments that tell MVC where to find the server-side code to run. Note that when you use this attribute, you must include a using directive for the `Microsoft.AspNetCore.Mvc` namespace.

Below the table, the first code example shows how to use the `Remote` attribute. To do that, you decorate a property in your model with this attribute and provide the appropriate constructor arguments. Here, the `Remote` attribute specifies "CheckEmail" as the name of the action method and "Validation" as the name of the controller.

When you code the action method for the `Remote` attribute, the method must have a return type of `JsonResult` and it must accept a parameter with the same name as the property that the `Remote` attribute decorates. Usually, it's best to code this parameter as a string. Then, you can convert the value to another type in the body of the method if necessary.

In the second example, the `CheckEmail()` action method checks whether the specified email already exists. To do that, it starts by calling the `CheckEmail()` method of a static data access class named `Utility`. Then, if the email does exist, the action method returns a validation message. Otherwise, it returns true. Either way, this code returns the data as a `JsonResult` object by calling the `Json()` method that's provided by the controller.

The second table presents the `AdditionalFields` property of the `RemoteAttribute` class. This property lets you identify additional values that the action method should retrieve from the POST request.

The code below this table shows how to use the `AdditionalFields` property. This code shows how you can pass additional parameters to an action method if necessary. In this figure, the code passes a `username` that's bound to the model as well as a `region` that's included in a hidden field.

There are two important things to understand about remote validation. First, it doesn't run if the field is left blank. So, you may want to code a `Required` attribute before your `remote` attribute. Second, even though the code for the `Remote` attribute is in a controller on the server, it's only called by the client. As a result, if a user has JavaScript disabled in their browser, the remote validation doesn't run. So, you should always code your app so the validation runs on the server too as shown later in this chapter.

## Two constructors of the RemoteAttribute class

Constructor	Description
<code>RemoteAttribute(act, ctl)</code>	Identifies the action method and controller to be called by the client.
<code>RemoteAttribute(act, ctl, area)</code>	Identifies the action method, controller, and area to be called by the client.

## A model property with a Remote attribute

```
[Remote("CheckEmail", "Validation")]
public string Email { get; set; }
```

## The CheckEmail() action method in the Validation controller

```
public JsonResult CheckEmail(string email)
{
    bool hasEmail = Utility.CheckEmail(email);      // checks database
    if (hasEmail)
        return Json($"Email address {email} is already registered.");
    else
        return Json(true);
}
```

## One property of the RemoteAttribute class

Property	Description
<code>AdditionalFields</code>	A comma-separated list of any additional values to be sent to the specified action method.

## Remote validation that gets data from additional fields

### The model

```
[Remote("CheckEmail", "Validation", AdditionalFields = "Username, Region")]
public string Email { get; set; }
public string Username { get; set; }
```

### The view

```
<input asp-for="EmailAddress" class="form-control" />
<input asp-for="Username" class="form-control" />
<input type="hidden" name="Region" value="West" />
```

### The CheckEmail() action method in the Validation controller

```
public JsonResult CheckEmail (string email, string username, string region){
    // validation code
}
```

## Description

- *Remote validation* allows you to write code on the server that's called by the client without reloading the whole page.
- The `AdditionalFields` property of the `Remote` attribute lets you identify other HTTP values the action method should retrieve from the POST request.
- The code in the controller is only called by the client. You should always make sure there's a server-side version, too.

Figure 11-13 How to work with remote validation

## The Registration app

---

The rest of this chapter presents the Registration page of the Registration app. This page uses many of the validation techniques presented so far in this chapter.

### The user interface and CSS

---

Figure 11-14 shows the user interface for the Registration page after the user has entered invalid data and clicked the Register button. This shows that the app uses the CSS presented at the bottom of this figure to style the text boxes with validation messages.

Most of the validation on this page is handled by the built-in data attributes that are provided by MVC. For instance, all of the fields use the Required attribute. In addition, the Name field uses the RegularExpression attribute to make sure the user doesn't enter special characters. Similarly, the Password field uses the Compare attribute to make sure the Password and Confirm Password fields match, and it uses the StringLength attribute to make sure the password is less than the maximum number of characters.

However, this page also uses two custom validation techniques. First, the Email Address field uses the Remote attribute to access the server to check that the email address entered by the user isn't already in use. As a convenience to the user, client-side code performs this check. For security reasons, server-side code also performs this check.

Second, the DOB field uses a custom data attribute to check that the user is at least 13 years old. Again, this check is performed by client-side code to keep the page responsive and by server-side code to keep the page secure.

## The Registration app with invalid data

A screenshot of a web browser window titled "Registration". The address bar shows "localhost:5001/Register". The main content is a "Registration" page with five input fields and their validation messages:

- Username: AmazingGrace! - "Username may not contain special characters."
- Email Address: gracehopper@gmail.com - "Email address gracehopper@gmail.com already in use."
- DOB: 12/09/2016 - "You must be at least 13 years old."
- Password: ..... - "'Password' and 'Confirm Password' do not match."
- Confirm Password: - "Please confirm your password."

A blue "Register" button is at the bottom.

## The CSS for the validation class

```
.input-validation-error {  
    border: 2px solid #dc3545; /* same red as text-danger */  
    background-color: #faebd7; /* antique white */  
}
```

### Description

- The Registration page uses data validation to make sure all the fields have a value, the username doesn't have special characters, the email address isn't already in use, the user is at least 13 years old, the password is less than 25 characters, and the confirm password matches the password.
- The Registration page validates data on the client as a convenience to users but performs the same validation on the server.
- The Registration page uses CSS to highlight the text boxes associated with a validation message.

---

Figure 11-14 The user interface and CSS for the Registration page

## The Customer and RegistrationContext classes

---

Figure 11-15 presents the Customer model class and the RegistrationContext database context class. The Customer class decorates its properties with several of the standard data attributes. Most of these attributes include custom validation messages, though the Compare attribute on the Password property uses the default validation message.

The ConfirmPassword property is decorated with the DisplayName attribute. This provides MVC with a friendly name to display in the validation messages. This property is also decorated with the NotMapped attribute. This attribute isn't used for data validation. Instead, it's used to prevent EF Core from creating a ConfirmPassword column in the Customers table that's created from this Customer class.

The EmailAddress property is decorated with a Remote attribute. The values passed to the constructor of this attribute tell it to use code in the CheckEmail() action method of the Validation controller to validate the email address. The code for this action method is shown later in this chapter.

The DOB property is decorated with a custom data attribute named MinimumAge. This attribute includes a constructor with an argument that specifies the minimum age as well as a property value that specifies a custom validation message. This custom data attribute class and the code that allows it to work on the client as well as the server is presented in the next figure.

## The Customer class

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using Microsoft.AspNetCore.Mvc;
...
public class Customer
{
    public int ID { get; set; } // automatically generated by database

    [Required(ErrorMessage = "Please enter a username.")]
    [RegularExpression("^[a-zA-Z0-9 ]+$",
        ErrorMessage = "Username may not contain special characters.")]
    public string Username { get; set; }

    [Required(ErrorMessage = "Please enter an email address.")]
    [Remote("CheckEmail", "Validation")]
    public string EmailAddress { get; set; }

    [Required(ErrorMessage = "Please enter a date of birth.")]
    [MinimumAge(13, ErrorMessage = "You must be at least 13 years old.")]
    public DateTime? DOB { get; set; }

    [Required(ErrorMessage = "Please enter a password.")]
    [Compare("ConfirmPassword")]
    [StringLength(25,
        ErrorMessage = "Please limit your password to 25 characters.")]
    public string Password { get; set; }

    [Required(ErrorMessage = "Please confirm your password.")]
    [Display(Name = "Confirm Password")]
    [NotMapped]
    public string ConfirmPassword { get; set; }
}
```

## The RegistrationContext class

```
public class RegistrationContext : DbContext
{
    public RegistrationContext(DbContextOptions<RegistrationContext> options)
        : base(options) { }

    public DbSet<Customer> Customers { get; set; }
}
```

## Description

- The Customer class provides data validation by decorating its properties with attributes from the System.ComponentModel.DataAnnotations namespace.
- The Customer class uses the Remote attribute to use server-side code to check if the email address entered by the user is already in the database.
- The Customer class uses a custom MinimumAge attribute to check if the user is at least 13 years old.
- The RegistrationContext class communicates with a database named Registration. To configure it, you must modify the Startup.cs and appsettings.json files as described in chapter 4.

---

Figure 11-15 The Customer and RegistrationContext classes

## The MinimumAgeAttribute class

---

Figure 11-16 presents the `MinimumAgeAttribute` class. The code for this class imports the namespace that contains the `ValidationAttribute` class that provides the basic functionality of a data attribute. Then, the `MinimumAgeAttribute` class inherits the `ValidationAttribute` class.

In addition, the `MinimumAgeAttribute` class imports the namespace that contains the `IClientModelValidator` interface. Then, this class implements the `AddValidation()` method of that interface so it can generate `data-val-*` attributes in the HTML. That way, the custom JavaScript function that's added to the jQuery validation libraries can use these attributes.

The `MinimumAgeAttribute` class has a constructor that accepts an `int` value and assigns it to a private variable named `minYears`. This is the value that determines the minimum age that's checked by the validation.

The `IsValid()` method overrides the virtual `IsValid()` method of the `ValidationAttribute` base class. First, it checks whether the value entered by the user is a date. If so, the code casts it to a `DateTime` value and calls the `AddYears()` method of the `DateTime` class to add the number of years in the `minYears` variable to the date to check.

After adding the years to the date, the code checks whether the date is less than or equal to today's date. If it is, the user is older than the minimum number of years. As a result, the code returns the `Success` field of the `ValidationResult` class. Otherwise, it returns a new `ValidationResult` object. To get the validation message for this object, it calls the `GetMsg()` method at the end of this class and passes it the `DisplayName` property of the `ValidationContext` parameter.

The `AddValidation()` method uses the `context` object parameter to add `data-val-*` attributes to the HMTL element. This code only adds the `data-val` attribute if it doesn't exist. In addition, it calls the `GetMsg()` method to get the validation message to embed in the HTML. Depending on whether the `DisplayName` property is null, this code passes the `GetMsg()` method the `DisplayName` property or the `Name` property of the `context` object parameter.

## The minimum-age JavaScript file

---

Figure 11-16 shows a JavaScript file named `minimum-age.js` that calls the `addMethod()` function of the jQuery validation library and passes it an anonymous function. This anonymous function starts by checking if the user entered a value and if that value is a valid date. If not, it returns `false`. Otherwise, it uses the parameter named `param` to get the value from the `data-val-minimumage-years` attribute and uses it to calculate the date to check. Then, the code returns a Boolean value that indicates whether the date to check is valid (less than or equal to the current date). Finally, the last line of code adds the name of the new validation function and its parameter to the `adapters` collection of the jQuery unobtrusive validation library.

## The MinimumAgeAttribute class

```
using System.ComponentModel.DataAnnotations;
using Microsoft.AspNetCore.Mvc.ModelBinding.Validation;
...
public class MinimumAgeAttribute : ValidationAttribute, IClientModelValidator
{
    private int minYears;
    public MinimumAgeAttribute(int years) {
        minYears = years;
    }

    // overrides IsValid() method of ValidationAttribute base class
    protected override ValidationResult IsValid(object value,
        ValidationContext ctx) {
        if (value is DateTime) {
            DateTime dateToCheck = (DateTime)value;
            dateToCheck = dateToCheck.AddYears(minYears);
            if (dateToCheck <= DateTime.Today) {
                return ValidationResult.Success;
            }
        }
        return new ValidationResult(GetMsg(ctx.DisplayName));
    }

    // implements AddValidation() method of IClientModelValidator interface
    public void AddValidation(ClientModelValidationContext ctx) {
        if (!ctx.Attributes.ContainsKey("data-val"))
            ctx.Attributes.Add("data-val", "true");
        ctx.Attributes.Add("data-val-minimumage-years",
            minYears.ToString());
        ctx.Attributes.Add("data-val-minimumage",
            GetMsg(ctx.ModelMetadata.DisplayName ?? ctx.ModelMetadata.Name));
    }
}

private string GetMsg(string name) =>
    base.ErrorMessage ?? $"{name} must be at least {minYears} years ago.";
}
```

## The minumim-age JavaScript file

```
jQuery.validator.addMethod("minimumage", function(value, element, param) {
    if (value === '') return false;

    var dateToCheck = new Date(value);
    if (dateToCheck === "Invalid Date") return false;

    var minYears = Number(param);

    dateToCheck.setFullYear(dateToCheck.getFullYear() + minYears);

    var today = new Date();
    return (dateToCheck <= today);
});

jQuery.validator.unobtrusive.adapters.addSingleVal("minimumage", "years");
```

---

Figure 11-16 The MinimumAgeAttribute class and JavaScript file

## The Validation and Register controllers

---

The Registration page uses remote validation to enable the client to check whether the email address is already in the database. However, if the user has JavaScript disabled, this check doesn't occur on the client. In that case, the Registration page also checks the email address on the server. To reduce duplication, both client-side and server-side code call the same method to perform the check.

The first code example in figure 11-17 shows the Validation controller. It starts with a constructor that accepts a database context object and assigns it to a private variable named context. Then, it defines an action method named CheckEmail() that accepts an email address. The client-side code calls this action method when it performs remote validation.

The CheckEmail() action method starts by passing the context object and email address to the static EmailExists() method of the static Check class that's shown in the third example. This method returns a validation message if the email address is already in the database or an empty string if it isn't.

The CheckEmail() action method continues by checking whether the message returned by the EmailExists() method is empty or null. If it is, the email address is OK, and the code stores a true value in TempData that the server-side check can use later. Otherwise, the code returns the validation message. Either way, the code calls the Json() method of the controller to convert the value that's sent to the client to JSON format.

The second example shows the Register controller. This controller also starts with a constructor that accepts a database context object and assigns it to a private variable named context. Then, it defines an overloaded action method named Index(). The overload that handles GET requests simply returns a view. The overload that handles POST requests, on the other hand, accepts a Customer object from the form and processes it.

To start, the Index() action method for POST requests uses TempData to determine whether it needs to perform a server-side check of the email address. If there's a value in TempData, the client-side check has succeeded, so the server-side check can be skipped. Also, reading the TempData value clears it, so you don't need to remove it manually.

If TempData returns a null value, it means that the client-side check didn't run, so the Index() action method needs to check the email. To do that, it passes the context object and the EmailAddress value to the static EmailExists() method shown in the third example. This time, if that method returns a validation message, the code adds it to the ModelState property of the controller with the EmailAddress property name as the dictionary key.

After the Index() action method adds any necessary messages to its ModelState property, it checks the IsValid property of the ModelState property and updates the database only if all validation has passed. This includes the validation check for the email address as well as the validation checks for all of the other data attributes that decorate the properties of the Customer class.

### The Validation controller

```
public class ValidationController : Controller
{
    private RegistrationContext context;
    public ValidationController(RegistrationContext ctx) => context = ctx;

    public JsonResult CheckEmail(string emailAddress)
    {
        string msg = Check.EmailExists(context, emailAddress);
        if (string.IsNullOrEmpty(msg)) {
            TempData["okEmail"] = true;
            return Json(true);
        }
        else return Json(msg);
    }
}
```

### The Register controller

```
public class RegisterController : Controller
{
    private RegistrationContext context;
    public RegisterController(RegistrationContext ctx) => context = ctx;

    public IActionResult Index() => View();

    [HttpPost]
    public IActionResult Index(Customer customer)
    {
        if (TempData["okEmail"] == null) {
            string msg = Check.EmailExists(context, customer.EmailAddress);
            if (!String.IsNullOrEmpty(msg)) {
                ModelState.AddModelError(nameof(Customer.EmailAddress), msg);
            }
        }
        if (ModelState.IsValid) {
            context.Customers.Add(customer);
            context.SaveChanges();
            return RedirectToAction("Welcome");
        }
        else return View(customer);
    }
}
```

### The static Check class

```
public static class Check
{
    public static string EmailExists(RegistrationContext ctx, string email) {
        string msg = "";
        if (!string.IsNullOrEmpty(email)) {
            var customer = ctx.Customers.FirstOrDefault(
                c => c.EmailAddress.ToLower() == email.ToLower());
            if (customer != null)
                msg = $"Email address {email} already in use.";
        }
        return msg;
    }
}
```

Figure 11-17 The Validation and Register controllers

## The layout

---

Figure 11-18 shows the Razor layout for the Registration app. It includes the main jQuery library because that library might be used by many pages of an app. For this app, the layout places the `<script>` tag for the jQuery library at the end of the `<body>` tag. This is a common practice that can make a page appear to load faster. However, it can also cause elements of your page to jump around on the user after all your JavaScript loads. To prevent this, you can include the jQuery library in the `<head>` tag.

After the `<script>` tag that includes the jQuery library, the layout calls the `RenderSection()` method that's available to layouts and views. The first argument to that method specifies the name of the section to render ("scripts"), and the second specifies a Boolean value indicating whether a view is required to have a section named "scripts". This Razor method allows individual views to optionally include additional JavaScript files. Since it comes *after* the tag that includes the jQuery library, any JavaScript files added by a view will have access to this library.

In summary, this layout loads the main jQuery library for every page in the Registration app. In addition, it provides a way for the other pages to load additional JavaScript files. This makes it possible for pages that need to validate user input to load the jQuery validation libraries as well as any custom JavaScript files for client-side validation as shown in the next figure. Since this approach only loads the JavaScript validation files when they're needed, it can decrease the amount of time that it takes to load a page.

## The layout

```
<!DOCTYPE html>
<html>
<head>
    <meta name="viewport" content="width=device-width" />
    <title>@ViewBag.Title</title>
    <link rel="stylesheet" type="text/css"
        href("~/lib/bootstrap/dist/css/bootstrap.min.css">
    <link href("~/css/site.css" rel="stylesheet" />
</head>
<body>
    <div class="container">
        <header class="bg-primary text-white text-center">
            <h1 class="m-3 p-3">Registration</h1>
        </header>
        <main>
            @RenderBody()
        </main>
    </div>
    <script src "~/lib/jquery/dist/jquery.min.js"></script>
    @RenderSection("scripts", false)
</body>
</html>
```

## Description

- The layout includes the jQuery library at the end of the `<body>` tag, though you can include it in the `<head>` tag instead if you prefer.
- The layout calls the `RenderSection()` method. This allows individual views to add additional JavaScript files. It places this statement *after* the `<script>` tag that includes the jQuery library.

---

Figure 11-18 The layout

## The Register/Index view

---

Figure 11-19 presents the Index view for the Registration page. It's a strongly-typed view with a Customer object as its model.

The Index view has a Razor section named scripts. This is the section that the layout in the previous figure renders after it includes the main jQuery library. In this view, the scripts section contains three `<script>` tags. The first two include the jQuery validation libraries that provide for client-side validation. The third includes the JavaScript file named minimum-age.js that's stored in the js folder. This is the JavaScript file that adds client-side validation to the custom MinimumAge attribute.

When you include JavaScript files, you need to make sure to include them in the correct order. For this app, the layout includes the main jQuery library that you need first. Then, within a view, you can include the jQuery validation library, followed by the jQuery unobtrusive validation library, followed by any custom validation files.

The Index view contains a `<form>` tag that posts to the Index() action method of the current controller, which is the Register controller. Within this tag, the view has several `<div>` tags that are formatted as a Bootstrap form-group. Each of these `<div>` tags contains an `<input>` tag that's bound to a property of the Customer model object. And after each `<input>` tag is a `<span>` tag that uses the `asp-validation-for` tag helper to display validation messages for that property.

Due to space constraints, this figure doesn't show the `<div>` tags for the password and confirm password fields. However, they look similar to the other `<div>` tags, except that their `<input>` tags specify a type of "password". If you'd like to view these tags, you can open the file for the Register/Index view that's in the downloadable app for this chapter.

## The Register/Index view

```
@model Customer

@{
    ViewData["Title"] = "Registration";
}

@section scripts {
    <script src="~/lib/jquery-validation/dist/jquery.validate.min.js"></script>
    <script src="~/lib/jquery-validation-unobtrusive/
                jquery.validate.unobtrusive.min.js"></script>
    <script src="~/js/minimum-age.js"></script>
}

<form asp-action="Index" method="post">
    <div class="form-group row">
        <div class="col-sm-2"><label>Username:</label></div>
        <div class="col-sm-4">
            <input asp-for="Username" class="form-control" /></div>
        <div class="col">
            <span asp-validation-for="Username" class="text-danger">
            </span></div>
        </div>
        <div class="form-group row">
            <div class="col-sm-2"><label>Email Address:</label></div>
            <div class="col-sm-4">
                <input asp-for="EmailAddress" class="form-control" /></div>
            <div class="col">
                <span asp-validation-for="EmailAddress" class="text-danger">
                </span></div>
            </div>
            <div class="form-group row">
                <div class="col-sm-2"><label>DOB:</label></div>
                <div class="col-sm-4">
                    <input type="text" asp-for="DOB"
                           class="form-control" /></div>
                <div class="col">
                    <span asp-validation-for="DOB" class="text-danger">
                    </span></div>
                </div>
                <!-- Password and ConfirmPassword fields -->
                <div class="row">
                    <div class="offset-2 col-sm-4">
                        <button type="submit" class="btn btn-primary">Register</button>
                    </div>
                </div>
            </div>
    </form>
```

### Description

- The Register/Index view has a Razor section named scripts that includes the jQuery validation libraries and the minimum-age.js file. That way, these JavaScript files are only loaded for the Registration page, not for other pages of the app that don't need them.

---

Figure 11-19 The Register/Index view

## Perspective

---

The goal of this chapter is to teach you how to validate the data that a user inputs into an ASP.NET MVC app. Now, if this chapter has worked, you should be able to use client-side and server-side code to validate data. This includes using the data attributes that are provided by MVC as well as creating your own custom data attributes. As usual, there's always more to learn. Still, this chapter should give you a good foundation for working with data validation.

## Terms

---

data attribute  
property-level validation  
class-level validation  
remote validation

## Summary

---

- You specify the validation to be applied to a model class by decorating its properties with *data attributes*.
- Code that validates a property of a class is called *property-level validation*. Code that validates the entire class instead of individual properties is called *class-level validation*. Class-level validation only runs if all the property-level validation has passed.
- *Remote validation* allows you to write code on the server that's called by the client without reloading the whole page.

## Exercise 11-1 Add data validation to an app

In this exercise, you'll review the default data validation provided by model binding and add some custom data validation to an app.

### View the domain model and test the default data validation

1. Open the Ch11Ex1TempManager web app in the ex\_starts directory.
2. Open the Package Manager Console and enter the Update-Database command to create the database for this web app.
3. In the Models folder, open the Temp class and view its code. Note that none of the properties have any data validation attributes.
4. Run the app and click the Add Temp button. Enter a date, a low temperature, and a high temperature. Then, click the Add button.
5. Click the Add Temp button again. This time, click the Add button without entering any data. Note that the page doesn't display validation messages. Instead, it displays blank data. Use the Delete button to delete the blank data.
6. Click the Add Temp button again. This time, enter "tomorrow" for the date and "four" for the low temperature. Click the Add button and review the validation messages that MVC displays because it can't convert the data you entered to the correct types.

### Add data validation to the Temp domain model

7. In the Temp class, add a using statement for the data annotations namespace.
8. Add the following data annotations to the Temp class:
  - The Date, Low, and High fields are required.
  - The Low and High fields must be within a range of -200 to 200.
9. Run the app and enter some incorrect Temp data. The app should display appropriate validation messages, so the user can correct the data.

### Display model-level and property-level validation messages

10. In the Controllers folder, open the HomeController class and find the Add() action method for POST requests.
11. In the else block that runs when the model state is not valid, add a model-level error with a message of "Please correct all errors" to the ModelState property.
12. In the Views folder, open the Home/Add view and modify the validation summary <div> tag so it only displays model-level validation messages.
13. Add a <span> tag after each <input> tag to display a property-level validation message.
14. Run and test the app. When you enter invalid data, it should display one model-level validation message plus one or more property-level messages.

**Make your data validation run on the client**

15. In the Add view, add a Razor @section block named scripts. Within this block, add <script> tags for the jQuery validation library and the jQuery unobtrusive validation library.
16. Run and test the app. When you enter invalid data, the model-level validation message should not display. That's because the validation is failing on the client, so the data isn't being sent to the server for validation there.

**Add remote validation that checks for a duplicate date in the database**

17. In the Controllers folder, add a new class named ValidationController.
18. Add a using statement for the TempManager.Models namespace.
19. Code a constructor that accepts a TempManagerContext object and stores it in a private property.
20. Code an action method named CheckDate(). This method should return a JsonResult object and define a string parameter named date.
21. In the method, convert the string parameter named date to a DateTime object and query the database for a Temp object with that date.
22. Use the Json() method to return true if that query returns null. Otherwise, return an error message.
23. In the Models folder, open the Temp class.
24. Add a using statement for the Mvc namespace and update the Date property to use the Remote attribute with the CheckDate() action method of the Validation controller.
25. Run the app and enter new Temp data with a duplicate date. This should display a validation message indicating that the date is already in the database.

**Add a server-side check for the remote validation**

26. Turn off JavaScript in your browser. To do that with Chrome, click the Chrome menu in the top right corner, select Settings, expand the Advanced node, select “Privacy and security”, click Site Settings, click Javascript, and change Allowed to Blocked.
27. Run the app and test it. It should allow you to enter Temp data with a duplicate date. When you're done, delete the Temp data that has the duplicate date.
28. In the HomeController class, add code to the Add() action method for POST requests that checks if the date is duplicate. If so, this code should add a property-level validation message to the ModelState property.
29. Run the app and enter new Temp data with a duplicate date. This should display a property-level validation message.
30. Make sure to turn Javascript back on in your browser!

# 12

## How to use EF Core

In this chapter, you'll learn how to use Entity Framework (EF) Core to work with data in a database. This includes how to create a new database from code (Code First development) and how to create code from an existing database (Database First development).

<b>How to create a database from code.....</b>	<b>440</b>
How to code entity and DB context classes.....	440
How to configure the database .....	442
How to manage configuration files.....	444
EF commands for working with a database .....	446
How to use EF migration commands .....	448
<b>How to work with relationships.....</b>	<b>450</b>
How entities are related .....	450
How to configure a one-to-many relationship.....	452
How to configure a one-to-one relationship.....	454
How to configure a many-to-many relationship.....	456
How to control delete behavior.....	458
<b>The Bookstore database classes .....</b>	<b>460</b>
The entity classes.....	460
The context and configuration classes.....	460
<b>How to create code from a database.....</b>	<b>466</b>
How to generate DB context and entity classes.....	466
How to configure a generated DB context class .....	468
How to modify a generated entity class .....	470
<b>How to work with data in a database.....</b>	<b>472</b>
How to query data.....	472
How to work with projections and related entities .....	474
How to insert, update, and delete data .....	476
<b>How to handle concurrency conflicts .....</b>	<b>478</b>
How to check for concurrency conflicts .....	478
How handle a concurrency exception.....	480
<b>How to encapsulate your EF code.....</b>	<b>482</b>
How to code a data access class .....	482
How to use a generic query options class.....	484
How to use the repository pattern.....	486
How to use the unit of work pattern .....	488
<b>Perspective .....</b>	<b>490</b>

## How to create a database from code

*Entity Framework (EF) Core* is an *object-relational mapping (ORM)* framework that allows you to map your entity classes to the tables of a database. The most common way to work with EF Core is to code your entity classes and a database context class first. Then, you can use EF to create a new database from these classes. This is called *Code First* development, and it's the approach that you'll learn about now.

### How to code entity and DB context classes

*Entity classes* represent the data structure for an app. For example, a bookstore has entities like books, authors, and genres. As a result, a Bookstore app needs entity classes to store data for those entities.

To illustrate, figure 12-1 begins with two examples that show the Book and Author entity classes. You can create objects from these classes to store the data that's needed by your app.

After you create your entity classes, you need to code a *database (DB) context class* that can communicate with a database. This class inherits the DbContext base class. To illustrate, the third example shows a context class named BookstoreContext that inherits the DbContext class. In addition, it has a constructor that accepts a DbContextOptions object and passes it to the constructor of the DbContext class. These options are usually configured by the Startup.cs class as described in figure 4-6 of chapter 4.

To enable your context class to work with collections of your entity classes, you need to add DbSet<Entity> properties. For instance, the BookstoreContext class has a Books property of the DbSet<Book> type and an Authors property of the DbSet<Author> type. When you create a database, these properties generate its tables. Then, you can use LINQ to query these properties.

The DbContext class has two virtual methods named OnConfiguring() and OnModelCreating(). You can override these methods to configure your context class and its DbSet properties. In this figure, the BookstoreContext class overrides both of these methods.

The OnConfiguring() method configures the context itself. For example, you can specify the connection string here. However, that's usually done in the appsettings.json file as shown by figure 4-6. You can also configure other features such as change tracking and logging, though that's beyond the scope of this book.

The OnModelCreating() method configures your entity classes. This, in turn, configures the database tables that are created from these entities. You'll learn more about how this works later in this chapter.

## A Book entity class

```
public class Book
{
    public int BookId { get; set; }
    public string Title { get; set; }
    public double Price { get; set; }
}
```

## An Author entity class

```
public class Author
{
    public int AuthorId { get; set; }
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

## A BookstoreContext class

```
using Microsoft.EntityFrameworkCore;
...
public class BookstoreContext : DbContext
{
    public BookstoreContext(DbContextOptions<BookstoreContext> options)
        : base(options) { }

    public DbSet<Book> Books { get; set; }
    public DbSet<Author> Authors { get; set; }

    protected override void OnConfiguring(
        DbContextOptionsBuilder optionsBuilder)
    {
        // code that configures the DbContext goes here
        base.OnConfiguring(optionsBuilder);
    }

    protected override void OnModelCreating(ModelBuilder modelBuilder)
    {
        // code that configures the DbSet entities goes here
        base.OnModelCreating(modelBuilder);
    }
}
```

## Description

- *Entity classes* define the data structure for the app and map to tables in a relational database. A *database (DB) context class* inherits the DbContext class and includes entity classes as DbSet properties.
- In the DB context class, the OnConfiguring() method allows you to configure the context, such as providing the connection string. Usually, though, you do this in the Startup.cs file.
- In the DB context class, the OnModelCreating() method allows you to configure the entity classes, such as providing initial seed data.
- To work with EF, you need to configure the database context. To do that, you can modify your appsettings.json and Startup.cs files as described in figure 4-6 of chapter 4.

---

Figure 12-1 How to code entity classes and a database context class

## How to configure the database

---

You can configure your entity classes so they create the database tables you want in three ways. First, you can configure them by convention. Second, you can configure them with data attributes. Third, you can configure them by using code.

When you configure by convention, you code your classes following established conventions, and EF takes it from there. Figure 12-2 starts by presenting some of these conventions. For example, if you code a property named Id (or ID) or join the name of the entity and Id (or ID), EF treats that property as the primary key of the database table. Additionally, if this property is of the int type, EF makes the primary key an identity column. Then, the database generates the key's value when a new row is inserted.

Other conventions are that string properties create columns of the nvarchar(max) type, and those columns allow null values. Primitive types like int and double, by contrast, create columns that don't allow null values.

The first code example in this figure shows how to create an identity column that serves as the primary key by convention. Here, the Book class has an int property named BookId. Since this is so easy, you should configure your classes by convention whenever possible.

Sometimes, though, the EF conventions may not yield the results you want. For instance, you may not want a string property to allow nulls. You may want a string property to be smaller than nvarchar(max). Or, you may not want the database to generate values for a primary key.

In these cases, you can use attributes like those presented in the first table in this figure. These attributes are stored in the DataAnnotations and DataAnnotations.Schema namespaces shown in the second example. Some of the data attributes in the DataAnnotations namespace can also be used for configuration. For example, you can use the Required and StringLength attributes to change the default configuration of a string property.

The second code example in this figure shows the Book entity class with some attributes for configuration. For instance, the Key attribute configures the ISBN property so it's the primary key even though it wouldn't be configured as the primary key by convention.

Sometimes, you might not want to configure by convention or by attributes. This could be because you have complex requirements, or it could be because you want to keep configuration information out of your entity classes. In these cases, you can use the methods of the *Fluent API* to write code that configures your classes.

The second table in this figure shows several of the Fluent API methods for configuration. Then, the code example below the table shows how you can use some of them. Here, the OnModelCreating() method shown in the previous figure contains some configuration code. This code uses the ModelBuilder parameter to configure the Title property of the Book entity. To do that, it chains together four methods of the Fluent API. This produces the same result as coding the Required and StringLength(200) attributes above the Title property.

## Some of the conventions for configuration in EF Core

- A property named Id or *ClassNameId* is the primary key. If the property is of the int type, the key is an identity column and its value is generated automatically by the database.
- A string property has a database type of nvarchar(max) and is nullable.
- An int or double property is not nullable.

## How to set an identity primary key by convention

```
public class Book {
    public int BookId { get; set; }
}
```

## Some of the data attributes for configuration

Attribute	Description
<b>Key</b>	The database column that's the primary key.
<b>NotMapped</b>	Indicates that a property or table shouldn't be mapped to the database.
<b>DatabaseGenerated</b>	Specifies how the database generates a value. Uses the Computed, Identity, and None values of the DatabaseGeneratedOption enum.

## How to use attributes to adjust the default configuration

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema
...
public class Book {
    [Key]
    public string ISBN { get; set; } // make primary key

    [Required]
    [StringLength(200)]
    public string Title { get; set; } // make string column not nullable
}
```

## Some of the Fluent API methods for configuration

Method	Description
<b>Entity&lt;T&gt;()</b>	Registers an entity for configuration.
<b>Property(lambda)</b>	Registers a property for configuration.
<b>HasKey(lambda)</b>	Configures the primary key or keys for the entity.
<b>HasData(entityList)</b>	Allows you to seed data for the entity when the database is created.
<b>ToTable(string)</b>	Identifies the table that an entity maps to.
<b>IsRequired()</b>	Configures a database column to be not nullable.
<b>HasMaxLength(size)</b>	Configures the size of a database column.

## How to chain Fluent API method calls to configure a property

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Book>()
        .Property(b => b.Title).IsRequired().HasMaxLength(200);
}
```

Figure 12-2 How to configure the database

## How to manage configuration files

---

The first code example in figure 12-3 shows the `OnModelCreating()` method of a context class with several statements that configure a `Book` entity. The first statement sets the `ISBN` property as the primary key. This is necessary because that property doesn't follow the naming convention for a primary key. The second statement makes the `Title` property not allow nulls and to have a maximum length of 200 characters. And the third statement uses the `HasData()` method to provide seed data when the table is created.

The statements in this example are for a single entity that has two properties. However, imagine how this code will grow as you add more properties to this entity and as you add other entities to the app. This is especially true if you decide to configure all of your entities with the Fluent API rather than with attributes. And the longer the code gets, the harder it becomes to read and maintain.

To keep this code manageable, you can store your configuration code in separate files, where each file stores a configuration class for each entity. Then, each class can implement the interface named `IEntityTypeConfiguration<T>` and override its `Configure()` method.

The second example shows a configuration class named `BookConfig` that configures the `Book` entity. This class implements the `IEntityTypeConfiguration<Book>` interface and overrides its `Configure()` method. This method accepts an `EntityTypeBuilder<Book>` object that represents the entity being configured. In this case, that entity is the `Book` entity. Then, this method uses that object to perform the same configuration shown in the first example. For this to work, the class must include a using directive for the `Microsoft.EntityFrameworkCore.Metadata.Builders` namespace, since it contains the `EntityTypeBuilder< TEntity >` class.

Once you've coded a separate configuration file, you apply it in the `OnModelCreating()` method of the context class. To do that, you create a new instance of the configuration class and pass it to the `ApplyConfiguration()` method of the `ModelBuilder` object as shown by the last example. This reduces the six lines of code in the first example to one line of code. As you can imagine, this goes a long way toward keeping the code in the `OnModelCreating()` method manageable, especially when an app contains many complex entities.

## Code that configures the Book entity in the OnModelCreating() method

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Book>().HasKey(b => b.ISBN);

    modelBuilder.Entity<Book>().Property(b => b.Title)
        .IsRequired().HasMaxLength(200);

    modelBuilder.Entity<Book>().HasData(
        new Book { ISBN = "1548547298", Title = "The Hobbit" },
        new Book { ISBN = "0312283709", Title = "Running With Scissors" }
    );
}
```

## Code that uses a separate configuration class for the Book entity

### A configuration class for the Book entity

```
internal class BookConfig : IEntityTypeConfiguration<Book>
{
    public void Configure(EntityTypeBuilder<Book> entity)
    {
        entity.HasKey(b => b.ISBN);

        entity.Property(b => b.Title)
            .IsRequired().HasMaxLength(200);

        entity.HasData(
            new Book { ISBN = "1548547298", Title = "The Hobbit" },
            new Book { ISBN = "0312283709", Title = "Running With Scissors" }
        );
    }
}
```

### Code that applies the configuration class in the OnModelCreating() method

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.ApplyConfiguration(new BookConfig());
}
```

## Description

- If you have a lot of Fluent API configuration code in the OnModelCreating() method of your context class, it can become difficult to understand and maintain. In that case, it's considered a best practice to divide this code into one configuration file per entity.
- To create a configuration file, you code a class that implements the IEntityTypeConfiguration<T> interface and its Configure() method.
- The Configure() method accepts an EntityTypeBuilder<T> object that represents the entity being configured.
- To apply the configuration code, you pass a new instance of the configuration class to the ApplyConfiguration() method of the ModelBuilder object that's passed to the OnModelCreating() method of the context class.

---

Figure 12-3 How to manage configuration files

## EF commands for working with a database

In chapter 4, you learned how to use the NuGet Package Manager Console (PMC) to execute PowerShell commands such as Add-Migration and Update-Database to create a database from DB context and entity classes. Figure 12-4 starts by reviewing how you open the PMC window. Then, it summarizes the Add-Migration and Update-Database commands, as well as some additional commands for working with a database. For any of these commands to work, your app needs to be configured to access a database as shown in figure 4-6.

The first command, Add-Migration, generates a migration file based on your context and entity classes. By default, this command creates the file with the name you specify in the Migrations folder. If that folder doesn't exist, this command creates it. If you want, you can use the -OutputDir parameter to specify another folder, as you'll see shortly.

The second command, Remove-Migration, removes the last migration file from the Migrations folder. This only works, though, if the migration file hasn't yet been applied to the database. If it has, you need to use the Update-Database command to revert the migration before you can remove the migration file. You'll see an example of this in the next figure.

The third command, Update-Database, applies any migration files that haven't yet been applied to the database. You can also use the Update-Database command to revert migrations as shown in the next figure.

The fourth command, Drop-Database, deletes the database. If this command doesn't work correctly, you can use SQL Server Object Explorer or another comparable tool to delete the database.

The fifth command, Script-Migration, generates a SQL script based on one or more migration files. Due to space considerations, this book doesn't show any examples of using this command. However, if you need to use it, you can find many examples online.

The sixth command, Scaffold-DbContext, generates DB context and entity classes from an existing database. Since you use this command with Database First development, it's described later in this chapter.

The next three tables in this figure show some of the parameters you can use with the Add-Migration, Update-Database, and Script-Migration commands to change their default behavior. The -Name parameter is required for the Add-Migration command. However, the rest of these parameters are optional. As a result, if you execute these commands without parameters, they use their default values.

The first time you run the Add-Migration command, EF creates a file named *DbContextNameModelSnapshot.cs* and stores it in the same folder as the migration files. For instance, a context class named BookstoreContext generates a file named BookstoreContextModelSnapshot.cs. This file contains a "snapshot" in code of the current database schema, or structure. When you add or remove migrations with the Add-Migration and Remove-Migration commands, EF updates this file.

## How to open the NuGet Package Manager Console (PMC) window

- Select Tools → NuGet Package Manager → Package Manager Console.

## Some of the PowerShell EF commands

Command	Description
<b>Add-Migration</b>	Adds a named migration file to the Migrations folder. If this folder doesn't exist yet, it creates the folder.
<b>Remove-Migration</b>	Removes the last migration file from the Migrations folder. Only works with migrations that have not yet been applied with the Update-Database command.
<b>Update-Database</b>	Updates the database to the last migration or reverts the database to the migration specified by the optional -Name parameter.
<b>Drop-Database</b>	Deletes the database.
<b>Script-Migration</b>	Generates a SQL script based on the migration file or files.
<b>Scaffold-DbContext</b>	Generates DB context and entity classes from an existing database. See figure 12-14.

### Parameters for the Add-Migration command

Parameter	Description
<b>-Name</b>	The name of the migration file. This is a required parameter.
<b>-OutputDir</b>	The folder where the file will be created. The default is “Migrations”.

### Parameter for the Update-Database command

Parameter	Description
<b>-Name</b>	The name of the migration file. The default is the last migration.

### Parameters for the Script-Migration command

Parameter	Description
<b>-From</b>	The starting migration file. The default is the first one.
<b>-To</b>	The ending migration file. The default is the last one.
<b>-Output</b>	The file to write to. Defaults to a generated name in the solution folder.
<b>-Idempotent</b>	Generates checks to make sure SQL commands aren't repeated.

### Description

- The PMC executes PowerShell commands to create, apply, and revert migration files.
- A migration file contains C# code for creating or updating database objects. Specifically, each migration file has an Up() method with code that runs when a migration is applied, and a Down() method with code that runs when a migration is reverted.
- The first time you run the Add-Migration command, EF creates a file named *DbContextNameModelSnapshot.cs*. This file contains a “snapshot” of the current database schema. When you add or remove subsequent migrations, that file is updated.

Figure 12-4 EF commands for working with a database

## How to use EF migration commands

Figure 12-5 presents some examples that use EF commands in the Package Manager Console (PMC) to work with the Bookstore database. The first scenario in this figure shows how to create and then update the database. Step 1 creates a migration file named Initial.cs in the Migrations folder. Then, step 2 updates the database. This applies the Initial migration and creates the database.

Although it's not required, you can include the -Name flag with the Add-Migration command like this:

```
PM> Add-Migration -Name Initial
```

If you want to change the name of the migrations folder, you can include the optional -OutputDir parameter like this:

```
PM> Add-Migration Initial -OutputDir MyMigrations
```

Finally, if you want to specify a path for the migrations folder, you can do that like this:

```
PM> Add-Migration Initial -OutputDir Models/DataLayer/Migrations
```

Here, the path is relative to the project folder.

Step 3 adds a new property named Discount to the Book entity. Then, step 4 generates a new migration file that includes this new column.

Steps 5 through 7 describe what to do if you realize that you made a mistake in adding your new property in steps 3 and 4. For example, suppose you forgot to indicate that the Discount property is nullable. To fix that, you can change the data type for this property in the Book class so it is nullable. Then, you can run the Add-Migration command again to create another migration file to correct the one that adds the Discount property. At this point, you have two migration files ready to be applied. So, when you update the database in step 8, both migration files are applied.

The next two scenarios show how to revert migrations that have been applied to the database. To do this, you use the Update-Database command and include the name of the migration you want to revert to. The first of these scenarios, for example, reverts to the AddDiscount migration, which means that the Discount property is no longer nullable. Then, you can use the Remove-Migration command to remove the migration file that made this property nullable.

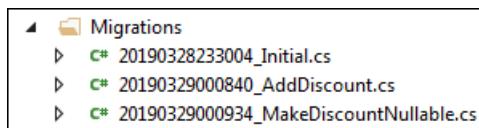
The second of these scenarios shows how to revert all migrations. To do that, you enter 0 for the migration name. Then, you can use the Remove-Migration command to delete the unused migration files. Or, you can delete them manually. When manually deleting all migration files, you also need to delete the snapshot file.

You can also use the Remove-Migration command to remove a migration that hasn't been applied yet. You might want to do that if you notice a problem with a migration file before you apply it. Then, you can adjust your code and run the Add-Migration command again to correct the problem.

## How to create and then update a database

1. Create a migration file named Initial based on the context and entity classes presented earlier in this chapter by entering this command:  
**PM> Add-Migration Initial**
2. Create a database based on the migration file by entering this command:  
**PM> Update-Database**
3. Add a property named Discount of the double type to the Book class.
4. Create a migration file named AddDiscount by entering this command:  
**PM> Add-Migration AddDiscount**
5. Review the migration file and note that the Discount property doesn't accept nulls.
6. Change the Discount property in the Book class to the data type of double?.
7. Generate another migration file by entering this command:  
**PM> Add-Migration MakeDiscountNullable**
8. Apply the migration files to the database by entering this command:  
**PM> Update-Database**

## The Migrations folder after completing these steps



## How to revert one or more migrations

1. To revert changes to the database by running the Down() method in every migration file that comes after the AddDiscount file, enter this command:  
**PM> Update-Database AddDiscount**
2. To remove the unapplied MakeDiscountNullable migration file that was reverted in step 1 from the Migrations folder, enter this command:  
**PM> Remove-Migration**

## How to revert all the migrations

1. To revert all changes that have been applied to the database by running the Down() method in all the migration files, enter this command:  
**PM> Update-Database 0**
2. To remove all migration files from the Migrations folder, enter the Remove-Migration command repeatedly. Or, manually delete all the migration files from the Migrations folder, including the snapshot file.

## Note

- When you run one of these commands, you might see a warning to update your tools. To do that, you can run an Install-Package command like this:  
**PM> Install-Package Microsoft.EntityFrameworkCore.Tools -Version 3.0.1**  
However, you should use the version number that's included in the warning message.

Figure 12-5 How to use EF migration commands

## How to work with relationships

So far, you have learned how to code entity classes and use those classes to create database tables. However, entities are typically related to other entities. Similarly, the corresponding database tables are typically related to other tables. For example, a genre might have many books. An author might have a detailed bio. A book might have several authors. And an author might have several books. That's why the next few figures show how to configure relationships between entities and their corresponding tables.

### How entities are related

Entities can be related to other entities by values in specific properties. The two entities at the top of figure 12-6 illustrate this concept. Here, the *Genre* entity is related to the *Book* entity because they share a *GenreId* property.

Typically, relationships exist between the primary key in one entity and the foreign key in another entity. A *primary key (PK)* uniquely identifies an instance of an entity. A *foreign key (FK)* refers to a primary key in another entity. In this figure, the diagram shows that the *GenreId* property in the *Genre* entity is a primary key, and the *GenreId* property in the *Book* entity is a foreign key. In other words, the foreign key in the *Book* entity refers to the primary key of the *Genre* entity.

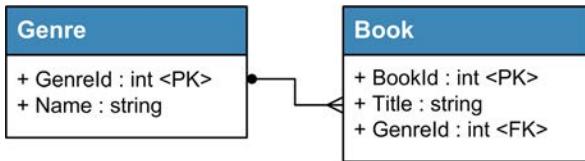
The most common relationship between entities is a *one-to-many relationship*. This is the relationship shown in the diagram. It's a one-to-many relationship because a *Book* entity can only have one *GenreId* value that relates it to one *Genre* entity (the *one* side). However, a *Genre* entity can be related to several *Book* entities (the *many* side). Entities can also have a *one-to-one* or *many-to-many relationship* as described in the first table in this figure.

You can configure relationships by convention, with data attributes, and by using the Fluent API. In the next three figures, you'll learn how to configure the three types of relationships described here. First, though, this figure presents some of the attributes and methods used to configure relationships in EF Core.

The second table presents two attributes used to configure relationships. The *ForeignKey* attribute specifies the property that's the foreign key, and the *InverseProperty* attribute specifies the navigation property on the other end of a relationship.

The third table presents some methods of the Fluent API for configuring relationships. This API uses the Has/With pattern to configure relationships. Here, methods that begin with *Has* represent the starting point of a relationship, and the methods that begin with *With* represent the ending point.

## Two entities that have a one-to-many relationship



## Three types of relationships between entities

Relationship	Example
One to many	Each book can be related to only one genre, but each genre can be related to one or more books.
One to one	Each author is related to only one author bio, and each author bio is related to only one author.
Many to many	Each book is related to one or more authors, and each author is related to one or more books.

## Two attributes that can be used to configure relationships

Attribute	Description
<b>ForeignKey</b>	Specifies the property that's the foreign key in a relationship.
<b>InverseProperty</b>	Specifies the navigation property on the other end of a relationship.

## The Has/With configuration pattern in the Fluent API

- *Has* represents the side of the relationship where the configuration starts.
- *With* represents the side of the relationship where the configuration ends.

## Fluent API methods used to configure relationships in EF Core

Methods	Description
<b>HasOne</b> (lambda)	Configures the <i>one</i> side of a one-to-many or one-to-one relationship.
<b>WithOne</b> (lambda)	
<b>HasMany</b> (lambda)	Configures the <i>many</i> side of a one-to-many or many-to-many relationship.
<b>WithMany</b> (lambda)	
<b>HasForeignKey</b> <T>(l)	Specifies which property is the foreign key in a relationship.
<b>onDelete</b> (behavior)	Specifies how the database deals with related rows when a row is deleted. See figure 12-10.

## Description

- Relationships are defined with a primary key and a foreign key. The *primary key* (**PK**) uniquely identifies an entity, and the *foreign key* (**FK**) relates rows in another table to the primary key.
- Relationships in EF Core can be configured by convention, with data attributes, or by using the Fluent API.

Figure 12-6 How entities are related

## How to configure a one-to-many relationship

The first code example in figure 12-7 shows the simplest way to configure a one-to-many relationship by convention. Here, the relationship is created by nesting a Genre entity as a property of the Book entity.

This is a common way of creating a one-to-many relationship. However, this approach can cause some problems. For example, you need to specifically include the nested entity and all of its data even if all you need is the ID value. In addition, you can run into problems if the nested entity has data validation requirements.

As a result, it's typically a better practice to fully define the relationship by explicitly coding the foreign key property. The second example in this figure shows how this works. Here, the Book entity explicitly includes the foreign key property, GenreId. This Book entity also includes the Genre entity, but now it's clear that this property is a *navigation property*. In other words, it's a way to navigate to the values of the related entity from the primary entity.

When you fully define a relationship, it's a good practice to include a navigation property at each end of the relationship. To illustrate, the second example includes a Genre navigation property in the Book entity (the one side) and a Book collection property in the Genre entity (the many side).

Here, the example still configures the relationship by convention. However, fully defining the foreign key and navigation properties makes the code easier to work with. In addition, it makes the relationship between the two entities more clear.

Most one-to-many relationships can be configured by convention. As a result, the main reason to use data attributes is if your naming doesn't follow conventions. For instance, the third example in this figure uses a name of Category for the Genre property and a name of CategoryId for the foreign key property of the Genre entity. Because of that, it uses the ForeignKey and InverseProperty attributes to define the relationship.

The last example shows how to use the Fluent API to configure a one-to-many relationship between the Book and Genre entities. This statement starts at the Book end of the relationship. However, you could also start at the Genre end. Then, the statement would look like this:

```
modelBuilder.Entity<Genre>()
    .HasMany(g => g.Books)
    .WithOne(b => b.Genre);
```

For this to work, you need to fully define the foreign key and navigation properties in your entities.

Most of the time, you only use the Fluent API to configure a one-to-many relationship if you also need to provide configuration that can't be provided by convention or with data attributes. For example, if you need to configure how the database handles deletions, you can use the Fluent API to configure a one-to-many relationship as shown in figure 12-10.

## How to configure a one-to-many relationship by convention

```
public class Book {
    public int BookId { get; set; }

    public Genre Genre { get; set; }
}

public class Genre {
    public int GenreId { get; set; }
    public string Name { get; set; }
}
```

## How to fully define the one-to-many relationship by convention (recommended)

```
public class Book {
    public int BookId { get; set; }

    public int GenreId { get; set; }                                // foreign key property
    public Genre Genre { get; set; }                                // navigation property
}

public class Genre {
    public int GenreId { get; set; }
    public string Name { get; set; }

    public ICollection<Book> Books { get; set; }                // navigation property
}
```

## How to configure a one-to-many relationship with attributes

```
public class Book {
    public int BookId { get; set; }
    public int CategoryId { get; set; }

    [ForeignKey("CategoryId")]                                     // FK property in Book class
    [InverseProperty("Books")]                                    // nav property in Genre class
    public Genre Category { get; set; }
}

public class Genre {
    public int GenreId { get; set; }
    public string Name { get; set; }

    [InverseProperty("Category")]                                // nav property in Book class
    public ICollection<Book> Books { get; set; }
}
```

## How to configure a one-to-many relationship with the Fluent API

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Book>()
        .HasOne(b => b.Genre)                               // nav property in Book class
        .WithMany(g => g.Books);                          // nav property in Genre class
}
```

---

Figure 12-7 How to configure a one-to-many relationship

## How to configure a one-to-one relationship

If an entity has a one-to-one relationship with another entity, the data in the two entities could be stored in one entity, and the data in the two underlying database tables could be stored in one table. This is useful when you need to store large objects such as images, sound, and videos. Then, you can store those objects in a separate entity and underlying table and only join them when you need to retrieve the large object.

A one-to-one relationship can also be useful when an entity has both essential data and data that is frequently null. For example, suppose that in most cases, the only data you need to store for an author is the name. In a few cases, though, suppose you want to store data in addition to the name. Then, instead of storing null values for all of the authors that don't have this data, you can store the values for authors that do have this data in a separate table and only retrieve it when you need it.

The first example in figure 12-8 shows how to configure a fully defined one-to-one relationship by convention. This example works similarly to the second example in the previous figure. To start, each entity configures its primary key by convention and defines a navigation property. In addition, the AuthorBio entity configures its foreign key property by giving it the same name as the primary key of the Author entity.

Most one-to-one relationships can be configured by convention. However, if your naming doesn't follow conventions, you can use data attributes. For instance, you might not want to follow the EF conventions because you want to use the same name for the primary key in both entities. The second example shows how to do this. Here, the Author entity is unchanged, but the AuthorBio entity now uses its AuthorId property as its primary key and its foreign key.

The third example shows how to use the Fluent API to configure a one-to-one relationship. Typically, you only need to use this API if you also need to perform other tasks in addition to configuring the relationship. For example, you may need to split a table as shown by the fourth example.

*Table splitting* allows you to use two entities to represent the data that's stored in a single table. This can be useful when the table in the database contains a lot of columns, but you don't always want to retrieve all those columns. In this example, the author and author bio data is stored in a single table. However, this table is configured so its data is split between two entities, one that contains the data for authors and one that contains the data for author bios.

## How to configure a one-to-one relationship by convention

```

public class Author {
    public int AuthorId { get; set; }                      // primary key property
    public string Name { get; set; }

    public AuthorBio Bio { get; set; }                      // navigation property
}

public class AuthorBio {
    public int AuthorBioId { get; set; }                  // primary key property
    public int AuthorId { get; set; }                      // foreign key property
    public DateTime? DOB { get; set; }

    public Author Author { get; set; }                      // navigation property
}

```

## How to configure a one-to-one relationship with attributes

```

public class Author {
    // same as above
}

public class AuthorBio {
    [Key]
    public int AuthorId { get; set; }                      // PK and FK property
    public DateTime? DOB { get; set; }

    [ForeignKey("AuthorId")]
    public Author Author { get; set; }                      // FK property
                                                                // navigation property
}

```

## How to configure a one-to-one relationship with the Fluent API

```

protected override void OnModelCreating(ModelBuilder modelBuilder) {
    modelBuilder.Entity<Author>()
        .HasOne(a => a.Bio)                                // nav property in Author class
        .WithOne(ab => ab.Author)                          // nav property in AuthorBio class
        .HasForeignKey<AuthorBio>(ab => ab.AuthorId);   // FK property
}

```

## How to configure a one-to-one relationship within a single table

```

protected override void OnModelCreating(ModelBuilder modelBuilder) {
    modelBuilder.Entity<Author>()
        .HasOne(a => a.Bio)
        .WithOne(ab => ab.Author)
        .HasForeignKey<AuthorBio>(ab => ab.AuthorId);

    modelBuilder.Entity<Author>().ToTable("Authors");
    modelBuilder.Entity<AuthorBio>().ToTable("Authors");
}

```

---

Figure 12-8 How to configure a one-to-one relationship

## How to configure a many-to-many relationship

A many-to-many relationship is configured by using an intermediate entity called a *join entity* or a *linking entity*. This creates an intermediate table in the database called a *join table* or a *linking table*.

The linking entity has a one-to-many relationship with the two entities in the many-to-many relationship. In other words, a many-to-many relationship is broken down into two one-to-many relationships.

Unlike the other two relationships, a many-to-many relationship can only be configured with the Fluent API. Figure 12-9 shows how to configure a many-to-many relationship between the Book entity and the Author entity.

The first example in this figure presents the Book and Author entities that have a many-to-many relationship. Each of these entities contains a navigation property of a collection of BookAuthor entities. These navigation properties represent the *many* side of the two one-to-many relationships.

The second example in this figure presents the linking entity, in this case, the BookAuthor entity. This entity has an int property named BookId, which is a foreign key to the Book entity. In addition, it has an int property named AuthorId, which is a foreign key to the Author entity. Both of these properties together are the primary key. A primary key like this one that consists of more than one property is called a *composite primary key*.

The BookAuthor entity also contains a Book navigation property and an Author navigation property. Each of these navigation properties represents the *one* side of the two one-to-many relationships.

The third example shows how to use the Fluent API to configure the many-to-many relationship between the Book and Author entities. The first statement configures the composite primary key for the BookAuthor entity. To do this, it creates an anonymous object with the two properties of the key and passes it as part of a lambda expression to the HasKey() method. The second statement configures the one-to-many relationship between BookAuthor and Book entities and identifies the foreign key. The third statement works similarly, but it configures the one-to-many relationship between BookAuthor and Author entities.

## The entities to be linked

```
public class Book {
    public int BookId { get; set; }
    public string Title { get; set; }

    // navigation property to linking entity
    public ICollection<BookAuthor> BookAuthors { get; set; }
}

public class Author {
    public int AuthorId { get; set; }
    public string Name { get; set; }

    // navigation property to linking entity
    public ICollection<BookAuthor> BookAuthors { get; set; }
}
```

## The linking entity

```
public class BookAuthor {
    // composite primary key
    public int BookId { get; set; }           // foreign key for Book
    public int AuthorId { get; set; }          // foreign key for Author

    // navigation properties
    public Book Book { get; set; }
    public Author Author { get; set; }
}
```

## How to configure a many-to-many relationship with the Fluent API

```
protected override void OnModelCreating(ModelBuilder modelBuilder) {
    // composite primary key for BookAuthor
    modelBuilder.Entity<BookAuthor>()
        .HasKey(ba => new { ba.BookId, ba.AuthorId });

    // one-to-many relationship between Book and BookAuthor
    modelBuilder.Entity<BookAuthor>()
        .HasOne(ba => ba.Book)
        .WithMany(b => b.BookAuthors)
        .HasForeignKey(ba => ba.BookId);

    // one-to-many relationship between Author and BookAuthor
    modelBuilder.Entity<BookAuthor>()
        .HasOne(ba => ba.Author)
        .WithMany(a => a.BookAuthors)
        .HasForeignKey(ba => ba.AuthorId);
}
```

## Description

- A many-to-many relationship requires a third entity known as a *join entity* or *linking entity*. The linking entity has two one-to-many relationships with the entities to be joined.
- The linking entity has a *composite primary key* that consists of the primary key of each linked entity.

---

Figure 12-9 How to configure a many-to-many relationship

## How to control delete behavior

When an app deletes a row from a database, related rows that are dependent on that row can become corrupted. For example, if an app deletes the genre “Novel” from the Genre table, all the rows in the Books table whose genre is “Novel” have invalid data.

To prevent this, most databases throw an exception when you try to delete a row that has dependent, or child, rows. However, most databases also allow you to configure foreign keys with cascading deletes. A *cascading delete* causes all child rows to be automatically deleted when a parent row is deleted.

By default, Code First development configures a foreign key with cascading deletes if the foreign key is not nullable. This is true whether you define the relationship by convention, with data attributes, or using the Fluent API.

Sometimes, cascading deletes are what you want. For example, when you delete a Book, you also want to delete related BookAuthor rows. Similarly, when you delete an Author, you also want to delete the related AuthorBio row.

Sometimes, though, cascading deletes are *not* what you want. For example, when you delete the “Novel” genre, you *don’t* want to delete all books that are novels from the Books table. In this case, it’s better for the database to throw an exception. This helps to prevent you from accidentally deleting data that you don’t want to delete.

You can use the `OnDelete()` method of the Fluent API to configure how dependent rows are handled when a parent row is deleted. This method accepts a value of the `DeleteBehavior` enum as an argument. The table at the top of figure 12-10 shows the values of this enum.

The `Cascade` value configures the foreign key so all dependent rows are deleted. Again, this is the default behavior in Code First development for foreign keys that are not nullable.

The `SetNull` value configures the foreign key so that when a parent row is deleted, the foreign key value of any dependent rows is set to null. This only works if the foreign key is nullable, and it isn’t usually what you want.

The `Restrict` value configures the foreign key so nothing happens to dependent rows when the parent row is marked for deletion. In other words, it “turns off” the cascading delete. If a property remains in this state when `SaveChanges()` is called, EF throws an exception, which is often what you want.

The code example below the table in this figure shows how to use the `OnDelete()` method. Here, the `GenreId` foreign key property in the `Book` entity is configured so it doesn’t have a cascading delete. As a result, if you try to delete a genre that has related books, EF throws an exception.

## The values of the DeleteBehavior enum

Value	Description
<b>Cascade</b>	Deletes dependent rows automatically. This is EF Core's default behavior for foreign keys that are not nullable.
<b>SetNull</b>	Sets the value of the foreign key in the dependent row to null. This is only possible when the foreign key is nullable. If the foreign key is not nullable, this causes an exception to be thrown.
<b>Restrict</b>	Prevents deletion of dependent rows. If a property remains in this state when SaveChanges() is called, this causes an exception to be thrown. This is EF Core's default behavior for foreign keys that are nullable.

## How to configure a relationship to do nothing on delete

```
protected override void OnModelCreating(ModelBuilder modelBuilder)
{
    modelBuilder.Entity<Book>()
        .HasOne(b => b.Genre)
        .WithMany(g => g.Books)
        .OnDelete(DeleteBehavior.Restrict); // don't delete dependent rows
}
```

### Description

- When a row is deleted from a database, related rows that are dependent on that row can become corrupted. To prevent this, most databases throw an exception when you try to delete a row that has dependent rows.
- Most databases also allow you to configure *cascading deletes*, which cause dependent rows to be automatically deleted.
- You can use the OnDelete() method of the Fluent API to configure how dependent rows are handled when a parent row is deleted.

---

Figure 12-10 How to control delete behavior

## The Bookstore database classes

---

Now that you know how to code entity classes that are related to each other, you're ready to see the entity, context, and configuration classes for the Bookstore website that's presented in the next chapter.

### The entity classes

---

Figure 12-11 shows the classes for the Author, Book, BookAuthor, and Genre entities that are used by the Bookstore website. For the most part, these entities are configured by convention. For example, the Author, Book, and Genre classes configure their primary keys by convention. Similarly, these classes configure their foreign keys by convention and include navigation properties to make it easy to work with related entities.

However, the BookAuthor class defines a linking entity that uses a composite primary key that consists of two properties. As a result, the context class shown in figure 12-12 uses the Fluent API to configure the composite primary key and the foreign keys for the BookAuthor entity.

Most of the properties of the Author, Book, and Genre classes are decorated with data attributes for validation. These attributes are also applied to the database when you create it, which is usually what you want. For example, the Required attribute causes the corresponding database column to not allow null values. Similarly, the StringLength attribute specifies the maximum number of characters that can be stored for a string column.

### The context and configuration classes

---

Figure 12-12 shows the context class for the Bookstore website. This class begins by defining the DbSet properties for the Author, Book, BookAuthor, and Genre entities presented in figure 12-11.

After defining the DbSet properties, the OnModelCreating() method starts by using the Fluent API to define the composite primary key and the foreign keys for the BookAuthor class. This configures the many-to-many relationship between the Book and Author entities.

Next, this method uses the Fluent API to turn off cascading deletes for the Genre entity. That way, if you attempt to delete a genre that is related to one or more books, EF won't delete the genre. Instead, it will throw an exception.

Finally, this method seeds the database tables with some initial data by applying the code that's stored in the configuration classes shown in figure 12-13. These configuration classes use the HasData() method of the entity parameter to specify the initial data for the database table that corresponds to the entity class. Note that you can seed the tables in any sequence, regardless of how they're related to each other.

### The Author class

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using Microsoft.AspNetCore.Mvc;

namespace Bookstore.Models
{
    public class Author
    {
        public int AuthorId { get; set; }

        [Required(ErrorMessage = "Please enter a first name.")]
        [StringLength(200)]
        public string FirstName { get; set; }

        [Required(ErrorMessage = "Please enter a last name.")]
        [StringLength(200)]
        [Remote("CheckAuthor", "Validation", "",
            AdditionalFields = "FirstName, Operation")]
        public string LastName { get; set; }

        // read-only property
        public string FullName => $"{FirstName} {LastName}";

        // navigation property
        public ICollection<BookAuthor> BookAuthors { get; set; }
    }
}
```

### The Book class

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;

namespace Bookstore.Models
{
    public partial class Book
    {
        public int BookId { get; set; }

        [Required(ErrorMessage = "Please enter a title.")]
        [StringLength(200)]
        public string Title { get; set; }

        [Range(0.0, 1000000.0, ErrorMessage = "Price must be more than 0.")]
        public double Price { get; set; }

        [Required(ErrorMessage = "Please select a genre.")]
        public string GenreId { get; set; }      // foreign key property
        public Genre Genre { get; set; }         // navigation property

        // navigation property
        public ICollection<BookAuthor> BookAuthors { get; set; }
    }
}
```

---

Figure 12-11 The entity classes for the Bookstore database (part 1)

## The BookAuthor class

```
namespace Bookstore.Models
{
    public class BookAuthor
    {
        // composite primary key and foreign keys
        public int BookId { get; set; }
        public int AuthorId { get; set; }

        // navigation properties
        public Author Author { get; set; }
        public Book Book { get; set; }
    }
}
```

## The Genre class

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using Microsoft.AspNetCore.Mvc;

namespace Bookstore.Models
{
    public class Genre
    {
        [StringLength(10)]
        [Required(ErrorMessage = "Please enter a genre id.")]
        [Remote("CheckGenre", "Validation", "")]

        public string GenreId { get; set; }

        [StringLength(25)]
        [Required(ErrorMessage = "Please enter a genre name.")]
        public string Name { get; set; }

        public ICollection<Book> Books { get; set; }
    }
}
```

---

Figure 12-11 The entity classes for the Bookstore database (part 2)

## The BookstoreContext class

```
using Microsoft.EntityFrameworkCore;

namespace Bookstore.Models
{
    public class BookstoreContext : DbContext
    {
        public BookstoreContext(DbContextOptions<BookstoreContext> options)
            : base(options)
        { }

        public DbSet<Author> Authors { get; set; }
        public DbSet<Book> Books { get; set; }
        public DbSet<BookAuthor> BookAuthors { get; set; }
        public DbSet<Genre> Genres { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            // BookAuthor: set composite primary key
            modelBuilder.Entity<BookAuthor>()
                .HasKey(ba => new { ba.BookId, ba.AuthorId });

            // BookAuthor: set foreign keys
            modelBuilder.Entity<BookAuthor>().HasOne(ba => ba.Book)
                .WithMany(b => b.BookAuthors)
                .HasForeignKey(ba => ba.BookId);
            modelBuilder.Entity<BookAuthor>().HasOne(ba => ba.Author)
                .WithMany(a => a.BookAuthors)
                .HasForeignKey(ba => ba.AuthorId);

            // Book: remove cascading delete with Genre
            modelBuilder.Entity<Book>().HasOne(b => b.Genre)
                .WithMany(g => g.Books)
                .OnDelete(DeleteBehavior.Restrict);

            // seed initial data
            modelBuilder.ApplyConfiguration(new SeedGenres());
            modelBuilder.ApplyConfiguration(new SeedBooks());
            modelBuilder.ApplyConfiguration(new SeedAuthors());
            modelBuilder.ApplyConfiguration(new SeedBookAuthors());
        }
    }
}
```

---

Figure 12-12 The BookstoreContext class

### The SeedAuthors class

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace Bookstore.Models
{
    internal class SeedAuthors : IEntityTypeConfiguration<Author>
    {
        public void Configure(EntityTypeBuilder<Author> entity)
        {
            entity.HasData(
                new Author { AuthorId = 1, FirstName = "Michelle",
                            LastName = "Alexander" },
                new Author { AuthorId = 2, FirstName = "Stephen E.",
                            LastName = "Ambrose" },
                ...
                new Author { AuthorId = 26, FirstName = "Seth",
                            LastName = "Grahame-Smith" }
            );
        }
    }
}
```

### The SeedBooks class

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace Bookstore.Models
{
    internal class SeedBooks : IEntityTypeConfiguration<Book>
    {
        public void Configure(EntityTypeBuilder<Book> entity)
        {
            entity.HasData(
                new Book { BookId = 1, Title = "1776", GenreId = "history",
                           Price = 18.00 },
                new Book { BookId = 2, Title = "1984", GenreId = "scifi",
                           Price = 5.50 },
                ...
                new Book { BookId = 29,
                           Title = "Harry Potter and the Sorcerer's Stone",
                           GenreId = "novel", Price = 9.75 }
            );
        }
    }
}
```

---

Figure 12-13 The configuration classes for the Bookstore database (part 1)

### The SeedBookAuthors class

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace Bookstore.Models
{
    internal class SeedBookAuthors : IEntityTypeConfiguration<BookAuthor>
    {
        public void Configure(EntityTypeBuilder<BookAuthor> entity)
        {
            entity.HasData(
                new BookAuthor { BookId = 1, AuthorId = 18 },
                new BookAuthor { BookId = 2, AuthorId = 20 },
                ...
                new BookAuthor { BookId = 28, AuthorId = 4 },
                new BookAuthor { BookId = 28, AuthorId = 26 },
                new BookAuthor { BookId = 29, AuthorId = 25 }
            );
        }
    }
}
```

### The SeedGenres class

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Metadata.Builders;

namespace Bookstore.Models
{
    internal class SeedGenres : IEntityTypeConfiguration<Genre>
    {
        public void Configure(EntityTypeBuilder<Genre> entity)
        {
            entity.HasData(
                new Genre { GenreId = "novel", Name = "Novel" },
                new Genre { GenreId = "memoir", Name = "Memoir" },
                new Genre { GenreId = "mystery", Name = "Mystery" },
                new Genre { GenreId = "scifi", Name = "Science Fiction" },
                new Genre { GenreId = "history", Name = "History" }
            );
        }
    }
}
```

---

Figure 12-13 The configuration classes for the Bookstore database (part 2)

## How to create code from a database

So far in this book, you have learned how to use Code First development to create a database from your context and entity classes. Sometimes, though, you already have a database. In that case, you need to create context and entity classes from your database. This approach is called *Database First* development.

### How to generate DB context and entity classes

To get started with Database First development, you can use the Scaffold-DbContext command to generate the code for the context and entity classes. The table at the top of figure 12-14 presents some of the parameters that are available for this command. The first two parameters, -Connection and -Provider, are required. The rest of the parameters are optional.

It's a best practice to store the connection string in the appsettings.json file. Then, you can use the name of the connection string setting as the value of this parameter. However, if that doesn't work, you can specify a string literal for the connection string as shown by the fourth example in this figure.

The -DataAnnotations parameter indicates whether you want to perform configuration by adding data attributes to your entity classes. If you include this parameter, EF uses data attributes to perform configuration as much as possible. However, EF may still use the Fluent API for some of the configuration. That's because it's often not possible to do all configuration with attributes.

The -Force parameter indicates whether you want to overwrite existing files. This is useful when you want to regenerate your code files after making changes to the database. If you don't include this flag and the files already exist, the PMC displays an error message.

The first example in this figure shows a connection string named BookstoreContext that's stored in the appsettings.json file. Due to space limitations, this string is displayed on two lines here. For it to work correctly, though, it must be stored on one line in the file.

The second and third examples show how to generate class files from a SQL Server database, output these files to the Models\DataLayer folder, and overwrite any existing files. Here, the second example includes the flags for the two required parameters, and the third omits them. If you omit the flags, you need to specify the parameters in the sequence shown here.

This figure finishes by showing the generated files in the Models\DataLayer folder. The code for these files includes a namespace based on the folder path. So, if these files are in a project named Bookstore, the generated classes are in the Bookstore.Models.DataLayer namespace. In addition, the names of these classes end with an s. That's because the names of the tables in the Bookstore database end with an s, and the generated class names match the table names.

## Parameters for the Scaffold-DbContext command

Parameter	Description
<b>-Connection</b>	The connection string. You can use name=ConnectionStringName to specify the name of a connection string that's stored in the appsettings.json file. This parameter is required.
<b>-Provider</b>	The provider, often a NuGet package name. This parameter is required.
<b>-OutputDir</b>	The folder to store the generated files. The default is the root folder for the project.
<b>-DataAnnotations</b>	Adds data attributes to entity classes where possible. Otherwise, EF uses the Fluent API in the DB context class to perform all configuration.
<b>-Force</b>	Overwrites existing files. Otherwise, EF doesn't overwrite existing files.

### A connection string that's stored in the appsettings.json file

```
"ConnectionStrings": {
    "BookstoreContext": "Server=(localdb)\\mssqllocaldb;
        Database=Bookstore;Trusted_Connection=True;MultipleActiveResultSets=true"
}
```

### An EF command that generates entity classes from a Sql Server database

```
PM> Scaffold-DbContext -Connection name=BookstoreContext
-Provider Microsoft.EntityFrameworkCore.SqlServer
-OutputDir Models\DataLayer -DataAnnotations -Force
```

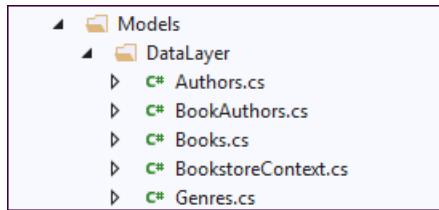
### The same command with the flags for the required parameters omitted

```
PM> Scaffold-DbContext name=BookstoreContext
Microsoft.EntityFrameworkCore.SqlServer
-OutputDir Models\DataLayer -DataAnnotations -Force
```

### A -Connection parameter with a string literal for the connection string

```
-Connection "Server=(localdb)\\mssqllocaldb;Database=Bookstore;
    Trusted_Connection=True;MultipleActiveResultSets=true"
```

### The generated files in the Models\DataLayer folder



### Description

- You can use the Scaffold-Database command to generate context and entity classes based on an existing database.
- You can omit the flag for required parameters.
- When specifying the connection string, it's considered a best practice to use a connection string that's stored in the appsettings.json file.

Figure 12-14 How to generate the DB context and entity classes

## How to configure a generated DB context class

---

When you use the Scaffold-DbContext command to generate context and entity classes, the `OnConfiguring()` method of the context class contains the connection string and provider information. Usually, though, you need to make some adjustments before your project is ready to run.

The first code example in figure 12-15 shows the `OnConfiguring()` method that's generated by a Scaffold-DbContext command like the one shown in the previous figure. Here, the `UseSqlServer()` method specifies that the provider is a SQL Server database. And the argument that's passed to this method specifies the name of the connection string that's stored in the `appsettings.json` file as shown in the previous figure.

However, your app won't run until you configure it to use dependency injection with your generated DB context. To do that, you can modify the `Startup.cs` file as shown in the second code example in this figure. This works like the Code First example presented in figure 4-6.

Since the `OnConfiguring()` method sets the provider (SQL Server) and the name of the connection string, it duplicates configuration options that are now stored in the `Startup.cs` file. If you want, you can leave the `OnConfiguring()` method as shown in this figure. That's because the `OnConfiguring()` method uses an `if` statement to only configure database options if they have not already been configured in the `Startup.cs` file. However, if you prefer, you can remove this statement from the `OnConfiguring()` method as shown in the third example.

## The OnConfiguring() method in the generated context class

```
protected override void OnConfiguring(
    DbContextOptionsBuilder optionsBuilder)
{
    if (!optionsBuilder.IsConfigured)
    {
        optionsBuilder.UseSqlServer("name=BookstoreContext");
    }
}
```

## The Startup.cs file that injects the DB context into the app

```
using Microsoft.EntityFrameworkCore;
using Bookstore.Models.DataLayer;

public class Startup
{
    public Startup(IConfiguration configuration)
    {
        Configuration = configuration;
    }

    public IConfiguration Configuration { get; }

    public void ConfigureServices(IServiceCollection services)
    {
        ...
        services.AddDbContext<BookStoreContext>(options =>
            options.UseSqlServer(
                Configuration.GetConnectionString("BookstoreContext")));
        ...
    }
}
```

## The OnConfiguring() method after it has been cleaned up

```
protected override void OnConfiguring(
    DbContextOptionsBuilder optionsBuilder)
{
    base.OnConfiguring(optionsBuilder);
}
```

## Description

- When you use the Scaffold-Database command to create DB context and entity classes, it adds configuration code to the OnConfiguring() method of the context class.
- To get the DB context to work, you need to edit the Startup.cs file so it injects the context into the app.
- For this code to work, the appsettings.json file must store the specified connection string. If it doesn't, you can edit this file so it contains the specified connection string.

---

Figure 12-15 How to configure a generated DB context class

## How to modify a generated entity class

---

When you work with Database First development, it's common to enhance the entity classes that it generates. For instance, you may want to add data validation attributes. Or, you may want to add more properties such as read-only properties that indicate the state of the entity.

However, if you make changes to the database, it's common to need to regenerate the entity classes. When this happens, though, you lose any additions you've made to those classes. Fortunately, you can use partial classes to avoid this problem.

The first example in figure 12-16 presents an entity class named Books that was generated by EF Core. You should notice two important things about this class. First, it's a partial class. Second, EF generated it in a namespace that corresponds to the folder structure for the class.

The second example presents another partial class named Books that adds to the generated partial class of the same name. This works because C# compiles these two partial classes into a single class named Books.

You should notice three important things about this second partial class. First, it's in the same namespace as the generated class. However, because the class file has the same name as the generated Books class file, it must be stored in a different folder than this file.

Second, it adds a read-only property named HasTitle. This property returns a Boolean value that indicates whether the Title property has a value. Because this custom property is not in the generated file, it won't be lost if the Books class is regenerated from the database.

Third, it's decorated with the ModelMetadataType attribute. This attribute uses the typeof operator to pass the type of a class as an argument. The class that it passes contains the attributes, or metadata, that should be applied to the generated entity class. In this case, the attributes in a class named BooksMetadata are applied. This class is shown in the third example in this figure.

The BooksMetadata class contains the data validation attributes for the generated Books entity class. Again, since these attributes are in a separate class file, they won't be lost if the Books class needs to be regenerated. Also, this class is in the same namespace as both of the partial Books classes. That's because all three of these classes need to be in the same namespace for this technique to work.

At this point, you might be wondering why you can't just include the data validation attributes in the Books partial class that adds the HasTitle property to the generated partial class. The reason is that, to add data validation attributes, you need to code the property you want to validate. For instance, you would need to code a Title property if you wanted to decorate it with data validation attributes. But, because the Title property is already coded in the generated Books partial class, the compiler won't let you code it a second time in another partial class. As a result, you must add the data validation in a different class. Then, you must use the ModelMetadataType attribute to apply that validation to the properties of the Books class.

### A partial class generated by EF Database First

```
namespace Bookstore.Models.DataLayer
{
    public partial class Books
    {
        [Key]
        public int BookId { get; set; }

        [Required]
        [StringLength(200)]
        public string Title { get; set; }

        public double Price { get; set; }
        ...
    }
}
```

### A partial class that adds a metadata class and a read-only property

```
using Microsoft.AspNetCore.Mvc;

namespace Bookstore.Models.DataLayer
{
    [ModelMetadataType(typeof(BooksMetadata))]
    public partial class Books
    {
        public bool HasTitle => !string.IsNullOrEmpty(Title);
    }
}
```

### A metadata class that adds validation attributes for two properties

```
using System.ComponentModel.DataAnnotations;

namespace Bookstore.Models.DataLayer
{
    public class BooksMetadata
    {
        [RegularExpression("^[a-zA-Z0-9 _.,!':]+$",
            ErrorMessage = "Title may not contain special characters.")]
        public string Title { get; set; }

        [Range(0.0, 1000000.0,
            ErrorMessage = "Price must be greater than zero.")]
        public double Price { get; set; }
    }
}
```

### Description

- When EF Core generates entity classes, it creates partial classes.
- If you add data validation attributes or custom properties directly to a generated entity class, those additions are lost if you regenerate the classes when the database changes. To prevent that, you can write your own partial classes that contain any additions you want to make.

---

Figure 12-16 How to modify a generated entity class

# How to work with data in a database

Chapter 4 introduced some basic skills for working with the data in a database. Now, you'll review those skills and learn some new ones.

## How to query data

You can use *LINQ to Entities* to query the DbSet properties of a context class. The first code example in figure 12-17 shows a context property named context that's used by subsequent examples. This context has DbSet properties named Books and Authors.

The second set of examples shows how to create and execute a query. When you do that using two statements, you create a query variable of the IQueryable<T> type that holds a query, not data. Then, when you execute the query, the data is retrieved from the database and stored in an object of the IEnumerable<T> type. When you do that using one statement, you can use the var keyword to declare the variable where the results are stored so the compiler will automatically determine the type. In this case, it uses the List<Book> type, which is usually what you want.

The third set of examples shows how to sort the results of a query. The first statement uses the OrderBy() method to sort the results in ascending order by title, and the second uses the OrderByDescending() method to sort the results in descending order by title.

The fourth set of examples shows how to filter a query to limit the number of rows it returns. Most of these examples use the Where() method to specify the criteria to filter by. However, the second statement uses the Find() method to get a book by its ID. This is a shortcut for the first statement that uses the Where() method. In these statements, the Find() and FirstOrDefault() methods execute the query and return the requested data.

The fourth set of examples finishes by showing how to conditionally filter the results by multiple criteria. First, this code declares an initial query that returns all books. This code doesn't use the var keyword to implicitly type the query variable because the compiler would infer the DbSet<T> type instead of the IQueryable<T> type. Then, this code uses if statements to build the where clause. If none of these conditions are true, this code executes the initial query to retrieve all books.

The fifth example shows how to get a subset of results using the Skip() and Take() methods. This is common in scenarios where you want to allow the user to be able to scroll through pages of results.

The sixth example shows how to disable change tracking. Normally, the DbContext object tracks all changes that are made to the data it retrieves. In read-only scenarios, though, this isn't necessary. As a result, you can use the AsNoTracking() method to disable it.

The last example shows how to get a random book. To do that, the OrderBy() method sorts books randomly by assigning a new globally unique identifier (GUID) to each book and sorting on that value. Then, this example gets the first book from the newly sorted list of books.

## The context class used in the following examples

```
private BookstoreContext context { get; set; }
```

### Code that creates and executes a query

#### In two statements

```
IQueryable<Book> query = context.Books;           // create the query
IEnumerable<Book> books = query.ToList();         // execute the query
```

#### In one statement

```
var books = context.Books.ToList();                // implicit typing is common
```

### Code that sorts the results

#### Sort by Title in ascending order (A to Z)

```
var books = context.Books.OrderBy(b => b.Title).ToList();
```

#### Sort by Title in descending order (Z to A)

```
var books = context.Books.OrderByDescending(b => b.Title).ToList();
```

### Code that filters the results

#### Get a single book by ID

```
var book = context.Books.Where(b => b.BookId == 1).FirstOrDefault();
var book = context.Books.Find(1);      // shortcut for above
```

#### Get a list of books by genre

```
var books = context.Books.Where(b => b.Genre.Name == "Mystery").ToList();
```

#### Get a list of books in a price range

```
var books = context.Books.Where(b => b.Price > 10 && b.Price < 20).ToList();
```

#### Conditionally filter by multiple criteria

```
// build the query (can't use implicit typing here)
IQueryable<Book> query = context.Books;
if (selectedMaxPrice != null)
    query = query.Where(b => b.Price < selectedMaxPrice);
if (selectedGenre != null)
    query = query.Where(b => b.Genre.Name == selectedGenre);

// execute the query
var books = query.ToList();
```

### Code that gets a subset of results

```
int pageNumber = 2, booksPerPage = 4;
var books = context.Books.Skip((pageNumber - 1) * booksPerPage)
                        .Take(booksPerPage)
                        .ToList();
```

### Code for a read-only query that disables change tracking

```
var books = context.Books.AsNoTracking().ToList();
```

### Code that gets a random book

```
var randBook = context.Books.OrderBy(r => Guid.NewGuid()).FirstOrDefault();
```

---

Figure 12-17 How to query data

## How to work with projections and related entities

When you query data as described in the last figure, the query retrieves all of the properties in an entity, whether you need them or not. Usually, this is fine. Sometimes, though, you may have entities with so many properties that it's inefficient to retrieve all of them. Or, you may have properties with sensitive data, like passwords, that you don't want to retrieve.

One way to limit the number of properties returned is to use table splitting as mentioned earlier in this chapter. Another way to do that is to use projections. A *projection* uses the LINQ Select() method to retrieve only some of an entity's properties.

One way to create a projection is to use an anonymous type like the one shown in the first example in figure 12-18. This type consists of the AuthorId property of the Authors entity and a property that's a combination of the FirstName and LastName properties. Because the second property doesn't match an entity property, it must be explicitly named as shown here.

Anonymous types are convenient and quick, but they can be hard to work with in an MVC app. That's because they're hard to pass to views, as the error message below the first example shows.

A better way to work with projections in MVC is to use a simple *data transfer object (DTO)* that's designed to transfer data from one place to another. For example, the DTO in the second set of examples provides the value and text for a drop-down list. This allows the Select() method to create a new instance of the DTO rather than creating an instance of an anonymous type. Then, it can pass the DTO to a view.

In addition to limiting the properties that are returned, you may sometimes want or need to include the data for related entities in your query. This is shown by the third set of examples in this figure. In the first example, the first Include() method accepts a lambda expression that specifies the entity to include. With this technique, you can include multiple related entities by chaining several Include() method calls. The second Include() method includes the same entity as the first by passing a string rather than a lambda expression. With this technique, you can include multiple entities by separating the strings with commas.

The second example shows how to retrieve related entities that are nested more than one layer deep. Here, the first Include() method gets the related BookAuthors entities. After that, the ThenInclude() method gets the Author entity that's related to each included BookAuthors entity. The second Include() method includes the same nested entities using a string rather than a lambda expression. With this technique, the nested entities are represented with dot notation.

The advantage of using lambda expressions is that they're checked at compile time. As a result, they're less error-prone because the IDE alerts you right away if you make a typo or the entity doesn't exist. By contrast, strings aren't checked until runtime. However, the string option can give you more flexibility as described later in this chapter.

## How to create a projection with an anonymous type

```
var authors = context.Authors
    .Select(a => new {
        a.AuthorId,                                     // can infer property name
        Name = a.FirstName + ' ' + a.LastName          // must specify property name
    })
    .ToList();
```

## Error when you pass the projection to a view that expects a list of Authors

```
InvalidOperationException: The model item passed into the ViewDataDictionary
is of type 'System.Collections.Generic.List`1[<>f__AnonymousType0`2 [System.
Int32, System.String]]', but this ViewDataDictionary instance requires a
model item of type System.Collections.Generic.IEnumerable`1 [BookList.
Models.Author].
```

## How to create a projection with a concrete type

### The concrete type

```
public class DropdownDTO
{
    public string Value { get; set; }
    public string Text { get; set; }
}
```

### The projection

```
var authors = context.Authors
    .Select(a => new DropdownDTO {
        Value = a.AuthorId.ToString(),
        Text = a.FirstName + ' ' + a.LastName
    })
    .ToList();
```

## Code that includes related entities

### Two techniques for getting a Book and its related Genre

```
var books = context.Books.Include(b => b.Genre).ToList();
var books = context.Books.Include("Genre").ToList();
```

### Two techniques for getting a Book and each related Author

```
var books = context.Books.Include(b => b.BookAuthors)
    .ThenInclude(ba => ba.Author)
    .ToList();
var books = context.Books.Include("BookAuthors.Author").ToList();
```

## Description

- A *projection* allows you to retrieve a subset of the properties of an entity.
- To create a projection, you can use an anonymous type or a concrete type. However, anonymous types can be hard to use in views.
- You can use the `Include()` and `ThenInclude()` methods to include related entities in your query. These methods accept lambda expressions to identify the entities to include.
- The `Include()` method also accepts a string literal to identify the entities to include.

Figure 12-18 How to work with projections and include related entities

## How to insert, update, and delete data

In addition to querying data, you need to know how to insert, update, and delete data. Fortunately, the methods of the DbSet and DbContext classes make that easy to do. You learned about these methods in chapter 4, but now figure 12-19 reviews them and explains them in more detail.

The Add(), Modify(), and Delete() methods of the DbSet class only affect the DbSet. The Add() method adds a new entity to the DbSet and marks it as Added, and the Update() and Remove() methods mark an existing entity as Modified or Deleted. To mark an entity, EF uses the EntityState enum. In most cases, you can let EF mark these changes for you automatically. However, you can also set the state of an entity manually, if necessary.

The SaveChanges() method of the DbContext class, by contrast, affects the underlying database. When you call this method, all the Added, Modified, and Deleted actions pending in the DbSet properties are executed against the database.

The code examples show how to use these methods. The first example adds a new entity, the second and third examples update an existing entity, and the fourth example deletes an existing entity. While this code is easy to follow, you should notice a few things.

First, although these examples all perform a single action and then call SaveChanges(), that's not a requirement. You can make several calls to Add(), Update(), or Delete() before you call SaveChanges(). In fact, in some scenarios, it can improve efficiency to only call SaveChanges() once after all your modifications are done.

Second, when it comes to updates, the second example shows how to work with a *disconnected scenario* because it's commonly used with web apps. In this scenario, the Edit() action method gets the Book entity from an HTTP POST request. As a result, the code in this figure must use Update() to update the state of the entity before calling SaveChanges().

With desktop apps, it's more common to use a *connected scenario*. However, a connected scenario can also occur in web apps as shown by the third example. Here, the first statement of the Edit() action method uses the Find() method to retrieve the Book entity from the database. This causes the context to track the state of this entity. As a result, you don't need to call the Update() method. Instead, this code just sets the new price for the book and calls the SaveChanges() method. This generates SQL that's slightly more efficient than calling Update(). However, if you wanted to make the code easier to read and understand, you could add a statement that calls Update() before calling SaveChanges().

Finally, when you delete a parent entity in EF Core, it's common for all of the related child entities to be automatically deleted too. That's because EF Core enables cascading deletes by default for foreign keys that are not nullable. However, if you disable cascading deletes as described earlier in this chapter, you'll need to delete all the child entities before you can delete the parent entity.

### Three of the methods of the DbSet class

Method	Description
<code>Add(entity)</code>	Adds the specified entity to the DbSet and sets its state to Added.
<code>Update(entity)</code>	Sets the state of the specified entity to Modified.
<code>Remove(entity)</code>	Sets the state of the specified entity to Deleted.

### One of the methods of the DbContext class

Method	Description
<code>SaveChanges()</code>	Saves all changes to the underlying database.

### Code that adds a new entity

```
[HttpPost]
public IActionResult Add(Book book)
{
    context.Books.Add(book);
    context.SaveChanges();
    return RedirectToAction("List", "Book");
}
```

### Code that updates an existing entity in a disconnected scenario

```
[HttpPost]
public IActionResult Edit(Book book) // Book object is disconnected
{
    context.Books.Update(book); // call to Update() required
    context.SaveChanges();
    return RedirectToAction("List", "Book");
}
```

### Code that updates an existing entity in a connected scenario

```
[HttpPost]
public IActionResult Edit(int id, double price)
{
    Book book = context.Books.Find(id); // Book object is connected
    book.Price = price; // call to Update() not required
    context.SaveChanges();
    return RedirectToAction("List", "Book");
}
```

### Code that deletes an entity

```
[HttpPost]
public IActionResult Delete(Book book)
{
    context.Books.Remove(book);
    context.SaveChanges(); // related data deleted if cascade delete on
    return RedirectToAction("List", "Book");
}
```

---

Figure 12-19 How to insert, update, and delete data

## How to handle concurrency conflicts

Concurrency allows two or more users to work with a database at the same time. However, if two users retrieve and then attempt to update the same entity (row in a table), their updates may conflict with each other, and you need to handle this *concurrency conflict*.

In EF Core, you have two options for concurrency control. The default option is called “*last in wins*”. This option doesn’t perform any checking. Instead, the last update overwrites any previous changes. In some cases, this option is adequate, but it can lead to corrupted data.

The other option is called *optimistic concurrency*. It checks whether a row has been changed since it was retrieved. If so, EF refuses the update or deletion and throws an exception. Then, the app can handle this exception. You can use data attributes or the Fluent API to configure your apps to use optimistic concurrency.

## How to check for concurrency conflicts

One common way to enable optimistic concurrency is to check an entire entity for changes. To do that, you add and configure a *rowversion* property as shown in the first and second examples of figure 12-20. The first example decorates a property named RowVersion with the `Timestamp` attribute, and the second example uses the `IsRowVersion()` method of the Fluent API to add the same property. For a *rowversion* property to work, it must be declared as a `byte[]` type.

After you’ve added and configured a *rowversion* property and updated the database, every query retrieves the value of this property. In addition, EF automatically updates its value whenever any column in an entity’s underlying table row is modified. That way, when your app attempts to update or delete a row, EF compares the *rowversion* value that was retrieved with the initial query to its current value. If they don’t match, EF refuses the action and throws an exception. For this to work, you need to make sure the original value is stored in the view and posted with the update or delete. Usually, you do this with a hidden field.

When you’re testing your app to make sure it handles concurrency conflicts the way you want it to, you may need to simulate a situation in which a row is modified after it’s retrieved but before it’s updated or deleted. To do that, you can use the `ExecuteSqlRaw()` method to execute a standard SQL statement like the one shown in the third example of this figure.

Another way to enable optimistic concurrency is to check an individual property for changes. To do that, you can configure an individual property with a *concurrency token*. However, it doesn’t usually make sense to check individual properties with an MVC app that uses a disconnected environment. That’s because it can be hard to track the original values of the checked properties, especially if there are a lot of them.

## How to configure a rowversion property with attributes

```
public class Book {  
    public int BookId { get; set; }  
    public string Title { get; set; }  
    public double Price { get; set; }  
  
    [Timestamp]  
    public byte[] RowVersion { get; set; }  
}
```

## How to configure a rowversion property with the Fluent API

```
protected override void OnModelCreating(ModelBuilder modelBuilder)  
{  
    modelBuilder.Entity<Book>()  
        .Property(b => b.RowVersion)  
        .IsRowVersion();  
}
```

## How to simulate a concurrency conflict

```
var book = context.Books.Find(1);           // get a book from the database  
book.Price = 14.99;                         // change price in memory  
  
context.Database.ExecuteSqlRaw(             // change price in the database  
    "UPDATE dbo.Books SET Price = Price + 1 WHERE BookId = 1");  
  
context.SaveChanges();
```

## Description

- A *concurrency conflict* is when data is modified after it's retrieved for editing or deletion.
- A *rowversion property* lets you check all the properties in an entity for conflicts and must be an array of bytes.
- The DbContext class has a Database property whose ExecuteSqlRaw() method can be used to simulate concurrency conflicts.
- A *concurrency token* lets you check an individual property for conflicts.

---

Figure 12-20 How to check for concurrency conflicts

## How handle a concurrency exception

If you've enabled optimistic concurrency and an update or delete operation fails because of a concurrency problem, EF throws a concurrency exception. To handle this exception, you call the `SaveChanges()` method within the try block of a try-catch statement, and you include a catch block that catches the exception.

Figure 12-21 shows a controller action method named `Edit()` that handles a concurrency exception. Here, within the catch block, this code uses the `Entries` property of the concurrency exception to get the current values of the entity being updated. The `Entries` property is a collection of `DbEntityEntry` objects, but it only has one entry here. As a result, the code uses the LINQ `Single()` method to retrieve the entry. If you want to be more cautious, you can use the `SingleOrDefault()` method and check for null.

The `DbEntityEntry` class has properties named `CurrentValues` and `OriginalValues`, but those properties typically aren't useful in a disconnected scenario. Instead, this example uses the `GetDatabaseValues()` method to retrieve the current values of the entity from the database.

After calling the `GetDatabaseValues()` method, the code checks whether the return value is null. If it is, the entity has been deleted and is no longer in the database. As a result, the code adds a class-level error to the `ModelState` object notifying the user that the row no longer exists.

If the object returned by the `GetDatabaseValues()` method is not null, the entity has been updated. As a result, the code adds a class-level error to the `ModelState` object notifying the user that the data has changed. After that, it casts the object holding the current database values to a `Book` object. Then, it compares each database value to the value passed in for editing. Whenever it finds a mismatch, it adds a property-level error to the `ModelState` object that notifies the user of the new value in the database.

In either case, this code displays the view with the values that were originally passed to this method. That way, any data that was entered is not lost, and your users can decide what they want to do next.

The last example shows a snippet of the `Edit` view that posts to this controller. Here, the form uses hidden fields to store both the book's primary key value and its `rowversion` value. As a result, these values are posted to the action method of the controller.

## A controller action method that handles a concurrency conflict

```
[HttpPost]
public IActionResult Edit(Book book)
{
    if (ModelState.IsValid) {
        context.Books.Update(book);

        // simulate the row being changed after retrieval and before save
        // to test a concurrency conflict

        try {
            context.SaveChanges();
            return RedirectToAction("Index");
        }
        catch (DbUpdateConcurrencyException ex) {
            var entry = ex.Entries.Single();
            var dbValues = entry.GetDatabaseValues();
            if (dbValues == null) {
                ModelState.AddModelError("", "Unable to save - "
                    + "this book was deleted by another user.");
            }
            else {
                ModelState.AddModelError("", "Unable to save - "
                    + "this book was modified by another user. "
                    + "The current database values are displayed "
                    + "below. Please edit as needed and click Save, "
                    + "or click Cancel.");

                var dbBook = (Book)dbValues.ToObject();

                if (dbBook.Title != book.Title)
                    ModelState.AddModelError("Title",
                        $"Current db value: {dbBook.Title}");

                // check rest of properties for equality
            }
            return View(book);
        }
    }
    else {
        return View(book);
    }
}
```

## Some of the code in the Edit view

```
@* both primary key and row version value needed for edit *@
<input type="hidden" asp-for="BookId" />
<input type="hidden" asp-for="RowVersion" />
<button type="submit" class="btn">Submit</button>
```

## Description

- The DbUpdateConcurrencyException is thrown when there's a concurrency conflict.
- The DbUpdateConcurrencyException has an Entries property that provides a way to get the new database values for the row that's being saved.

---

Figure 12-21 How to handle a concurrency exception

## How to encapsulate your EF code

So far, the EF Core code you've seen has been in the action methods of controllers. Usually, though, this code should be in its own data layer. This makes your code easier to test. In addition, it allows you to store your data access code in a separate project. Finally, it makes it possible to change from EF to another ORM framework without affecting the rest of your code.

### How to code a data access class

To encapsulate your data access code, it's a good practice to add extension methods to the `IQueryable` interface. An *extension method* is similar to a regular method except that it's defined outside the data type that it's used with. The first example in figure 12-22 shows how to code an extension method.

In this example, a generic extension method named `PageBy<T>()` is added to the `IQueryable<T>` interface, as indicated by the first parameter for this method. In addition, this method accepts the number and size values needed for paging. Then, it adds the `Skip()` and `Take()` methods to the query so the requested pages will be retrieved, and it returns the `IQueryable` object. This allows your extension method to be chained, if needed, with other LINQ methods. Note that without this extension method, you'd have to code the `Skip()` and `Take()` methods in multiple places to provide for paging. As a result, it's a better practice to use an extension method.

In the data layer, a method can return `IQueryable` or `IEnumerable` objects. The advantage of returning an `IQueryable` object is that, because it contains a query and not data, a calling method can modify it before executing it. The disadvantage of returning an `IQueryable` object is that it mixes data access code with other layers of the app, which reduces the benefits of having a data layer. On the other hand, if a method returns an `IEnumerable` object, the data access code can be kept in the data layer.

The second example presents a class named `Data` that encapsulates two data access methods. Here, both methods return an `IEnumerable` object. To start, the `GetPageOfBooks()` method uses the `PageBy()` extension method to get a page of `Book` objects. Then, it executes the query and returns an `IEnumerable` object.

For some data access methods, you may want to make the method flexible by accepting LINQ expressions as arguments. To do that, you use the `Expression` class of the `System.Linq.Expressions` namespace to represent a lambda expression. For example, the `GetSortedFilteredBooks()` method accepts a lambda expression for a `where` parameter and a lambda expression for an `orderby` parameter. Within the method, the code uses these lambdas as the arguments for the `Where()` and `OrderBy()` methods.

This figure shows two ways to call this data access method. The first way is shorter, but the second way uses named parameters to make the code easier to read and understand. Either way, this code returns all books with a price that's less than 10, and it sorts those books by title.

### A class that adds an extension method to the `IQueryable<T>` interface

```
public static class QueryExtensions
{
    public static IQueryable<T> PageBy<T>(this IQueryable<T> query,
        int pageNumber, int pageSize)
    {
        return query
            .Skip((pageNumber - 1) * pageSize)
            .Take(pageSize);
    }
}
```

### A data access class with a method that accepts LINQ expressions

```
using System.Linq.Expressions;

public class Data
{
    private BookstoreContext context { get; set; }
    public Data(BookstoreContext ctx) => context = ctx;

    public IEnumerable<Book> GetPageOfBooks(int pageNumber, int pageSize)
    {
        return context.Books.PageBy(pageNumber, pageSize).ToList();
    }

    public IEnumerable<Book> GetSortedFilteredBooks(
        Expression<Func<Book, bool>> where,
        Expression<Func<Book, Object>> orderby)
    {
        return context.Books
            .Where(where)
            .OrderBy(orderby)
            .ToList();
    }
}
```

### Code that creates a data access object

```
var data = new Data(context);
```

### Code that passes a lambda expression to a data access method

```
var books = data.GetSortedFilteredBooks(
    b => b.Price < 10,
    b => b.Title);
```

### Code that uses named arguments to make the code easier to understand

```
var books = data.GetSortedFilteredBooks(
    where: b => b.Price < 10,
    orderby: b => b.Title);
```

### Description

- It's a good practice to encapsulate the code that works with data by coding a class in the data layer. To do that, you can add extension methods to the `IQueryable` interface and code methods that accept LINQ expressions as arguments.

---

Figure 12-22 How to code a data access class

## How to use a generic query options class

In the previous figure, you learned how to code a data access class that provides two methods for querying: one for retrieving a page of data and one for sorting and filtering data. However, these methods might cause logistical problems. For instance, what if you don't need to sort the data? What if you don't need to filter the data? Where does paging fit?

One way to avoid these problems is to code a generic class for query options like the one presented in the first example of figure 12-23. Here, the `QueryOptions` class begins by defining public lambda expression properties for sorting and filtering and public int properties for paging. Then, it defines a private array for storing strings that indicate any related entities the query should include. The write-only `Includes` property accepts a comma-separated string, removes any spaces, splits the string at the commas, and stores the resulting string array in the private field. The method named `GetIncludes()` is used to return the value of the private field. Or, if the private field is null, it returns an empty string array.

After defining the properties that get and set data, the `QueryOptions` class defines three read-only Boolean properties. These properties indicate whether the filtering, sorting, and paging properties have values.

The second example shows the `Data` class from the last figure after it has been updated to use the `QueryOptions` class. Here, the name of the method is changed to `GetBooks()`. That's because the method is more general now that it accepts a `QueryOptions` object as its only argument.

Within the `GetBooks()` method, the code starts with a query that returns all the books in the table. Then, it uses the properties of the `QueryOptions` class to refine that query. First, the code loops through the strings returned by the `GetIncludes()` method and passes them to the `Include()` method of the query. Next, the code uses the `HasWhere`, `HasOrderBy`, and `HasPaging` properties to check whether the arguments for filtering, sorting, and paging have been set. If so, it passes the appropriate query option argument to the appropriate query method. Finally, the code calls the `ToList()` method to execute the query and return the resulting `IEnumerable` object.

The third example calls the `GetBooks()` method and passes it a `QueryOptions` object. This example returns all books because it doesn't specify the query options for filtering and paging. However, it includes related author and genre data and sorts the results by book title.

A `QueryOptions` class makes your data layer more flexible and easier to use. For example, code that calls the `GetBooks()` method only needs to include values for the `QueryOptions` properties it needs. In addition, if you later need to add another query property, such as one for sorting in descending order, you can update the `GetBooks()` method to use it without having to change any existing code that calls that method.

## A generic query options class

```
using System.Linq.Expressions;

public class QueryOptions<T>
{
    // public properties for sorting, filtering, and paging
    public Expression<Func<T, Object>> OrderBy { get; set; }
    public Expression<Func<T, bool>> Where { get; set; }
    public int PageNumber { get; set; }
    public int PageSize { get; set; }

    // public write-only property for includes private string array
    private string[] includes;
    public string Includes {
        set => includes = value.Replace(" ", "").Split(',');
    }
    // public method returns includes array
    public string[] GetIncludes() => includes ?? new string[0];

    // read-only properties
    public bool HasWhere => Where != null;
    public bool HasOrderBy => OrderBy != null;
    public bool HasPaging => PageNumber > 0 && PageSize > 0;
}
```

## A data access class with a method that uses the options class

```
public class Data
{
    private BookstoreContext context { get; set; }
    public Data(BookstoreContext ctx) => context = ctx;

    public IEnumerable<Book> GetBooks(QueryOptions<Book> options)
    {
        IQueryable<Book> query = context.Books;
        foreach (string include in options.GetIncludes()) {
            query = query.Include(include);
        }
        if (options.HasWhere)
            query = query.Where(options.Where);
        if (options.HasOrderBy)
            query = query.OrderBy(options.OrderBy);
        if (options.HasPaging)
            query = query.PageBy(options.PageNumber, options.PageSize);
        return query.ToList();
    }
}
```

## Code that uses the method

```
var books = data.GetBooks(new QueryOptions<Book> {
    IncludeString = "BookAuthors.Author, Genre",
    OrderBy = b => b.Title
});
```

## Description

- You can use a class for query options to make your data layer more flexible.

---

Figure 12-23 How to use a generic query options class

## How to use the repository pattern

One popular way to implement a data layer is to use the *repository pattern*. This pattern encapsulates data code within a data access layer and also uses interfaces to provide a layer of abstraction. One benefit of this pattern is that it makes it easier to automate testing as shown in chapter 14.

The first example in figure 12-24 presents a simple generic interface named IRepository that you can create to work with a repository. This interface has six methods. The first two query data, the next three modify data, and the last one saves changes to the data.

When using the repository pattern, you should use one repository per entity. For example, a Bookstore app shouldn't use one big Bookstore repository. Instead, it should use a Book repository, an Author repository, a Genre repository, and so on.

One way to do that is to have each repository implement the IRepository interface. For example, you could have a BookRepository class, an AuthorRepository class, and so on. However, there are some problems with this approach. First, if you later change your interface, say to add a Count() method, you'll need to update all your repositories. Second, you'll have to implement all the methods of the interface for every repository, even if you're not going to use all of them for an entity. Third, this will lead to code duplication among repositories.

A better approach is to implement a generic Repository class like the one in the second example. This class uses a generic DbSet object. Then, its constructor calls the Set<T>() method of the context object to get a DbSet object for the specified type. Next, it uses that DbSet object to implement all the methods of the IRepository interface.

You can use this generic repository class to create various entity collections. For example, this code creates an Author repository:

```
var data = new Repository<Author>(ctx);
```

Often a generic Repository class like this is all you need for basic *CRUD* (*Create, Read, Update, Delete*) operations for each entity.

The third example shows some code from a controller that uses the generic Repository class. Here, the Author controller begins by defining a private property named data of the Repository<Author> type. Then, the constructor initializes this private property. Next, the Index() action method uses the repository to get a list of Author objects sorted by the author's first name.

If you need more specialized operations than the generic Repository class provides, you can inherit that class and override one or more of its methods. For example, if you need to insert a new author into the Author table but you don't want to add a new row to the AuthorBio table if all properties in the AuthorBio entity are null, you can code an AuthorRepository class that inherits the generic Repository class. Then, you can override its Insert() method to create an Insert() method that works the way you want.

## The generic IRepository interface

```
public interface IRepository<T> where T : class {
    IEnumerable<T> List(QueryOptions<T> options);
    T Get(int id);
    void Insert(T entity);
    void Update(T entity);
    void Delete(T entity);
    void Save();
}
```

## A generic Repository class that implements IRepository

```
public class Repository<T> : IRepository<T> where T : class {
    protected BookstoreContext context { get; set; }
    private DbSet<T> dbset { get; set; }

    public Repository(BookstoreContext ctx) {
        context = ctx;
        dbset = context.Set<T>();
    }

    public virtual IEnumerable<T> List(QueryOptions<T> options) {
        IQueryable<T> query = dbset;
        foreach (string include in options.GetIncludes()) {
            query = query.Include(include);
        }
        if (options.HasWhere)
            query = query.Where(options.Where);
        if (options.HasOrderBy)
            query = query.OrderBy(options.OrderBy);
        if (options.HasPaging)
            query = query.PageBy(options.PageNumber, options.PageSize);
        return query.ToList();
    }

    public virtual T Get(int id) => dbset.Find(id);
    public virtual void Insert(T entity) => dbset.Add(entity);
    public virtual void Update(T entity) => dbset.Update(entity);
    public virtual void Delete(T entity) => dbset.Remove(entity);
    public virtual void Save() => context.SaveChanges();
}
```

## A controller that uses the generic Repository class

```
public class AuthorController : Controller {
    private Repository<Author> data { get; set; }

    public AuthorController(BookstoreContext ctx) =>
        data = new Repository<Author>(ctx);

    public ViewResult Index() {
        var authors = data.List(new QueryOptions<Author> {
            OrderBy = a => a.FirstName
        });
        return View(authors);
    }
    ...
}
```

---

Figure 12-24 How to use the repository pattern

## How to use the unit of work pattern

Although some programmers think that including a Save() method in each repository is OK, others think that a repository should never have a Save() method. Either way, if you need to coordinate between multiple repositories, including a Save() method in each repository doesn't work. In those cases, you can combine the repository pattern with the unit of work pattern.

The *unit of work* pattern adds a central class that has repository objects as properties. Then, the central class passes each repository its context object, so they all share the same DB context. Finally, the central class includes a Save() method that calls the SaveChanges() method of the DB context. That way, when you call the Save() method of the central class, EF executes all changes in all repositories against the database. Then, if one change fails, they all fail.

The first code example in figure 12-25 shows a unit of work class for a Bookstore app. This class implements an interface that specifies all of its methods. This isn't required, but it's generally considered a good practice since it makes it easier to automate testing.

The repository properties are read-only properties that check whether the private backing field is null. If so, the property creates a new repository object and stores it in its private field. Then, the property returns the repository object. Due to space considerations, this figure only shows the repository property for Book objects, but you can use the same technique to create repository properties for Author, BookAuthor, and Genre objects.

The second code example shows an action method in a controller that uses this unit of work class. Here, the Book controller declares a private property named data and initializes this property to an instance of the unit of work class.

Within the controller class, the Edit() action method accepts an argument for a view model for the Book view. This method begins by checking whether the model state is valid. If so, it calls methods from the unit of work class to delete existing authors from the book specified by the view model and to add the new authors that it gets from the view model. Next, it calls the Update() method from the Books repository to update the book. Finally, it calls the Save() method from the unit of work class.

Note that this code begins by using the BookAuthors repository to delete the current authors and add the new authors. Then, it uses the Books repository to update the book, which automatically updates the BookAuthors repository. Finally, it uses the DbContext to attempt to save the changes. Because this code uses the unit of work class, all of these operations succeed or fail together. By contrast, if each repository had its own DbContext and Save() method, you could have a situation where some of the operations succeed and others fail.

If this code looks familiar, that's because EF Core uses the unit of work/repository pattern. The DbContext object is the unit of work class, and the DbSet properties are the repositories.

## A unit of work class with four repositories

```
public class BookstoreUnitOfWork : IBookstoreUnitOfWork
{
    private BookstoreContext context { get; set; }
    public BookstoreUnitOfWork(BookstoreContext ctx) => context = ctx;

    private Repository<Book> bookData;
    public Repository<Book> Books {
        get {
            if (bookData == null)
                bookData = new Repository<Book>(context);
            return bookData;
        }
    }

    // properties for Authors, BookAuthors, and Genres repositories go here

    public void DeleteCurrentBookAuthors(Book book) {
        var currentAuthors = BookAuthors.List(new QueryOptions<BookAuthor> {
            Where = ba => ba.BookId == book.BookId
        });
        foreach (BookAuthor ba in currentAuthors) {
            BookAuthors.Delete(ba); // deletes from BookAuthors repository
        }
    }

    public void AddNewBookAuthors(Book book, int[] authorids) {
        foreach (int id in authorids) {
            BookAuthor ba =
                new BookAuthor { BookId = book.BookId, AuthorId = id };
            BookAuthors.Insert(ba); // adds to BookAuthors repository
        }
    }

    public void Save() => context.SaveChanges();
}
```

## A controller that uses the unit of work class to update a book

```
public class BookController : Controller
{
    private BookstoreUnitOfWork data { get; set; }
    public BookController(BookstoreContext ctx) =>
        data = new BookstoreUnitOfWork(ctx);

    [HttpPost]
    public IActionResult Edit(BookViewModel vm) {
        if (ModelState.IsValid) {
            data.DeleteCurrentBookAuthors(vm.Book);
            data.AddNewBookAuthors(vm.Book, vm.SelectedAuthors);
            data.Books.Update(vm.Book);
            data.Save();
        }
        ...
    }
    ...
}
```

---

Figure 12-25 How to use the unit of work pattern

## Perspective

This chapter showed how to use EF Core to work with data in a database, including how to encapsulate your EF code in a data layer. This presents the most important classes of the data layer that are used by the Bookstore website presented in the next chapter. As a result, studying that app is a great way to get started with EF Core.

## Terms

entity class	linking table
database (DB) context class	composite primary key
Entity Framework (EF) Core	cascading delete
object relational mapping (ORM)	LINQ to Entities
Code First development	projection
Database First development	data transfer object (DTO)
Fluent API	disconnected scenario
primary key (PK)	connected scenario
foreign key (FK)	concurrency
one-to-many relationship	concurrency conflict
many-to-many relationship	optimistic concurrency
one-to-one relationship	rowversion property
navigation property	concurrency token
table splitting	extension method
join entity	repository pattern
linking entity	CRUD (Create, Read, Update, Delete)
join table	unit of work pattern

## Summary

- *Entity classes* define the data structure for an app and map to the tables in a relational database.
- A *database (DB) context class* inherits the DbContext class and includes one DbSet property for each entity class.
- *Entity Framework (EF) Core* is an *object relational mapping (ORM)* framework that allows you to map your entity classes to the tables of a database.
- With *Code First* development, you code your entity classes and a context class first. Then, you use EF to create a database from these classes.
- With *Database First* development, you create your database tables first. Then, you generate the context and entity classes from your database.
- You can configure your database by convention, by adding data attributes to your entity classes, or by writing code that uses the *Fluent API*.

- Relationships are defined with a primary key and a foreign key. The *primary key (PK)* uniquely identifies an entity, and the *foreign key (FK)* identifies related entities.
- A *one-to-many relationship* relates one entity to many entities. A *many-to-many relationship* relates many entities to many other entities. And a *one-to-one relationship* relates one entity to one other entity.
- A *navigation property* provides a way to navigate to a related entity from the primary entity.
- *Table splitting* allows you to use two entities to represent the data that's stored in a single table.
- A many-to-many relationship uses an intermediate entity called a *join entity* or *linking entity* that maps to an intermediate table in the database called a *join table* or *linking table*.
- A primary key that consists of more than one property is called a *composite primary key*.
- A *cascading delete* causes all dependent child rows to be automatically deleted when a parent row is deleted.
- You can use *LINQ to Entities* to query the DbSet properties of a context class.
- A *projection* allows you to retrieve a subset of the properties of an entity.
- A *data transfer object (DTO)* is a simple object that's designed to store data that's being transferred from one place to another.
- With the *disconnected scenario*, an action method gets an entity from an HTTP POST request. Then, the method must call the Update() method to update the state of the entity before calling the SaveChanges() method.
- With the *connected scenario*, an action method retrieves an entity from the database so the context can track the state of the entity. Then, you don't need to call the Update() method. Instead, you just change the entity and call the SaveChanges() method.
- *Concurrency* allows two or more users to work with a database at the same time.
- A *concurrency conflict* occurs when data is modified by one user after it's retrieved for editing or deletion by another user.
- *Optimistic concurrency* checks whether a row has been changed since it was retrieved. If so, it refuses the update or deletion and throws an exception.
- A *rowversion property* lets you check all properties of an entity for concurrency conflicts.
- A *concurrency token* lets you check an individual property of an entity for concurrency conflicts.

- An *extension method* is similar to a regular method except that it's defined outside the data type that it's used with. You can use extension methods to extend the IQueryable interface so you can encapsulate your data access code.
- With the *repository pattern*, you create one repository per entity, and each repository has its own DB context.
- You can use a generic repository class that implements a repository interface to provide for basic *CRUD (Create, Read, Update, Delete)* operations for each entity in a DB context.
- The *unit of work pattern* extends the repository pattern by adding a central class that contains the repositories and shares a single DB context among them.

## Exercise 12-1 Review the data layer of an app and improve it

In this exercise, you'll review domain model and configuration code as well as the migration file EF Code First creates from that code. Then, you'll modify this code to improve it.

### Review the domain model and configuration files

1. Open the Ch12Ex1ClassSchedule web app in the ex\_starts directory.
2. Open the Models/DomainModels folder and review the code in the three class files it contains.
3. Open the Models/Configuration folder and review the code in the three configuration files it contains.

### Review the migration file

4. Open the Migrations folder and review the code in the Initial migration file.
5. Note how the migration file determines the primary and foreign keys based on how the properties in the domain model are coded (configuration by convention).

### Create the database

6. Open the Package Manager Console and enter the Update-Database command to run the migration and create the database.
7. Take a moment to review the SQL code in the console after the command runs.
8. Run the app and review the data it displays for classes and teachers.

### Test the cascading delete behavior of the Teacher object

9. Run the app and navigate to the Show All Teachers page. Click the Add Teacher link and add your name as a teacher.
10. Navigate to the Show All Classes page. Click the Add Class link and add a new class with yourself as the teacher.

11. Review the updated class list to make sure it displays the class you just added.
12. Navigate back to the Show All Teachers page and delete your name.
13. Navigate back to the Show All Classes page, and note that the class you previously added has also been deleted.

### Restrict the cascading delete behavior of the Teacher object

14. In the Models/DomainModels folder, open the configuration file for the domain model named Class.
15. Add code to change the delete behavior for the Teacher foreign key to Restrict.
16. Open the Package Manager Console and use the Add-Migration command to create a new migration file. When you do that, use a descriptive name for the migration file.
17. Review the code in the migration file that's generated.
18. Enter the Update-Database command to apply the migration to the database.
19. Repeat steps 9 through 13 to test the cascading delete behavior. When you attempt to delete your name from the teacher list, you should get an error message.
20. Navigate to the Show All Classes page, find the class you added, click on Edit, and change the teacher to another teacher.
21. Navigate to the Show All Teachers page and delete your name from the teacher list. This time, the app should let you.

### Update the app to use the unit of work pattern

22. Open the Controllers folder and review the three controller class files. Note that two of them initialize more than one Repository class in the constructor.
23. Open the Models/DataLayer folder, add an interface named IClassScheduleUnitOfWork and a class named ClassScheduleUnitOfWork.
24. Adjust the namespaces of the interface and class to match the namespaces of the other classes in the DataLayer folder.
25. Code the interface to have read-only properties for a Class, Teacher, and Day repository and a Save() method that returns void.
26. Code the class to implement the interface, have a private ClassSchedule-Context object that it gets in its constructor, initialize and return Repository objects in its properties, and call the context object's SaveChanges() method in the Save() method.
27. Update the Home and Class controllers to use the unit of work class.
28. Run the app and make sure it still works the same.

### Change how the list of classes is ordered

29. Open the Home controller and review the code in its Index() action method. Note that the classes are ordered by day on first load and ordered by time when filtering by day.

30. Run the app. When it displays all classes, note that it doesn't display the classes for Monday in ascending order by time. Then, click on the filter link for Monday and note that it *does* display the classes in ascending order by time.
31. In the Models/DataLayer folder, open the QueryOptions class. Note that this version is shorter than the QueryOptions class presented in the chapter. That's because it doesn't provide for paging.
32. Add a new property named ThenOrderBy that works like the OrderBy property.
33. Add a new read-only property named HasThenOrderBy that works like the HasOrderBy property.
34. Open the Repository class and find the List() method. Update the code that handles ordering so it uses the ThenOrderBy property like this:

```
if (options.HasOrderBy) {
    if (options.HasThenOrderBy) {
        query = query.OrderBy(options.OrderBy).ThenBy(options.ThenOrderBy);
    }
    else {
        query = query.OrderBy(options.OrderBy);
    }
}
```

35. In the Home controller, update the Index() action method to use the new property to sort by time as well as day on first load.
36. Repeat step 30. The class times should be in ascending order on both pages.

### Add an overload for the Get() method in the Repository class

37. Open the Class controller and review the private helper method named GetClass(). Note that it uses the List() method of the Repository<Class> class to get an IEnumerable<Class> with one Class item. Then, it uses the LINQ FirstOrDefault() method to retrieve that item.
38. In the Models folder, open the IRepository<T> interface and add a second Get() method that accepts a QueryOptions<T> object.
39. Open the Repository<T> class and implement the new Get() method. To do that, you can copy the code from the repository's List() method that builds a query, leaving out the code for ordering the list. Then, you can return the single object by calling the FirstOrDefault() method instead of the List() method.
40. In the Class controller, update the GetClass() method to use the new Get() method.
41. Run the app and test it. It should work the same as it did before.

# The Bookstore website

The Bookstore website presented in this chapter is a realistic website that uses the skills presented so far in this book. Due to space considerations, this chapter doesn't present every line of code for this website. However, the complete code for this website is available in the download for this book. So, as you read this chapter, you can run the website to see how it works, and you can review its code. That's a great way to learn!

<b>The user interface and folder structure .....</b>	<b>496</b>
The end user pages.....	496
The admin user pages .....	498
The folders and files.....	500
<b>Some general-purpose code.....</b>	<b>504</b>
Extension methods for sessions and strings .....	504
The generic QueryOptions class .....	506
The generic Repository class.....	508
<b>The paging and sorting of the Author Catalog.....</b>	<b>510</b>
The custom route and the GridDTO class.....	510
The RouteDictionary class .....	512
The GridBuilder class.....	514
The Author/List view model and the Author controller.....	516
The Author/List view.....	518
<b>The paging, sorting, and filtering of the Book Catalog ..</b>	<b>520</b>
The custom route and the BooksGridDTO class.....	520
The FilterPrefix class and the updated RouteDictionary class .....	522
The BooksGridBuilder class .....	524
The BookQueryOptions and BookstoreUnitOfWork classes .....	526
The Book/List view model and the Book controller.....	528
The Book/List view .....	530
<b>The Cart page .....</b>	<b>532</b>
Extension methods for cookies.....	532
The user interface .....	534
The model classes .....	534
The Cart controller .....	542
The Cart/Index view .....	544
<b>The book search of the Admin pages .....</b>	<b>546</b>
The user interface .....	546
The SearchData and SearchViewModel classes.....	548
The Search() action methods of the Book controller.....	550
The Delete() action method of the Genre controller.....	552
<b>Perspective .....</b>	<b>554</b>

## The user interface and folder structure

This chapter begins by presenting some of the pages of the Bookstore website. Then, it shows how the folders and files for this website are structured. This should give you a good idea of how this website works.

### The end user pages

Like all the pages of the Bookstore website, the pages shown in figure 13-1 use a layout that begins with a Bootstrap navbar followed by a Bootstrap jumbotron that displays an image that includes a heading and logo. In addition, this app has custom CSS that styles the jumbotron.

Each link in the Bootstrap navbar also has a Font Awesome icon. For example, the Home link has an icon of a house, and the Books link has an icon of an open book. Also, the app marks the link and icon for the current page as active. The navbar also contains a Bootstrap badge that indicates how many books are currently in the user's shopping cart. If there are no books in the cart, the app doesn't display this badge.

The Home page uses EF Core to access a database like the one described in chapter 12 that contains data about books, authors, and genres. It selects a random book and displays it as the staff pick. The title of the book is a link to the details page for the book. This link uses the default route to add the book ID and a title slug to the route. For instance, the URL for the link shown here is:

```
/book/details/28/pride-and-prejudice-and-zombies
```

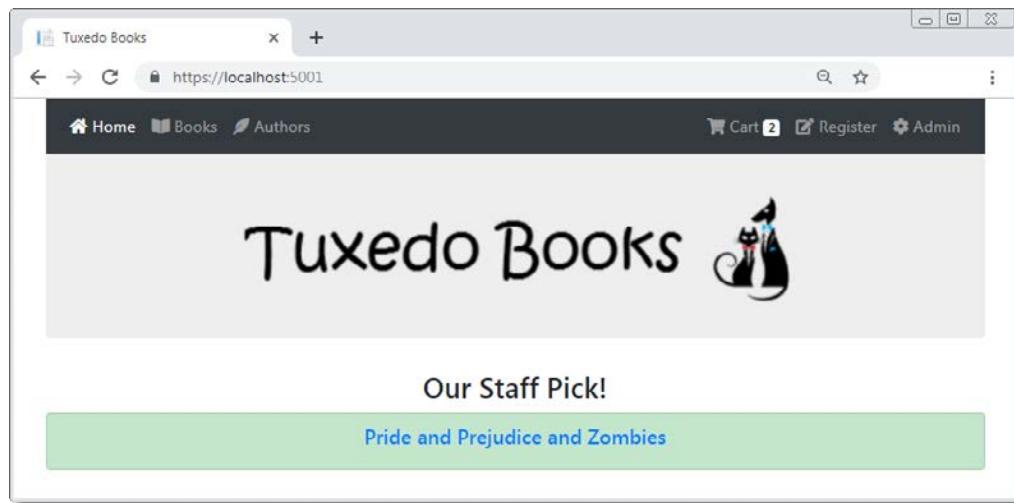
The Book Catalog page uses EF Core to query the database for a list of books, and it displays those books in a table styled with Bootstrap. For each book, it displays the title, author(s), genre, and price, as well as a button to add the book to the shopping cart. The title and authors displayed for each book are also links that take the user to a details page. The details page for a book includes the same information that's shown on the Book Catalog page. The details page for an author includes a list of books for the author.

The Book Catalog page doesn't display all the books in the database. Rather, it displays a page of no more than four books at a time. It also lets the user know how many of these pages exist by presenting links the user can click to navigate through the pages and view the books.

In addition to paging, the Book Catalog page allows the user to sort the list of books by title, genre, or price in ascending or descending order. To do that, it makes the headers for these columns links that the user can click to change the sort order.

Finally, the Book Catalog page provides three drop-down lists that the user can use to limit the books in the list to a specific author, genre, or price. The user can combine these filters, too. For example, a user can filter books by a specific author in a specific genre and within a specific price range.

## The Home page



## The Book Catalog page

A screenshot of a web browser window showing the Tuxedo Books Book Catalog page. The title bar says "Tuxedo Books | Book Catalog" and the address bar shows "localhost:5001/book/list". The page features a large "Tuxedo Books" logo with a cartoon cat icon. Below the logo is a section titled "Book Catalog". It includes a search bar and filter options for Author, Genre, and Price. A table lists books with columns for Title, Author(s), Genre, and Price, each with an "Add To Cart" button. At the bottom, there are navigation links for pages 1 through 8.

Title	Author(s)	Genre	Price	
1776	David McCullough	History	\$18.00	Add To Cart
1984	George Orwell	Science Fiction	\$5.50	Add To Cart
And Then There Were None	Agatha Christie	Mystery	\$4.50	Add To Cart
Band of Brothers	Stephen E. Ambrose	History	\$11.50	Add To Cart

Figure 13-1 The end user and admin user pages (part 1)

The Author Catalog page, shown in part 2 of figure 13-1, uses EF Core to query the database for a list of authors. Then, it displays those authors in a table styled with Bootstrap. For each author, the table displays the author's first name, last name, and the book or books written by that author.

The first and last names for the author are also links that use the default route to add the author ID and a name slug to the route to take the user to a details page. For instance, the URL for the link for both the first name and last name of the first author shown here is:

```
/author/details/7/agatha-christie
```

Similarly, each book title is a link that takes the user to a details page for the book as described in the last figure.

Like the Book Catalog page, the Author Catalog page provides links for paging through the data that's retrieved from the database. In addition, it provides for sorting by first or last name. However, this page doesn't provide for sorting by the Book(s) column.

## The admin user pages

---

The Bookstore website has an Admin area that you can use to add, edit, or delete a book, author, or genre in the database. Right now, this Admin area can be accessed by any user. However, you'll learn how to limit access to authorized users in section 3 of this book.

The Admin page, also shown in part 2 of figure 13-1, has three tabs with the titles Manage Books, Manage Authors, and Manage Genres. These tabs are created by using Bootstrap classes, and they allow the user to manage the books, authors, and genres used by the Bookstore website.

When the Admin page loads, it displays the Manage Books tab by default. This page contains a link you can click to add a book and a search box you can use to locate an existing book so you can edit or delete it. When you search for a book, you can search by book title, author, or genre. This search functionality is described in more detail later in this chapter.

## The Author Catalog page

The screenshot shows a web browser window for "Tuxedo Books | Author Catalog" at [localhost:5001/author/list](http://localhost:5001/author/list). The page features a header with "Home", "Books", "Authors", "Cart 2", "Register", and "Admin". Below the header is a logo of a black cat. The main content area is titled "Author Catalog" and contains a table listing authors and their books:

First Name	Last Name	Book(s)
Agatha	Christie	<a href="#">And Then There Were None</a> <a href="#">Murder on the Orient Express</a>
Aldous	Huxley	<a href="#">Brave New World</a>
Augusten	Burroughs	<a href="#">Running With Scissors</a>
Daphne	Du Maurier	<a href="#">Rebecca</a>

Pagination links at the bottom range from 1 to 7.

## The Manage Books tab of the Admin page

The screenshot shows a web browser window for "Tuxedo Books | Manage Books" at <https://localhost:5001/admin>. The page features a header with "Home", "Books", "Authors", "Cart 2", "Register", and "Admin". Below the header is a logo of a black cat. The main content area is titled "Manage Books" and includes tabs for "Manage Books", "Manage Authors", and "Manage Genres". Under the "Manage Books" tab, there is a "Add a book" button and a search bar with fields for "Search Term:", "Search By:" (with radio buttons for Title, Author, and Genre), and a "Find" button.

Figure 13-1 The end user and admin user pages (part 2)

## The folders and files

---

Figure 13-2 shows the Controllers and Models folders of the Bookstore website. The Controllers folder contains a controller file for the first four navigation links: Home, Books, Authors, and Cart. The controllers for the Admin link are stored in their own area, as you'll see in part 2 of this figure.

The Models folder, by contrast, contains a much larger assortment of files, as well as several subfolders. This makes sense because it's considered a best practice for an MVC app to have a "fat" model and "skinny" controllers. The idea is that the controllers should only be in charge of getting data from the model and passing it to the views. However, the model should contain the data itself as well as the business logic for working with that data.

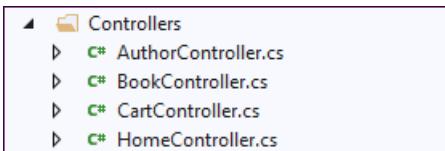
Of course, a "fat" model might lead to its own problems if you try to store all of the code in just one or two class files. That's why it's considered a good practice to create a model like the one shown here, with the business logic and data files broken down into several smaller files. It's also common, in larger apps, for different parts of the model to be stored in separate projects, though that's beyond the scope of this book. Still, the same concept applies. Namely, the bulk of your code should be stored in your model, and you should organize this model in a way that makes it easy to understand and maintain.

In this figure, the Models folder has several subfolders. The DataLayer folder contains the DB context class, the repository and unit of work classes and interfaces, the QueryOptions classes, the seed classes, and the DomainModels folder contains the entity classes that EF Core uses to create the Bookstore database. If you want, you could keep the entity model classes in the DataLayer folder, since they're related to the other classes in that folder. However, storing these classes in a separate folder makes it easier to find the class that you're looking for.

The ExtensionMethods folder holds class files that add extension methods to the String class, to a custom list of CartItem objects, and to the session and cookie objects. The Grid folder holds classes that the Bookstore website uses to create the grids of data on the Author Catalog and Book Catalog pages.

The ViewModels folder contains classes that hold data needed by views. In this folder, the Nav class doesn't have the conventional ViewModel suffix. That's because it isn't really a view model. Rather, it's a static class that the layouts use to mark the navigation link for the current page as active. In other words, it's more of a utility class. Since it's used with views, though, it makes sense to store it with the view models.

## The Controllers folder



## The Models folder and its subfolders

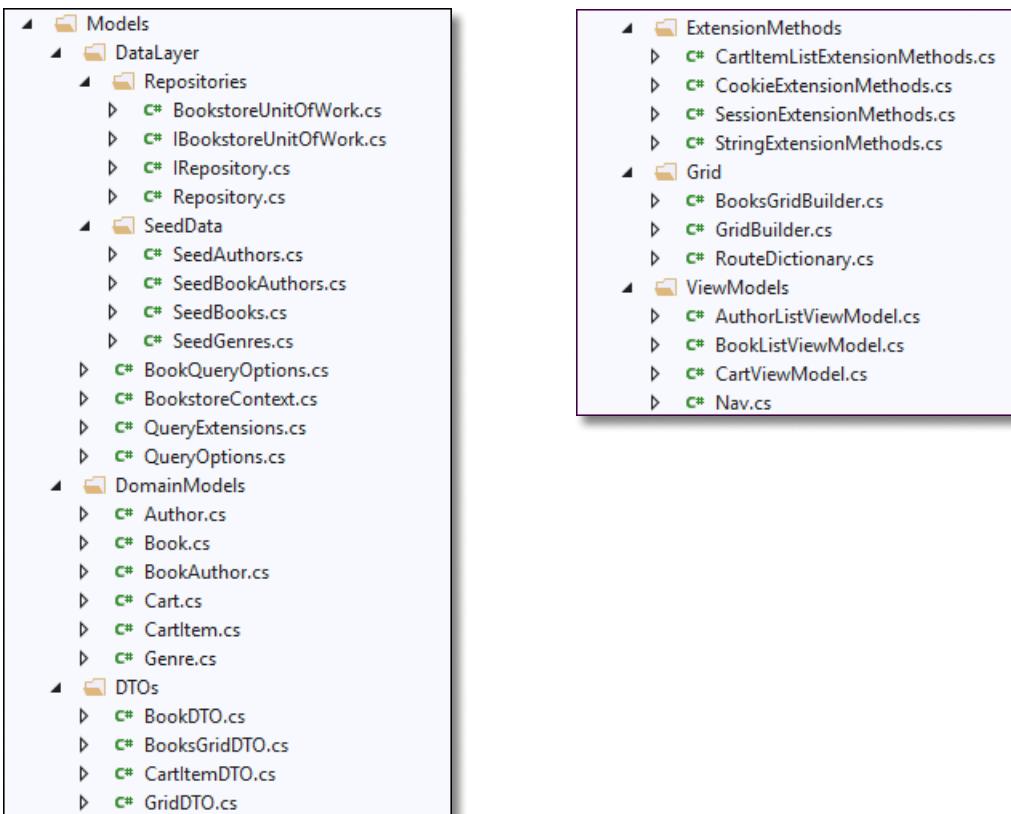


Figure 13-2 The folders and files (part 1)

The DTOs folder contains data transfer object (DTO) classes used to transfer data. Although view models are also used to transfer data, they're used specifically with views. By contrast, a DTO is a more general-purpose object that can be used to transfer data to other locations. For example, the Bookstore website uses DTOs to store data in session state or in cookies.

Part 2 of figure 13-2 shows the Views and Admin folders of the Bookstore website. The Views folder contains subfolders that correspond to the controllers for the end user pages: Home, Book, Author, and Cart. In addition, it contains the standard Shared folder that contains the layout for the views, as well as the \_ViewImports.cshtml and \_ViewStart.cshtml files.

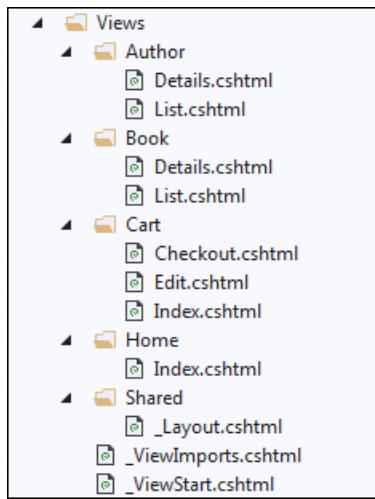
The Home and Cart folders each contain a default Index view. In addition, the Cart folder contains an Edit view to edit a cart item and a Checkout view. The Author and Book folders, by contrast, don't have a default Index view. That's because the Book and Author controllers have Index() action methods that redirect to the List view in each folder. That way, the Author Catalog and Book Catalog pages have URL segments of author/list and book/list, respectively. Since these URLs describe the pages more accurately, they're more user friendly. The Book and Author folders also each contain a Details view that displays details about an individual book or author.

The Areas folder contains the Admin folder of the Bookstore website. This folder contains the models, views, and controllers used by the Admin page. As shown in part 2 of figure 13-1, the Admin page has three tabs: Manage Books, Manage Authors, and Manage Genres. As a result, the Controllers folder has Book, Author, and Genre controllers that correspond to those tabs. Similarly, the Views folder has Book, Author, and Genre subfolders that correspond to those tabs.

The Models folder doesn't contain many files because the Admin page does most of its work using the classes contained in the main Models folder. However, the Models folder does contain some classes that are specific to the Admin page. In particular, this folder contains two view model classes that pass data to the Book and SearchResults views, a SearchData class used to work with TempData, and an Operation class with static methods that determine if an action being taken is an add, edit, or delete action.

The Models folder also contains a class that provides for remote validation, and the Controllers folder contains a Validation controller that works with this class. Since these files work like the ones described in chapter 11, this chapter doesn't show how they work.

## The Views folder



## The Admin area folder and its subfolders

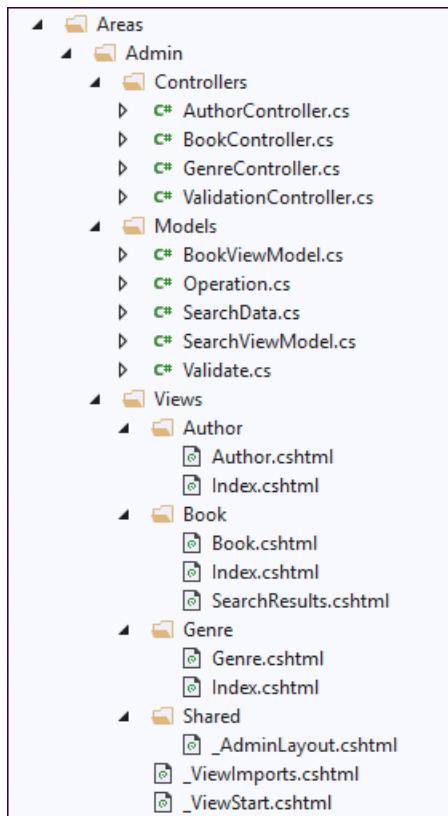


Figure 13-2 The folders and files (part 2)

## Some general-purpose code

---

In the next few topics, you'll review some of the general-purpose code that the Bookstore website uses. This code consists of extension methods for strings and sessions as well as versions of the generic `QueryOptions` and `Repository` classes that provide more functionality than the classes presented in chapter 12.

### Extension methods for sessions and strings

---

The first example in figure 13-3 adds two extension methods to the `ISession` interface. The first, `SetObject<T>()`, serializes an object of type `T` to a JSON string and then stores that string in session state. The second, `GetObject<T>()`, deserializes a JSON string stored in session state back to an object of type `T`. These methods allow you to store and retrieve complex types in session state.

The second example adds several extension methods to the `String` class. The first, `Slug()`, creates a dash-separated string. To do that, this method begins by removing all punctuation in the string except for the dash character (-). That way, dashes that are a part of a book's or author's name aren't removed from the string. Then, it replaces spaces with dashes and converts the characters to lowercase.

The `EqualsNoCase()` method uses the string's `ToLower()` method to perform a case-insensitive comparison. Unfortunately, this method doesn't work with EF Core queries. For those, you must use a standard equality operator and the `ToLower()` method if you want the comparison to be case-insensitive.

The `ToInt()` method uses the static `TryParse()` method of the `int` type to convert a string to an `int`. If the string can't be converted, this method returns the default value of an `int`, which is zero.

The `Capitalize()` method makes the first character of the string uppercase and all the remaining characters lowercase. To do this, it uses the string's `Substring()` method.

## Extension methods for the `ISession` interface

```
using Microsoft.AspNetCore.Http;
using Newtonsoft.Json;
...
public static class SessionExtensions
{
    public static void SetObject<T>(this ISession session, string key,
    T value) =>
        session.SetString(key, JsonConvert.SerializeObject(value));

    public static T GetObject<T>(this ISession session, string key) {
        var value = session.GetString(key);
        return value == null ? default(T) :
            JsonConvert.DeserializeObject<T>(value);
    }
}
```

## Extension methods for the `String` class

```
public static class StringExtensions
{
    public static string Slug(this string s) {
        var sb = new StringBuilder();
        foreach (char c in s) {
            if (!char.IsPunctuation(c) || c == '-') {
                sb.Append(c);
            }
        }
        return sb.ToString().Replace(' ', '-').ToLower();
    }

    public static bool EqualsNoCase(this string s, string tocompare) =>
        s?.ToLower() == tocompare?.ToLower();

    public static intToInt(this string s) {
        int.TryParse(s, out int id);
        return id;
    }

    public static string Capitalize(this string s) =>
        s?.Substring(0, 1)?.ToUpper() + s?.Substring(1).ToLower();
}
```

## Description

- These extension methods allow you to store complex objects in session state and to perform tasks with strings like creating slugs, comparing strings in a case-insensitive way, casting a string value to an int, and capitalizing strings.

---

Figure 13-3 Extension methods for sessions and strings

## The generic QueryOptions class

---

Chapter 12 presented a QueryOptions class that you can use to pass things such as include and order by expressions to a repository class. Now, figure 13-4 presents a version of a QueryOptions class that adds some additional features.

First, it adds an OrderByDirection property of the string type that a repository class can use with the OrderBy property to sort items in ascending or descending order. This property has a default value of “asc”, which means that if its value isn’t explicitly set, the query sorts the items in ascending order.

Second, it adds a new WhereClauses property of the WhereClauses<T> type. The bottom of this figure shows that this WhereClauses<T> class inherits a class that’s a list of where expression objects of the T type. In other words, the WhereClauses property contains a list of where expressions. This allows you to filter by more than one where expression.

The WhereClauses<T> class doesn’t add any functionality to the list it inherits. Rather, it’s used as an alias to make the code for working with where expressions simpler. For instance, if you want to declare a list of where expressions when you create a QueryOptions object, you can code it like this:

```
var options = new QueryOptions<Book>
{
    Include = "BookAuthors.Author, Genre",
    WhereClauses = new WhereClauses<Book> {
        { b => b.GenreId == "novel" },
        { b => b.Price < 10 }
    }
};
```

That’s easier to get right and to read than attempting to directly initialize an object of this type:

```
List<Expression<Func<Book, bool>>>
```

Third, the Where property in this class is write-only. The setter of this property initializes the WhereClauses property if it’s null. Then, the Add() method of the WhereClauses list is called to add the where expression value of the Where property to the WhereClauses list. This way, you can add one where expression to the list at a time rather than having to declare them all at once or work directly with the methods of the WhereClauses<T> object.

## The generic QueryOptions class

```
public class QueryOptions<T>
{
    public Expression<Func<T, Object>> OrderBy { get; set; }
    public string OrderByDirection { get; set; } = "asc"; // default
    public int PageNumber { get; set; }
    public int PageSize { get; set; }

    private string[] includes;
    public string Includes {
        set => includes = value.Replace(" ", "").Split(',');
    }
    public string[] GetIncludes() => includes ?? new string[0];

    public WhereClauses<T> WhereClauses { get; set; }
    public Expression<Func<T, bool>> Where {
        set {
            if (WhereClauses == null)
            {
                WhereClauses = new WhereClauses<T>();
            }
            WhereClauses.Add(value);
        }
    }

    public bool HasWhere => WhereClauses != null;
    public bool HasOrderBy => OrderBy != null;
    public bool HasPaging => PageNumber > 0 && PageSize > 0;
}

public class WhereClauses<T> : List<Expression<Func<T, bool>>> {}
```

### Description

- This version of the generic QueryOptions class adds an OrderByDirection property and a WhereClauses property. In addition, it makes the Where property write-only.
- The OrderByDirection property allows you to sort in ascending or descending order. The default value for this property is “asc” for ascending.
- The Where property adds the where expression it receives to the WhereClauses property, which is a list of where expressions.
- The type of the WhereClauses property is the generic WhereClauses class, which inherits a list of where clause expressions. This simple class functions as an alias and makes the code for working with where expressions cleaner.

---

Figure 13-4 The generic QueryOptions class

## The generic Repository class

---

Chapter 12 showed how to build a generic Repository class that you can use to encapsulate code that interacts with the database. Now, figure 13-5 presents a version of a Repository class that's similar to the one you saw in that chapter. However, this Repository class adds several features.

First, it adds a private count variable of the nullable int type. Then, it adds a public read-only Count variable of the int type that returns the value of the private count variable. If that value is null, however, it returns the value returned by the Count() method of the private DbSet<T> property. This Count property determines the number of paging links a view should provide.

Second, it adds a third overload for the Get() method that has a string parameter. This overload, like the one that has an int parameter, calls the Find() method of the DbSet<T> class and passes it the argument it receives.

Third, it has a private BuildQuery() method that accepts a QueryOptions object and uses it to build a query expression. This private method is used by both the public List() method and the public Get() method. In fact, the only difference between those two methods is that the List() method calls the ToList() method to execute the query and return a list of entities, and the Get() method calls the FirstOrDefault() method to execute the query and return a single entity.

Fourth, the query expression built by the BuildQuery() method has more options for sorting and filtering than the one presented in chapter 12. Now, the if statement that checks for a where clause loops through the where expressions in the WhereClauses list and adds them to the query expression. That way, it can filter by more than one where expression.

When the where expression loop completes, the code makes a separate query to the database to get the number of items that the expression being built will return. It's important to understand that that this line of code actually executes and retrieves a value from the database. However, it doesn't execute the query that's being built. Rather, it executes a different query, but with the same where expressions. Note that if there aren't any where expressions in the WhereClauses property, this code doesn't run and the private count variable remains null.

Finally, the if statement that checks for an order by clause contains another if statement that checks the OrderByDirection property. If it's "asc", the code uses the OrderBy() method to build a query that sorts in ascending order. Otherwise, it uses the OrderByDescending() method to build a query that sorts in descending order.

In chapter 12, the Repository class implemented the IRepository interface. That's true of this Repository class as well. While not shown here, the IRepository interface that this class implements also has the read-only Count property and the third overload of the Get() method.

## The generic Repository class

```
public class Repository<T> : IRepository<T> where T : class
{
    protected BookstoreContext context { get; set; }
    private DbSet<T> dbset { get; set; }
    public Repository(BookstoreContext ctx) {
        context = ctx;
        dbset = context.Set<T>();
    }
    private int? count;
    public int Count => count ?? dbset.Count();

    public virtual IEnumerable<T> List(QueryOptions<T> options) {
        IQueryable<T> query = BuildQuery(options);
        return query.ToList();
    }
    public virtual T Get(int id) => dbset.Find(id);
    public virtual T Get(string id) => dbset.Find(id);
    public virtual T Get(QueryOptions<T> options) {
        IQueryable<T> query = BuildQuery(options);
        return query.FirstOrDefault();
    }
    public virtual void Insert(T entity) => dbset.Add(entity);
    public virtual void Update(T entity) => dbset.Update(entity);
    public virtual void Delete(T entity) => dbset.Remove(entity);
    public virtual void Save() => context.SaveChanges();

    private IQueryable<T> BuildQuery(QueryOptions<T> options)
    {
        IQueryable<T> query = dbset;
        foreach (string include in options.GetIncludes()) {
            query = query.Include(include);
        }
        if (options.HasWhere) {
            foreach (var clause in options.WhereClauses) {
                query = query.Where(clause);
            }
            count = query.Count(); // get filtered count
        }
        if (options.HasOrderBy) {
            if (options.OrderByDirection == "asc")
                query = query.OrderBy(options.OrderBy);
            else
                query = query.OrderByDescending(options.OrderBy);
        }
        if (options.HasPaging)
            query = query.PageBy(options.PageNumber, options.PageSize);
        return query;
    }
}
```

### Description

- The private BuildQuery() method builds the query expression. If there's filtering, it executes a separate query to get the count of items that the query expression will return.

---

Figure 13-5 The generic Repository class

## The paging and sorting of the Author Catalog

With ASP.NET Web Forms, a table that provides for paging and sorting is available out of the box via the GridView server control. With ASP.NET Core MVC, however, you must “roll your own” grid for paging and sorting. The next few figures present the code that the Bookstore website uses for the grid that provides for paging and sorting the data that’s displayed on the Author Catalog page.

### The custom route and the GridDTO class

The first two columns of the table in the Author Catalog page has column headers that are links you can click to sort the data for the page. When you first click a link, the sort order is ascending. On subsequent clicks of the same link, the order toggles between descending and ascending. Below the table, you can click the paging links to view all the authors.

Figure 13-6 shows the Author Catalog page after a user has navigated to the third page of authors and sorted the authors by last name in descending order. The URL that produces this page is shown below the screen.

For this to work, you need to add a custom route to the Configure() method in the Startup.cs file as shown below the URL. This custom route specifies static segments of “page”, “size”, and “sort” that make the route more user friendly. In addition, it specifies four route parameters named pagenumber, pagesize, sortfield, and sortdirection. Since this route is more specific than the default route shown in figure 13-1, it must be coded before that route.

The Bookstore website uses a simple class named GridDTO to work with the values of this custom route. Notice that the names of the properties of this DTO class match the names of the route parameters. This is important for MVC model binding to work. However, the capitalization of these properties doesn’t need to match the route parameters.

The GridDTO class provides default values for three of its four properties. As a result, if no values are provided in the URL route, the app uses these defaults. Note that there’s no default provided for the SortField property. That’s because the PageNumber, PageSize, and SortDirection values are general purpose. That is, these default values make sense in any app. The name of the field to sort by, on the other hand, is app specific. As a result, the code for the Bookstore website needs to specify the default value for this property.

The Bookstore website doesn’t provide a way for the user to change the PageSize value from its default value of 4. However, that’s just because of space considerations. It would be easy to allow users to change the number of items that display in a grid, and many real-world web apps provide this functionality.

### Page 3 of the Author Catalog sorted by last name in descending order

First Name	Last Name	Book(s)
Frank	Herbert	Dune
Dashiell	Hammett	The Maltese Falcon
Seth	Grahame-Smith	Pride and Prejudice and Zombies
Roxane	Gay	Hunger

1 2 3 4 5 6 7

### The URL for the above page

```
https://localhost:5001/author/list/page/3/size/4/sort/lastname/desc
```

### The custom route in the Startup.cs file

```
endpoints.MapControllerRoute(
    name: "",
    pattern: "{controller}/{action}/page/{pagenumber}/size/{pagesize}/
        sort/{sortfield}/{sortdirection}");
```

### The GridDTO class with some default paging and sorting values

```
public class GridDTO {
    public int PageNumber { get; set; } = 1;
    public int PageSize { get; set; } = 4;
    public string SortField { get; set; }
    public string SortDirection { get; set; } = "asc";
}
```

### Description

- The Author Catalog page displays four authors at a time and provides links styled as buttons that allow the user to page through the authors.
- In the grid, the first two column headers are links that allow the user to sort the authors by first name or last name in ascending or descending order.
- The Bookstore website adds a custom route for paging and sorting and uses a *data transfer object (DTO)* class to bind those route values to an action method.

Figure 13-6 The custom route for paging and sorting and the GridDTO class

## The RouteDictionary class

---

Figure 13-7 presents the RouteDictionary class. This class inherits a string-string dictionary and adds several properties and methods. The Bookstore website uses this class to store and retrieve route parameters.

This may seem confusing at first. After all, isn't the GridDTO object used to store route parameter values? Yes, but only to transfer data from the route to a controller via model binding. In that case, couldn't you use a RouteDictionary for model binding and skip the GridDTO altogether? Again, yes, but it would need to initialize with default values, which would make it more complex and therefore harder to understand and maintain. In this case, it's simpler to have one class (GridDTO) responsible for model binding the route values, and another class (RouteDictionary) responsible for working with those values in a string-string dictionary.

Chapter 8 showed that you can use a string-string dictionary to build links that have the correct route segment values. Because the RouteDictionary class is a string-string dictionary, then, it's easy to use it to build links in views as shown later in this chapter.

The RouteDictionary class has four properties that get or set a specified route segment in the string-string dictionary. Note that these properties use the names of the GridDTO properties as keys. This allows the `<asp-all-route-data>` tag helper shown in figure 13-10 to work correctly.

The property getters pass the key value to the private `Get()` method. This method starts by checking if the key is in the dictionary. If so, it returns the associated string value. Otherwise, it returns null. This prevents null reference exceptions.

The RouteDictionary class has two public methods named `SetSortAndDirection()` and `Clone()`. The views of an app can use these methods to build the paging and sorting links as shown later.

The `SetSortAndDirection()` method determines whether a sorting link should order items in ascending or descending order. To do that, it compares the value of the `fieldName` argument to the `SortField` value of the current route. If they are the same, the user has clicked on the header for the column that's already being used for the sort. In that case, the method toggles between ascending and descending order. If the values are different, the user has clicked on a different grid column. In that case, sorting always starts in ascending order.

The `Clone()` method returns a copy of the current RouteDictionary object. This is important because a view that has paging and sorting uses a RouteDictionary object to both keep track of the current route values and to create the route values for each paging and sorting link. If you tried to do both with the same dictionary, you'd lose the current route values as you changed the values to create a link. With the `Clone()` method, though, you can use a copy for building links and not affect the current route data.

## The RouteDictionary class

```
public class RouteDictionary : Dictionary<string, string>
{
    public int PageNumber {
        get => Get(nameof(GridDTO.PageNumber)).ToInt();
        set => this[nameof(GridDTO.PageNumber)] = value.ToString();
    }

    public int PageSize {
        get => Get(nameof(GridDTO.PageSize)).ToInt();
        set => this[nameof(GridDTO.PageSize)] = value.ToString();
    }

    public string SortField {
        get => Get(nameof(GridDTO.SortField));
        set => this[nameof(GridDTO.SortField)] = value;
    }

    public string SortDirection {
        get => Get(nameof(GridDTO.SortDirection));
        set => this[nameof(GridDTO.SortDirection)] = value;
    }

    private string Get(string key) => Keys.Contains(key) ? this[key] : null;

    public void SetSortAndDirection(string fieldName, RouteDictionary current)
    {
        this[nameof(GridDTO.SortField)] = fieldName;

        if (current.SortField.EqualsNoCase(fieldName) &&
            current.SortDirection == "asc")
            this[nameof(GridDTO.SortDirection)] = "desc";
        else
            this[nameof(GridDTO.SortDirection)] = "asc";
    }

    public RouteDictionary Clone()
    {
        var clone = new RouteDictionary();
        foreach (var key in Keys) {
            clone.Add(key, this[key]);
        }
        return clone;
    }
}
```

## Description

- The RouteDictionary class inherits a string-string dictionary. It stores and retrieves the route segment values for paging and sorting.
- It uses the property names of the GridDTO class as dictionary keys. This makes sure that the route values in the dictionary correspond to the correct route parameter name.
- The SetSortAndDirection() method makes sure the direction of a sorting link is correct.
- The Clone() method returns a copy of the current RouteDictionary object.

---

Figure 13-7 The RouteDictionary class

## The GridBuilder class

---

Figure 13-8 presents the GridBuilder class. This class stores route parameters in and retrieves them from session state. To do that, it uses the RouteDictionary and GridDTO classes, as well as the session extension methods. This is a good example of how having several simple classes with limited responsibilities allows you to compose classes that do a lot of work without a lot of code. Imagine how long the GridBuilder class would be if it tried to do all that work itself!

The GridBuilder class starts by defining a private constant named RouteKey that it uses to set and get the current route from session state. Then, the class has a protected RouteDictionary property named routes and a private ISession property named session. The RouteDictionary property is protected because it needs to be accessible to any derived classes. The ISession property, on the other hand, doesn't need to be accessible, so it can be private.

The GridBuilder class provides two overloaded constructors. The first accepts a session object and stores it in the private session property. Then, it calls the GetObject() extension method from the session property and passes it the RouteKey constant to load the route dictionary from session state. Or, if there is no route dictionary in session state, this code initializes the property with a new route dictionary. You can use this constructor when you just need to retrieve the current route data from session state.

The second constructor also accepts a session object and stores it in the private property named session. In addition, it accepts a GridDTO object and a string that contains a default sort field. Then, it initializes the routes property with a new route dictionary. This clears any previous items in the dictionary.

After that, the constructor assigns the values in the GridDTO object to the route dictionary. If there's no SortField value in the GridDTO, it uses the value in the defaultSortField argument. Finally, it calls the SaveRouteSegments() method to save the route dictionary to session state. You can use this constructor when you need to save new route segment values to session state.

The public SaveRouteSegments() method uses the SetObject() extension method of the session object and the RouteKey constant to store the RouteDictionary property in session state. The public GetTotalPages() method uses the count argument it receives and the value in the PageSize property of the RouteDictionary property to calculate the total number of pages. And the read-only CurrentRoute property returns the value of the RouteDictionary property.

## The GridBuilder class

```
public class GridBuilder
{
    private const string RouteKey = "currentroute";

    protected RouteDictionary routes { get; set; }
    private ISession session { get; set; }

    // this constructor used when just need to get route data from the session
    public GridBuilder(ISession sess)
    {
        session = sess;
        routes = session.GetObject<RouteDictionary>(RouteKey) ??
            new RouteDictionary();
    }

    // this constructor used when need to store paging-sorting route segments
    public GridBuilder(ISession sess, GridDTO values, string defaultSortField)
    {
        session = sess;

        routes = new RouteDictionary(); // clear previous route segment values
        routes.PageNumber = values.PageNumber;
        routes.PageSize = values.PageSize;
        routes.SortField = values.SortField ?? defaultSortField;
        routes.SortDirection = values.SortDirection;

        SaveRouteSegments();
    }

    public void SaveRouteSegments() =>
        session.SetObject<RouteDictionary>(RouteKey, routes);

    public int GetTotalPages(int count)
    {
        int size = routes.PageSize;
        return (count + size - 1) / size;
    }

    public RouteDictionary CurrentRoute => routes;
}
```

## Description

- The GridBuilder class works with a RouteDictionary object that's stored in session state.
- The first constructor accepts an ISession argument and uses it to retrieve a RouteDictionary object from session state. If there's no RouteDictionary object in session state, it creates a new one.
- The second constructor accepts an ISession argument, a GridDTO object, and a default sort field. It uses these arguments to create and initialize a new RouteDictionary object and save the RouteDictionary object in session state for later use.
- The SaveRouteSegments() method saves the route values to session state, and the GetTotalPages() method gets the total pages based on the page size route value.
- The CurrentRoute property is a read-only property that returns a RouteDictionary object with the route segment values of the current route.

---

Figure 13-8 The GridBuilder class

## The Author/List view model and the Author controller

---

Figure 13-9 presents the `AuthorListViewModel` class. This class uses its three properties to store the data a view needs to display a grid of authors with paging and sorting. The `Authors` property contains a collection of the `Author` objects to display. The `CurrentRoute` property contains a `RouteDictionary` object with the paging and sorting route segments values from the current URL. And the `TotalPages` property contains an `int` indicating how many paging links to display.

This figure also presents the `Author` controller. This controller has a constructor that creates an `Author` repository and assigns it to a private property named `data`. In addition, it has a default `Index()` action method that redirects to the `List()` action method.

The `List()` action method accepts a `GridDTO` object. Remember, the `GridDTO` class has properties that match the custom route for paging and sorting, and three of the four properties have default values. As a result, after model binding, this object contains the parameter values in the URL if they exist or the default values if the route parameter values don't exist.

The `List()` action method starts by setting the default value for the `SortField` property to the `FirstName` property. That's necessary because the `GridDTO` class can't define a default value for this property since the default sort field is app specific. Then, it creates a new `GridBuilder` object. To do that, it passes the session object, the `GridDTO` object, and the default sort field to the `GridBuilder` constructor. As you saw in the previous figure, this constructor creates a route dictionary, assigns it the values in the `GridDTO`, assigns the default sort field if necessary, and stores the route dictionary in session state.

After creating the `GridBuilder` object, the code creates a new `QueryOptions` object for the `Author` class. Then, it uses the `CurrentRoute` property of the `GridBuilder` object to set the query options. Next, the code assigns an expression to the `OrderBy` property. If the `SortField` property of the current route matches the value in the `defaultSort` variable, the expression sorts by the `FirstName` property. Otherwise, it sorts by the `LastName` property.

You could make an argument that the code that sets the `OrderBy` property is business logic that shouldn't be in the controller. In this case, though, the code is short so it's OK to store it here. However, if this code got more complex, you probably would want to move it to the model. In fact, that's how the code for the Book Catalog works.

Once the `QueryOptions` object is complete, the code initializes a view model object and loads it with data from the database and the `GridBuilder` object. Then, it passes the view model to the view.

The `Author` controller also has a `Details()` action method that accepts an ID. Although it isn't shown here in full, it works similarly to other `Details()` action methods presented in this book. In short, it uses the ID to query the database for an `Author` object that it passes to the view.

## The Author/List view model

```
public class AuthorListViewModel
{
    public IEnumerable<Author> Authors { get; set; }
    public RouteDictionary CurrentRoute { get; set; }
    public int TotalPages { get; set; }
}
```

## The Author controller

```
public class AuthorController : Controller
{
    private Repository<Author> data { get; set; }
    public AuthorController(BookstoreContext ctx) =>
        data = new Repository<Author>(ctx);

    public IActionResult Index() => RedirectToAction("List");

    public ViewResult List(GridDTO vals)
    {
        // get GridBuilder object, load route segment values, store in session
        string defaultSort = nameof(Author.FirstName);
        var builder = new GridBuilder(HttpContext.Session, vals, defaultSort);

        // create options for querying authors
        var options = new QueryOptions<Author> {
            Includes = "BookAuthors.Book",
            PageNumber = builder.CurrentRoute.PageNumber,
            PageSize = builder.CurrentRoute.PageSize,
            OrderByDirection = builder.CurrentRoute.SortDirection
        };
        // OrderBy depends on value of SortField route
        if (builder.CurrentRoute.SortField.EqualsNoCase(defaultSort))
            options.OrderBy = a => a.FirstName;
        else
            options.OrderBy = a => a.LastName;

        var vm = new AuthorListViewModel {
            Authors = data.List(options),
            CurrentRoute = builder.CurrentRoute,
            TotalPages = builder.GetTotalPages(data.Count)
        };
        return View(vm);
    }

    public IActionResult Details(int id) {...}
}
```

## Description

- The List() action method of the Author controller accepts a GridDTO object and passes it to the constructor of a GridBuilder object that stores its values in session state.
- The List() action method uses the paging and sorting values passed to the GridBuilder object to create query options and retrieve the specified authors from the database.
- The List() action method creates a view model with the author, route, and paging data the view needs to create an author grid with sorting and paging.

Figure 13-9 The Author/List view model and the Author controller

## The Author/List view

---

Figure 13-10 presents the Author/List view. This is a strongly-typed view whose model object is an instance of the AuthorListViewModel class.

The Razor code in the block at the top of the view creates two RouteDictionary variables named current and routes. The current variable holds the CurrentRoute property of the view model. As a result, it's a dictionary that contains the route segment values from the current URL. This variable functions as an alias, and it makes code that works with the CurrentRoute property shorter and easier to read.

The routes variable, by contrast, holds a *copy* of the CurrentRoute property. This allows the view to create URLs for the sorting and paging links without losing the values of the current URL. In other words, to do its work, this view needs a RouteDictionary object with values it can change, and a RouteDictionary object with values that don't change.

In the view, an HTML table styled with Bootstrap displays the author data. In the <thead> element, the column headers for the first two columns are links that the user can click to sort the author data by that field.

To create these sorting links, the view has a Razor code block before the HTML for each column header. Within each block, the code calls the SetSortAndDirection() method of the routes object and passes it the name of the current column header and the RouteDictionary object for the current URL. After the method completes, the routes dictionary holds the correct sort field and direction for the column header. Then, the header link assigns the routes object, which is a string-string dictionary, to the asp-all-route-data tag helper. This creates a URL for the link that has the correct routing segments.

In the <tbody> element, a Razor foreach statement loops through the Author collection of the view model and displays each author's first name, last name, and the books associated with them. This displays each author name and book title as a link to a details page with a value for the ID and slug route parameters. Remember, though, that the Details() action method of the Authors controller only accepts an ID parameter. As a result, the slug route segment isn't used by the controller. However, it makes the URL more user friendly. The Book Details page works the same way.

Below the HTML table, a Razor code block builds the paging links. This code starts by re-assigning a copy of the CurrentRoute property of the view model to the routes variable. This resets that variable to hold the segment values of the current URL. In other words, it clears the changes made while creating the sorting URLs. Then, it defines a for loop that starts with an index of 1 and runs until the index equals the value in the TotalPages property of the view model. Within this loop, the code uses the index value to set both the PageNumber property of the routes object and the text of the link. Finally, the code calls the static Nav.Active() method to check whether the paging link matches the current page. If so, this method returns "active", which is added to the list of classes in the class attribute. This is a Bootstrap class that marks the paging link as active.

## The Author/List view

```

@model AuthorListViewModel

@{
    ViewData["Title"] = " | Author Catalog";

    RouteDictionary current = Model.CurrentRoute;
    RouteDictionary routes = Model.CurrentRoute.Clone();
}
<h1>Author Catalog</h1>



| @if (routes.SetSortAndDirection(nameof(Author.FirstName), current))     <a href="#" asp-action="List" asp-all-route-data="@routes"         class="text-white">First Name</a> | @if (routes.SetSortAndDirection(nameof(Author.LastName), current))     <a href="#" asp-action="List" asp-all-route-data="@routes"         class="text-white">Last Name</a> | Books(s)                                                                                                                                                                   |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <a asp-action="Details" asp-route-id="@author.AuthorId" asp-route-slug="@author.FullName.Slug()" href="#">@author.FirstName</a>                                              | <a asp-action="Details" asp-route-id="@author.AuthorId" asp-route-slug="@author.FullName.Slug()" href="#">@author.LastName</a>                                             | @foreach (var ba in author.BookAuthors) { <p> <a asp-controller="Book" asp-route-id="@ba.BookId" asp-route-slug="@ba.Book.Title.Slug()" href="#">@ba.Book.Title</a> </p> } |



@{
    routes = Model.CurrentRoute.Clone(); // Reset to current route values *@
    for (int i = 1; i <= Model.TotalPages; i++) {
        routes.PageNumber = i;
        <a href="#" asp-action="List" asp-all-route-data="@routes"
            class="btn btn-primary @Nav.Active(i, current.PageNumber)">@i</a>
    }
}

```

---

Figure 13-10 The Author/List view

## The paging, sorting, and filtering of the Book Catalog

Like the Author Catalog page, the Book Catalog page provides for paging and sorting. In addition, it provides for filtering the data in the grid by one or more parameters. The next few figures present the code the Bookstore website uses to page, sort, and filter a grid of books.

### The custom route and the BooksGridDTO class

Figure 13-11 shows the Book Catalog page after a user has filtered by genre to display only novels, navigated to the second page of those books, and sorted them by price in ascending order. The URL fragment below the screen shows the route segments that produce this page.

For this to work, you need to add another custom route to the `Configure()` method in the `Startup.cs` file as shown below the URL. This custom route adds a static segment of “filter” and three parameters (author, genre, and price) to the custom route for paging and sorting presented in figure 13-6. Since this route is more specific than that custom route and the default route, it must be coded before both of them in the `Startup.cs` file.

The Bookstore website uses a simple class named `BooksGridDTO` to work with the filtering values of this custom route. However, this class inherits the `GridDTO` class that works with the paging and sorting values of the route. As before, the names of the properties of this `DTO` class match the names of the route parameters so the MVC model binding works.

The `BooksGridDTO` class provides the same default value for all three of its properties. So, if no values are provided in the URL route, the app uses this default. To do this, the class uses a constant of the string type. The code defines this constant as public so it can be used by other code.

This `DTO` class uses a default value of “all” for its three properties. However, the values in the route are “author-all”, “genre-all”, and “price-all”. That’s because the classes presented in the next figure add these prefixes to make the URL more user friendly.

So, why can’t you just add the filtering properties to the existing `GridDTO` class? Because, if you did that, the filter route segments and default values would show in the URL for every page. That is, an Author Catalog page URL would include “/filter/author-all/genre-all/price-all”, even though that page doesn’t do filtering.

Similarly, if you want a page to filter a grid by other criteria, you would need to create its own `DTO`. Or, if you only want to page through a grid without sorting or filtering, you’d need to create a `DTO` class with just the paging properties. In that case, you’d probably want to change the `GridDTO` to inherit that `DTO` class rather than duplicate the paging properties.

## Page 2 of the Book Catalog filtered by the novel genre and sorted by price in ascending order

The screenshot shows a web browser window for 'Tuxedo Books | Book Catalog'. The URL in the address bar is `localhost:5001/book/list/page/2/size/4/sort/price/asc/filter/author-all/genre-novel/price-all`. The page title is 'Book Catalog'. At the top, there are dropdown menus for 'Author' (set to 'All'), 'Genre' (set to 'Novel'), and 'Price' (set to 'All'). Below these are 'Filter' and 'Clear' buttons. The main content is a table grid displaying three books:

Title	Author(s)	Genre	Price	
<a href="#">Harry Potter and the Sorcerer's Stone</a>	JK Rowling	Novel	\$9.75	<a href="#">Add To Cart</a>
<a href="#">Go Tell it on the Mountain</a>	James Baldwin	Novel	\$10.25	<a href="#">Add To Cart</a>
<a href="#">Beloved</a>	Toni Morrison	Novel	\$10.99	<a href="#">Add To Cart</a>

Below the table are navigation links '1' and '2'.

### The route segments for the URL for the above page

```
/page/2/size/4/sort/price/asc/filter/author-all/genre-novel/price-all
```

### The custom route in the Startup.cs file

```
endpoints.MapControllerRoute(
    name: "",
    pattern: "{controller=Home}/{action=Index}/
        {page}/{pagenumber}/{size}/{pagesize}/{sort}/{sortfield}
        /{sortdirection}/{filter}/{author}/{genre}/{price}");
```

### The BooksGridDTO class with default filtering values

```
using Newtonsoft.Json;
...
public class BooksGridDTO : GridDTO {
    [JsonIgnore]
    public const string DefaultFilter = "all";

    public string Author { get; set; } = DefaultFilter;
    public string Genre { get; set; } = DefaultFilter;
    public string Price { get; set; } = DefaultFilter;
}
```

### Description

- The Book Catalog page is similar to the Author Catalog page but adds the ability to filter.

---

Figure 13-11 The custom route to page, sort, and filter and the BooksGridDTO class

## The FilterPrefix class and the updated RouteDictionary class

---

As described in the last figure, the route segments for filtering add a prefix to make the segments user friendly. However, the app only uses those prefixes for display to the user. As a result, it needs to add them to the URL, but remove them when working with the URL route values in code.

To facilitate this, the app uses the static class named FilterPrefix that's shown at the top of figure 13-12. This class defines three public constants that contain the prefix values. Then, the app uses this class to work with prefixes in both the RouteDictionary class shown in this figure and the BooksGridBuilder class shown in the next figure.

The RouteDictionary class shown in this figure updates the RouteDictionary class from figure 13-7 by adding properties and methods for filtering. The three filtering properties get and set values in the string-string dictionary. However, the getter properties have to do a little extra work to strip the prefix from the value. To do that, the getter for each property replaces the FilterPrefix constant with an empty string. That's necessary because the BooksGridBuilder class shown in the next figure stores that prefix as well as the parameter value for these filter properties.

The AuthorFilter property has to do even more work after that. Since the filter value for an author is a numeric author ID, the route parameter value contains a slug as well as a prefix, such as author-8-ta-nehisi-coates. Because of that, the getter for AuthorFilter also needs to remove the slug. To do that, it needs to find the index of the dash after the author ID. Since the prefix has already been removed, that dash is the first one in the remaining string. So, the code uses the IndexOf() method of the string to find it. Then, it uses the Substring() method of the string to get the ID, which is in the characters from the start of the string to the index value.

The ClearFilters() method resets the filter properties to the default. To do this, it uses the public DefaultFilter constant of the BooksGridDTO class for all three filter properties.

At this point, you may be wondering why this code updates the RouteDictionary class instead of creating a derived class that inherits it. The main reason is that adding the properties to the existing class doesn't cause any problems, like it would with the GridDTO. So, it's easier to just update the class. Second, it allows every view to work with the RouteDictionary class, which keeps things simple. If the code used a derived class, a view would need to know which type of RouteDictionary to work with.

Of course, there are also good reasons to have separate class files. It keeps each file shorter and easier to understand. In addition, it keeps the general-purpose properties and methods for paging and sorting separate from the app-specific properties and methods for filtering. To use separate class files without using inheritance, you could use partial classes for your RouteDictionary class. That way, you could store your paging, sorting, and filtering properties and methods in separate files, but a view can still just work with a RouteDictionary object.

## The static FilterPrefix class

```
public static class FilterPrefix
{
    public const string Genre = "genre-";
    public const string Price = "price-";
    public const string Author = "author-";
}
```

## The updated RouteDictionary class

```
public class RouteDictionary : Dictionary<string, string>
{
    // paging and sorting properties and methods here - see figure 13-7

    public string GenreFilter {
        get => Get(nameof(BooksGridDTO.Genre))?.Replace(
            FilterPrefix.Genre, "");
        set => this[nameof(BooksGridDTO.Genre)] = value;
    }

    public string PriceFilter {
        get => Get(nameof(BooksGridDTO.Price))?.Replace(
            FilterPrefix.Price, "");
        set => this[nameof(BooksGridDTO.Price)] = value;
    }

    public string AuthorFilter {
        get {
            string s = Get(nameof(BooksGridDTO.Author))?.Replace(
                FilterPrefix.Author, "");
            int index = s?.IndexOf('-') ?? -1;
            return (index == -1) ? s : s.Substring(0, index);
        }
        set => this[nameof(BooksGridDTO.Author)] = value;
    }

    public void ClearFilters() =>
        GenreFilter = PriceFilter = AuthorFilter = BooksGridDTO.DefaultFilter;
}
```

## Description

- This code updates the RouteDictionary class presented earlier to enable filtering.
- The filter properties are specific to the app.
- The static FilterPrefix class contains public constants that the app uses to add and remove user-friendly prefixes from the route parameters for filtering. These constants are used by the RouteDictionary and BooksGridBuilder classes.
- The setters store the route parameters for filtering in the dictionary, and the getters remove the prefixes when retrieving these values.
- The filter route value for an author includes a slug. As a result, after removing the prefix, the getter for the AuthorFilter property removes the slug.

---

Figure 13-12 The FilterPrefix class and the updated RouteDictionary class

## The BooksGridBuilder class

---

Figure 13-13 presents the BooksGridBuilder class. This class inherits the GridBuilder class and stores and retrieves route parameters from session state.

The BooksGridBuilder class has two overloaded constructors. The first accepts a session object and passes it to the base constructor. This initializes the protected RouteDictionary property with the values retrieved from session state or creates a new RouteDictionary object.

The second constructor accepts a session object, a BooksGridDTO object, and a default sort field value and passes them to the base constructor. Then, it stores the filter route values in the dictionary. First, though, it checks if the route values have prefixes yet by testing for the genre prefix. If it's not there, that means this is the initial load of the page with default values. In that case, the constructor adds the prefix to the default filter value before it stores it. That's how a default value of "all" becomes a route value of "genre-all", "author-all", or "price-all".

The BooksGridBuilder class has two public methods that work with filter values. The LoadFilterSegments() method stores filter values in the dictionary that it gets from the array posted by the filter drop-down lists. Since the values posted by the drop-down lists don't have prefixes, this method doesn't need to check before adding them. However, it does need to check to see if it needs to add a slug to the author filter value. To do that, it checks the author argument it receives. If that object has a value, it adds the slug.

The ClearFilterSegments() method calls the ClearFilters() method of the protected RouteDictionary property. Remember from the last figure that this method sets all the filter properties to the value of the public DefaultFilter constant of the BooksGridDTO class.

The BooksGridBuilder class has several app-specific Boolean flags that indicate the values the grid is filtering by and the field the grid is sorting by. The filter flags use the public DefaultFilter constant of the BooksGridDTO class for this.

So, why not just add code to the GridBuilder class instead of creating a BooksGridBuilder subclass? After all, that technique worked for the RouteDictionary class. Well, since the views don't work directly with the grid builders, there's no compelling reason to store all the code in one class. Instead, using inheritance allows you to keep the general-purpose code for paging and sorting in the GridBuilder class. Then, you can store the app-specific code for filtering and sorting in the BooksGridBuilder class.

## The BooksGridBuilder class

```
public class BooksGridBuilder : GridBuilder
{
    // this constructor gets route data from session state
    public BooksGridBuilder(ISession sess) : base(sess) { }

    // this constructor stores filtering route segments, as
    // well as paging and sorting segments stored by the base constructor
    public BooksGridBuilder(ISession sess, BooksGridDTO values,
        string defaultSortField) : base(sess, values, defaultSortField)
    {
        bool isInitial = values.Genre.IndexOf(FilterPrefix.Genre) == -1;
        routes.AuthorFilter = (isInitial) ?
            FilterPrefix.Author + values.Author : values.Author;
        routes.GenreFilter = (isInitial) ?
            FilterPrefix.Genre + values.Genre : values.Genre;
        routes.PriceFilter = (isInitial) ?
            FilterPrefix.Price + values.Price : values.Price;
    }

    public void LoadFilterSegments(string[] filter, Author author) {
        if (author == null) {
            routes.AuthorFilter = FilterPrefix.Author + filter[0];
        } else {
            routes.AuthorFilter = FilterPrefix.Author + filter[0]
                + "-" + author.FullName.Slug();
        }
        routes.GenreFilter = FilterPrefix.Genre + filter[1];
        routes.PriceFilter = FilterPrefix.Price + filter[2];
    }

    public void ClearFilterSegments() => routes.ClearFilters();

    // filter flags
    string default = BooksGridDTO.DefaultFilter;
    public bool IsFilterByAuthor => routes.AuthorFilter != default;
    public bool IsFilterByGenre => routes.GenreFilter != default;
    public bool IsFilterByPrice => routes.PriceFilter != default;

    // sort flags
    public bool IsSortByGenre =>
        routes.SortField.EqualsNoCase(nameof(Genre));
    public bool IsSortByPrice =>
        routes.SortField.EqualsNoCase(nameof(Book.Price));
}
}
```

### Description

- The BooksGridBuilder class inherits the GridBuilder class and adds functionality for filtering.
- The first constructor calls the base constructor that gets the route data. The second constructor calls the base constructor that gets the route data and stores the paging and sorting route segments. Then, it stores the route segments for filtering, adding the user-friendly filter prefixes as needed.

---

Figure 13-13 The BooksGridBuilder class

## The BookQueryOptions and BookstoreUnitOfWork classes

---

Figure 13-14 presents the BookQueryOptions class that inherits the generic QueryOptions class for Book objects. This class adds a single method named SortFilter() to the class it inherits.

The SortFilter() method accepts a BooksGridBuilder object and uses its Boolean flag and CurrentRoute properties to add the appropriate where and order by expressions to the query expression that's being built. This method starts by checking the filter flags of the BooksGridBuilder class and adding a where expression for any flag that returns true.

Because it works with a nested object, the where expression for the author filter is more involved than the other two. This code starts by converting the author filter value to an int value. Then, it passes a lambda expression to the LINQ Any() method of the BookAuthors property of the Book object. This lambda expression then queries for any nested Author object whose AuthorId value matches the author filter value.

The SortFilter() method adds the order by expression. To do this, it uses the sort flags of the BooksGridBuilder class. Unlike the if statements for filtering, this code doesn't check both sort flags. In other words, if the first sort flag is true, the else if block is skipped. That's because the Book Catalog page only sorts by one column at a time. In addition, this if statement includes an else block with a default expression that sorts the books by title.

This figure also reviews the BookstoreUnitOfWork class that was presented in the previous chapter. This class implements the properties and methods specified by the IBookstoreUnitOfWork interface. The properties named Books, Genres, BookAuthors, and Authors are for the Book, Genre, BookAuthor, and Author repositories. That way, the app can use the Genres and Authors properties to populate the drop-down lists for filtering. In addition, the app can use these repository properties to query the database for a list of books, a list of authors, or a single author. In short, the app uses this unit of work class when a controller needs to work with data from multiple tables, and it uses a repository class when a controller only needs to work with data from a single table.

## The BookQueryOptions class

```
public class BookQueryOptions : QueryOptions<Book>
{
    public void SortFilter(BooksGridBuilder builder)
    {
        if (builder.IsFilterByGenre) {
            Where = b => b.GenreId == builder.CurrentRoute.GenreFilter;
        }
        if (builder.IsFilterByPrice) {
            if (builder.CurrentRoute.PriceFilter == "under7")
                Where = b => b.Price < 7;
            else if (builder.CurrentRoute.PriceFilter == "7to14")
                Where = b => b.Price >= 7 && b.Price <= 14;
            else
                Where = b => b.Price > 14;
        }
        if (builder.IsFilterByAuthor) {
            int id = builder.CurrentRoute.AuthorFilter.ToInt();
            if (id > 0)
                Where = b => b.BookAuthors.Any(a => a.Author.AuthorId == id);
        }

        if (builder.IsSortByGenre) {
            OrderBy = b => b.Genre.Name;
        }
        else if (builder.IsSortByPrice) {
            OrderBy = b => b.Price;
        }
        else {
            OrderBy = b => b.Title;
        }
    }
}
```

## The BookstoreUnitOfWork class

```
public class BookstoreUnitOfWork : IBookstoreUnitOfWork
{
    private BookstoreContext context { get; set; }
    public BookstoreUnitOfWork(BookstoreContext ctx) => context = ctx;

    // properties for Books, Authors, BookAuthors, and Genres repositories

    // helper methods for deleting current authors from existing book
    // or loading book authors into a new book

    public void Save() => context.SaveChanges();
}
```

---

Figure 13-14 The BookQueryOptions and BookstoreUnitOfWork classes

## The Book/List view model and the Book controller

---

Figure 13-15 presents the BookListViewModel class. This view model is similar to the AuthorListViewModel class, with a collection of Book objects to display, a RouteDictionary object that holds the values for the current URL, and an int that stores the number of paging links to display.

In addition, this view model has properties that hold data for the drop-down lists for filtering. The Authors property is a collection of Author objects, and the Genres property is a collection of Genre objects. The Prices property, by contrast, is a string-string dictionary of price ranges. To keep things simple, these dictionary values are hard-coded.

Below the view model, this figure presents the List() and Filter() action methods of the Book controller. Although it isn't shown here, the constructor for the controller creates a BookstoreUnitOfWork object and stores it in a private property named data. That's why the data property can access the Books, Authors, and Genres repositories.

The List() action method accepts a BooksGridDTO object. After model binding, this object contains the route segments for paging, sorting, and filtering. If there aren't any, it contains the default values coded in the BooksGridDTO class and the base GridDTO class.

The List() action method starts by creating a new BooksGridBuilder object from the session object, the DTO object, and a default sort field. This creates and populates a new route dictionary and stores it in session state.

After creating the builder object, the code initializes a new BookQueryOptions object. As before, it uses the CurrentRoute property of the BooksGridBuilder object to set paging and sorting properties. Then, it passes the builder object to the SortFilter() method, which builds the rest of the query expression. Next, the code creates a view model object and loads it with data from the database and the builder before it passes that view model to the view.

The Filter() action method accepts a string array and a Boolean value. These parameters are bound to the filter drop-down lists and the Clear button in the view. Within the method, the code starts by updating the filter route segments based on the value of the Boolean clear parameter.

If the clear parameter is true, the code calls the ClearFilterSegments() method. Otherwise, the code uses the first filter in the array to get the selected author. If the filter value is "all", theToInt() method returns 0, and the GetAuthor() method returns null. Then, the code passes the filter string array and the author to the LoadFilterSegments() method. This adds the appropriate route segments for filtering to the current route dictionary. Either way, this code saves the updated route segments to session state.

Finally, this code creates a new BooksGridBuilder object by passing a session object to a constructor that retrieves the current route from session state. Then, that method redirects to the List() action method. To build the route segments, it passes the value of the CurrentRoute property, which is a string-string dictionary, to the RedirectToAction() method.

## The Book/List view model

```
public class BookListViewModel
{
    public IEnumerable<Book> Books { get; set; }
    public RouteDictionary CurrentRoute { get; set; }
    public int TotalPages { get; set; }

    // for filter drop-down data - one hard-coded
    public IEnumerable<Author> Authors { get; set; }
    public IEnumerable<Genre> Genres { get; set; }
    public Dictionary<string, string> Prices =>
        new Dictionary<string, string> {
            { "under7", "Under $7" },
            { "7to14", "$7 to $14" },
            { "over14", "Over $14" } };
}
```

## The List() and Filter() action methods of the Book controller

```
public ViewResult List(BooksGridDTO values) {
    var builder = new BooksGridBuilder(HttpContext.Session, values,
        defaultSortField: nameof(Book.Title));

    var options = new BookQueryOptions {
        Include = "BookAuthors.Author, Genre",
        OrderByDirection = builder.CurrentRoute.SortDirection,
        PageNumber = builder.CurrentRoute.PageNumber,
        PageSize = builder.CurrentRoute.PageSize
    };
    options.SortFilter(builder);

    var vm = new BookListViewModel {
        Books = data.Books.List(options),
        Authors = data.Authors.List(new QueryOptions<Author> {
            OrderBy = a => a.FirstName }),
        Genres = data.Genres.List(new QueryOptions<Genre> {
            OrderBy = g => g.Name }),
        CurrentRoute = builder.CurrentRoute,
        TotalPages = builder.GetTotalPages(data.Books.Count)
    };
    return View(vm);
}

[HttpPost]
public RedirectToActionResult Filter(string[] filter, bool clear = false) {
    if (clear) {
        builder.ClearFilterSegments();
    }
    else { // get author so you can add slug if needed
        var author = data.Authors.Get(filter[0].ToInt());
        builder.LoadFilterSegments(filter, author);
    }
    builder.SaveRouteSegments();

    var builder = new BooksGridBuilder(HttpContext.Session);
    return RedirectToAction("List", builder.CurrentRoute);
}
```

---

Figure 13-15 The Book/List view model and the Book controller

## The Book/List view

---

Figure 13-16 presents the Book/List view. This is a strongly-typed view whose model object is an instance of the BookListViewModel class.

The Razor code block at the top of the view file creates two RouteDictionary variables named current and routes. The current variable holds the CurrentRoute property of the view model, and the routes variable holds a copy of this property. As with the Author/List view, this first variable stays the same to preserve the current URL values, and the second changes as the code creates the URLs for the sorting and paging links.

The first `<form>` tag contains the drop-down lists for filtering. This form posts to the Filter() action method from the previous figure. You should notice three things about the `<select>` tags for these drop-down lists. First, they all have the same name attribute of “filter”, and this name attribute matches a parameter name of the Filter() action method. That way, the values stored in these `<select>` tags post to that action method as an array.

Second, each tag uses the `asp-items` tag helper and the SelectList class to populate the options for the drop-down lists. The SelectList constructor for each tag accepts the appropriate collection from the view model object, the name of the field for the value and text, and the currently selected value from the current RouteDictionary.

Third, in addition to the `<option>` tags that are generated by the `asp-items` tag helper, each `<select>` tag adds an `<option>` tag that displays “All” to the user. Each of these tags uses the public DefaultFilter constant of the BooksGridDTO class to set the value attribute of the option tag to “all”.

The first two `<select>` tags pass collections of Author and Genre objects, respectively, to the SelectList constructor. So, the next two constructor arguments identify which fields of those objects to use for the value and text. The third `<select>` tag, on the other hand, passes a string-string dictionary to the SelectList constructor. As a result, the next two arguments tell the SelectList object to use the dictionary key for the value and the dictionary value for the text.

Below the `<select>` tags, this view contains two submit buttons that post the form to the server. The second submit button, however, has a name attribute of “clear” and a value attribute of “true”. Since the name attribute of “clear” matches the clear parameter of the Filter() action method, clicking this button passes a value of true to that action method.

The second `<form>` tag contains an HTML table styled with Bootstrap that displays the book data. The code that displays this book data is similar to the code presented in figure 13-10 that displays author data. As a result, this code isn’t presented here. One difference is that each row of the book table includes an Add to Cart button. That’s why the table is coded within a `<form>` tag that posts to the Add() action method of the Cart controller. To do that, the Add to Cart button uses a name attribute of “id” and a value attribute that specifies the ID of the book.

Below the HTML table, a Razor code block builds the paging links. This works the same as the code presented in figure 13-10.

## The Book/List view

```
@model BookListViewModel

@{
    ViewData["Title"] = " | Book Catalog";

    RouteDictionary current = Model.CurrentRoute;
    RouteDictionary routes = Model.CurrentRoute.Clone();
}

<h1>Book Catalog</h1>

<form asp-action="Filter" method="post" class="form-inline">
    <label>Author: </label>
    <select name="filter" class="form-control m-2"
        asp-items="@new SelectList(
            Model.Authors, "AuthorId", "FullName", current.AuthorFilter)">
        <option value="@BooksGridDTO.DefaultFilter">All</option>
    </select>

    <label>Genre: </label>
    <select name="filter" class="form-control m-2"
        asp-items="@new SelectList(
            Model.Genres, "GenreId", "Name", current.GenreFilter)">
        <option value="@BooksGridDTO.DefaultFilter">All</option>
    </select>

    <label>Price: </label>
    <select name="filter" class="form-control m-2"
        asp-items="@new SelectList(
            Model.Prices, "Key", "Value", current.PriceFilter)">
        <option value="@BooksGridDTO.DefaultFilter">All</option>
    </select>

    <button type="submit" class="btn btn-primary mr-2">Filter</button>
    <button type="submit" class="btn btn-primary"
        name="clear" value="true">Clear</button>
</form>
<form asp-action="Add" asp-controller="Cart" method="post">
    <table class="table table-bordered table-striped table-sm">
        <!-- table with column headers for sorting - similar to figure 13-10 -->
    </table>
</form>

@{ // Razor code block with paging links - same as figure 13-10 }
```

### Description

- The select elements for filtering post to the Filter() action method of the Book controller.
- Each select element has the same name, so their values post as an array named filter. Additionally, the Clear submit button posts a value for a parameter named clear.
- Like the Author Catalog, this page displays a table of books with clickable column headers for sorting and paging links at the bottom.
- Each book has an Add to Cart button that posts to the Add() action method of the Cart controller.

---

Figure 13-16 The Book/List view

## The Cart page

The Bookstore website has a shopping cart that keeps track of books that a user has selected by clicking on the “Add to Cart” button. The app uses session state to store these books during the user’s session. In addition, the app stores the cart in a persistent cookie. That way, the Bookstore website “remembers” a user’s books between sessions.

### Extension methods for cookies

The Bookstore website uses extension methods to make it easier to work with cookies. Figure 13-17 shows these extension methods. Like the extension methods for session state shown in figure 13-3, these extension methods make it possible to store and retrieve complex objects. In addition, they make it easier to store integers and strings by adding methods like the ones for working with session state.

The extension methods that get values are added to the request cookie collection. The GetString() method just accepts a string key and returns the associated string value.

The GetInt32() method also accepts a string key. To start, it returns a string value for the associated string key. Then, it uses the TryParse() method of the int class to convert the string to an int value. If the TryParse() method returns true, the string value in the cookie was successfully converted to an int. As a result, the code returns the int value. Otherwise, the method returns null. To do that, though, it needs to cast null to match the return type of the method.

The GetObject<T>() method also accepts a string key. To start, it retrieves the string value associated with the key. If it’s null, the method returns the default value of type T. Otherwise, the method uses a static method of the JsonConvert class to convert the string value to an object of type T.

The extension methods that set values are added to the response cookie collection. Each set method accepts a key, a value, and an optional days argument with a default value of 30. As a result, these methods create a persistent cookie that expires after 30 days by default. If you want to create a session cookie, you must explicitly pass a value of 0 for the days argument. Of course, you can change these methods to create session cookies by default by changing the default value for this argument to 0.

The SetString() method starts by deleting any previous cookie for the specified key. Then, it creates and stores a session or persistent cookie based on the value of the days argument.

The SetInt32() method works by passing the arguments it receives to the SetString() method. But first, it converts the int value to a string by calling its ToString() method.

The SetObject<T>() method works similarly. However, it converts the object value it receives to a JSON string by calling the static SerializeObject<T>() method of the JsonConvert class.

## Extension methods for cookies

```
using Microsoft.AspNetCore.Http;
using Newtonsoft.Json;
...
public static class CookieExtensions
{
    public static string GetString(this IRequestCookieCollection cookies,
        string key) => cookies[key];

    public static int? GetInt32(this IRequestCookieCollection cookies,
        string key) => int.TryParse(cookies[key], out int i) ? i : (int?) null;

    public static T GetObject<T>(this IRequestCookieCollection cookies,
        string key)
    {
        var value = cookies[key];
        return value == null ? default(T) :
            JsonConvert.DeserializeObject<T>(value);
    }

    public static void SetString(this IResponseCookies cookies, string key,
        string value, int days = 30)
    {
        cookies.Delete(key); // delete old value first
        if (days == 0) { // session cookie
            cookies.Append(key, value);
        }
        else { // persistent cookie
            CookieOptions options = new CookieOptions {
                Expires = DateTime.Now.AddDays(days)
            };
            cookies.Append(key, value, options);
        }
    }

    public static void SetInt32(this IResponseCookies cookies, string key,
        int value, int days = 30) =>
        cookies.SetString(key, value.ToString(), days);

    public static void SetObject<T>(this IResponseCookies cookies, string key,
        T value, int days = 30) =>
        cookies.SetString(key, JsonConvert.SerializeObject(value), days);
}
```

## Description

- The `GetString()`, `GetInt32()`, `SetString()`, and `GetInt32()` extension methods make working with cookies similar to working with session state.
- The `GetObject()` and `SetObject()` extension methods use JSON to allow complex objects to be stored in and retrieved from cookies.

---

Figure 13-17 Extension methods for cookies

## The user interface

---

Figure 13-18 begins by showing the user interface for the Cart page. This page displays all the books in a user's shopping cart, as well as a subtotal of the cost of each book. The Cart page has buttons and links that allow the user to go to a checkout page, clear the cart, or continue shopping. In addition, each cart item has two buttons, one that allows the user to edit the item and another that removes the item from the cart.

To add a book to the cart, the user can click the "Add to Cart" button on either the Book Catalog page or the Book Details page. When a book is first added to the cart, it has a default quantity of 1. Then, to change that quantity, the user can edit the cart item by clicking on the Edit button. The edit page isn't shown in this chapter because it's similar to edit pages you've seen in previous chapters. However, you can view it by running the Bookstore website that's included in the download for this book.

The Cart page uses session state to store the books in the cart between requests. That's how the app can display the number of items in the cart in the badge that's displayed at the top of each page. In addition, the Cart page stores information about the cart items in a persistent cookie. As a result, if you add books to the shopping cart and close your web browser, the next time you use that web browser to view the Bookstore website it "remembers" those books.

## The model classes

---

Figure 13-18 also presents the CartItem class. The Bookstore website uses this class to store the selected book and quantity in session state. It also has a read-only Subtotal property that calculates the subtotal for the item by multiplying the selected book's price by the quantity for the item.

The CartItem class is designed to be serialized to JSON and stored in session state. You should be aware of two parts of this design. First, the Subtotal property is decorated with the `JsonIgnore` attribute. That's because this is a calculated value, so there's no reason to store it in session state.

Second, the Book property is of the BookDTO type rather than of the Book type. That's because the JSON serializer doesn't always work correctly with EF domain models like the Book class. One common issue is that the serializer ends up with circular references when it tries to follow all the navigation properties of a domain model. To fix this issue, the CartItem class uses the BookDTO class that's presented in the next figure.

Another good reason to use a DTO class for serialization is that sometimes, the domain model retrieves more data from the database than you want to store. Of course, you can always decorate any properties you don't want to store with the `JsonIgnore` attribute. But, if there are a lot of them, that can get tedious. In addition, even if you mark the properties to avoid serialization, you're still retrieving data from the database that you don't need. In a case like this, a DTO is a better option than a domain model.

## The Cart page

The screenshot shows a web browser window for 'Tuxedo Books | Cart' at the URL <https://localhost:5001/cart>. The page features a header with 'Home', 'Books', 'Authors', 'Cart 2', 'Register', and 'Admin'. Below the header is a logo of a tuxedo cat. The main content area is titled 'Your Cart' and displays a table with two items:

Title	Author(s)	Price	Quantity	Subtotal		
<a href="#">Between the World and Me</a>	Ta-Nehisi Coates	\$13.50	1	\$13.50	<a href="#">Edit</a>	<a href="#">Remove</a>
<a href="#">The Maltese Falcon</a>	Dashiell Hammett	\$10.99	1	\$10.99	<a href="#">Edit</a>	<a href="#">Remove</a>

At the top of the cart table, there is a message 'Subtotal: \$24.49' and three buttons: 'Checkout', 'Clear Cart', and 'Back to Shopping'.

## The CartItem class

```
using Newtonsoft.Json;
...
public class CartItem
{
    public BookDTO Book { get; set; }
    public int Quantity { get; set; }

    [JsonIgnore]
    public double Subtotal => Book.Price * Quantity;
}
```

Figure 13-18 The user interface for the Cart page and the CartItem class

Figure 13-19 begins by presenting the BookDTO class. This class holds only those properties of the Book class that are needed by the cart. When you use a DTO class like this, you need to be able to convert the domain model object to the DTO object. To do this, you can code a constructor for the BookDTO class that accepts a Book object and assigns the required data. You can also code a method for the Book class that creates, populates, and returns a BookDTO object. Or, you can add a Load() or Init() method to the BookDTO class that accepts a Book object and assigns the required data. That's the approach taken by the BookDTO class in this figure.

Another way to convert domain model objects to DTO objects and back again is to use a third-party mapping tool like AutoMapper. While that's beyond the scope of this book, this is the method that many developers prefer. If you want to learn more, many good resources are available online.

If you wanted to, you could use the SetObject<T>() and GetObject<T>() cookie extension methods to store and retrieve CartItem objects in cookies. In other words, you could have the data stored in the persistent cookie exactly match the data stored in session state.

However, many programmers prefer to store the minimum amount of data possible in a persistent cookie. Then, when the user returns to the web app and starts a new session, the app uses the data in the cookie to query a data store and restore the data in session state. That's the approach used by the Bookstore website.

The minimum data needed to restore a CartItem object is the ID of the book and the quantity value. That's why the Bookstore website uses the CartItemDTO class presented in this figure.

To facilitate converting CartItem objects to CartItemDTO objects, the Bookstore website adds an extension method to a list of CartItem objects as shown in this figure. Within this method, the code calls the LINQ Select() method of the list of CartItems and passes it a lambda expression. This lambda expression creates and populates a new CartItemDTO object. Then, it calls the ToList() method to create and return the list of CartItemDTO objects.

This figure also presents the CartViewModel class that's used to pass data to the Cart view. This view model is similar to the other view models presented in this chapter. To start, it contains a collection of CartItem objects to display and a RouteDictionary property to create links back to the Book Catalog page. In addition, it has a Subtotal property that provides a subtotal for all the books in the cart.

## The BookDTO class

```
public class BookDTO
{
    public int BookId { get; set; }
    public string Title { get; set; }
    public double Price { get; set; }
    public Dictionary<int, string> Authors { get; set; }

    public void Load(Book book)
    {
        BookId = book.BookId;
        Title = book.Title;
        Price = book.Price;
        Authors = new Dictionary<int, string>();
        foreach (BookAuthor ba in book.BookAuthors) {
            Authors.Add(ba.Author.AuthorId, ba.Author.FullName);
        }
    }
}
```

## The CartItemDTO class

```
public class CartItemDTO
{
    public int BookId { get; set; }
    public int Quantity { get; set; }
}
```

## An extension method for a list of CartItem objects

```
public static class CartItemListExtensions
{
    public static List<CartItemDTO> ToDTO(this List<CartItem> list) =>
        list.Select(ci => new CartItemDTO {
            BookId = ci.Book.BookId,
            Quantity = ci.Quantity
        }).ToList();
}
```

## The CartViewModel class

```
public class CartViewModel
{
    public IEnumerable<CartItem> List { get; set; }
    public RouteDictionary BookGridRoute { get; set; }
    public double Subtotal { get; set; }
}
```

---

Figure 13-19 The DTO classes and the CartViewModel class

Figure 13-20 presents the Cart class. The Bookstore website uses this class to store and retrieve cart data in session state and in a persistent cookie.

The Cart class starts by declaring two private string constants. It uses these constants as keys for the cart itself and for the number of items in the cart. This class stores the count value separately so the layout page doesn't need to retrieve and deserialize the entire cart to be able to display the item count in the navbar badge for the cart.

After the constants, the Cart class declares several private properties. The first two store a list of CartItem objects and a list of CartItemDTO objects, respectively. The next three store objects to work with session state and cookies.

After the private properties, the Cart class contains a constructor that accepts an HttpContext object. It uses this object to initialize all three of the session and cookie private properties.

After the constructor, the Cart class contains a method named Load() that gets the items for the cart from session state and loads them into the cart. Or, if session state doesn't contain any items, it attempts to use cookies to get the cart items from the database and load them into the cart.

To start, the Load() method uses the session property to get the collection of CartItem objects from session state. If there are no cart items in session state, the code initializes the private collection of CartItem objects and retrieves the collection of CartItemDTO objects from the persistent cookie.

After getting the cart items from the cookie, it checks whether the cart items need to be restored to session state. To do that, it compares the count of the stored items collection, which is retrieved from the cookie, to the count of the items collection, which is retrieved from session state. If there are more items in the stored items collection, session state needs to be reloaded.

To load items into session state, the code loops through the CartItemDTO objects in the stored items collection. For each DTO object, it uses the BookId property to retrieve a Book object from the database. Then, it checks whether that Book object is null. If so, the book was deleted from the database after the cookie was stored. As a result, the code doesn't process the book.

However, if the Book object contains data, the code creates a new BookDTO object and passes the Book object to the DTO's Load() method. Then, it creates a new CartItem object with the book and the quantity, and adds it to the items collection. After the loop finishes, this code calls the Save() method to save the new cart to session state and the persistent cookie.

The Cart class has several read-only properties. The Subtotal property uses the Sum() method of the items collection to add up the values of the Subtotal property for every CartItem object in the collection. The Count property returns the number of items in the cart. It gets this count from session state if possible. But, if there's no count value in session state, it gets the count from the persistent cookie. And the List property returns the CartItem objects in the private items collection.

**The Cart class****Page 1**

```
public class Cart
{
    private const string CartKey = "mycart";
    private const string CountKey = "mycount";

    private List<CartItem> items { get; set; }
    private List<CartItemDTO> storedItems { get; set; }

    private ISession session { get; set; }
    private IRequestCookieCollection requestCookies { get; set; }
    private IResponseCookies responseCookies { get; set; }

    public Cart(HttpContext ctx) {
        session = ctx.Session;
        requestCookies = ctx.Request.Cookies;
        responseCookies = ctx.Response.Cookies;
    }

    public void Load(Repository<Book> data)
    {
        items = session.GetObject<List<CartItem>>(CartKey);
        if (items == null) {
            items = new List<CartItem>();
            storedItems =
                requestCookies.GetObject<List<CartItemDTO>>(CartKey);
        }
        if (storedItems?.Count > items?.Count) {
            foreach (CartItemDTO storedItem in storedItems) {
                var book = data.Get(new QueryOptions<Book> {
                    Include = "BookAuthors.Author, Genre",
                    Where = b => b.BookId == storedItem.BookId
                });
                if (book != null) {
                    var dto = new BookDTO();
                    dto.Load(book);

                    CartItem item = new CartItem {
                        Book = dto,
                        Quantity = storedItem.Quantity
                    };
                    items.Add(item);
                }
            }
            Save();
        }
    }

    public double Subtotal => items.Sum(c => c.Subtotal);

    public int? Count => session.GetInt32(CountKey) ??
        requestCookies.GetInt32(CountKey);

    public IEnumerable<CartItem> List => items;
```

Figure 13-20 The Cart class (part 1)

The  `GetById()` method uses the `FirstOrDefault()` method of the `items` collection to get the `CartItem` object whose `Book` property has a `BookId` value that matches the value in the `ID` argument. If there's no `Book` object with that `ID`, this method returns `null`.

The `Add()` and `Edit()` methods each accept a `CartItem` object and use it to update the `items` collection. They both start by calling the  `GetById()` method to check if the `CartItem` object is already in the collection. They differ, however, in how they treat a `CartItem` object that already exists.

When the `Add()` method receives a `CartItem` object that's already in the `items` collection, it increments the value of the `Quantity` property by one. Otherwise, it calls the `Add()` method of the `items` collection to add the new object. By contrast, when the `Edit()` method receives a `CartItem` object that's already in the `items` collection, it replaces the previous value of the `Quantity` property with the new value.

The `Remove()` and `Clear()` methods both remove `CartItem` objects from the `items` collection. The difference is that the `Remove()` method accepts and removes a specific `CartItem` object, and the `Clear()` method removes all the `CartItem` objects in the collection.

The `Save()` method starts by checking how many `CartItem` objects are in the `items` collection. If there are none, the code removes all the cart items from session state and cookies. This way, when there aren't any items in the cart, the badge in the navbar doesn't display. If the code didn't do this, the badge in the navbar would display with a value of zero.

If there are `CartItem` objects in the `items` collection, the code stores the `items` collection and a separate count value in session state. In addition, the code stores a collection of `CartItemDTO` objects and a separate count value in persistent cookies. To do that, it gets the collection of `CartItemDTO` objects by calling the `ToDTO()` method of the `items` collection. This way, the number of items in the cart and the `BookId` value and `Quantity` value for each item are retained between sessions.

**The Cart class****Page 2**

```
public CartItem GetById(int id) =>
    items.FirstOrDefault(ci => ci.Book.BookId == id);

public void Add(CartItem item) {
    var itemInCart = GetById(item.Book.BookId);
    if (itemInCart == null) { // if new, add
        items.Add(item);
    }
    else { // otherwise, increase quantity by 1
        itemInCart.Quantity += 1;
    }
}

public void Edit(CartItem item) {
    var itemInCart = GetById(item.Book.BookId);
    if (itemInCart != null) {
        itemInCart.Quantity = item.Quantity;
    }
}

public void Remove(CartItem item) => items.Remove(item);

public void Clear() => items.Clear();

public void Save() {
    if (items.Count == 0) {
        session.Remove(CartKey);
        session.Remove(CountKey);
        responseCookies.Delete(CartKey);
        responseCookies.Delete(CountKey);
    }
    else {
        session.SetObject<List<CartItem>>(CartKey, items);
        session.SetInt32(CountKey, items.Count);
        responseCookies.SetObject<List<CartItemDTO>>(
            CartKey, items.ToDTO());
        responseCookies.SetInt32(CountKey, items.Count);
    }
}
}
```

**Description**

- The Cart class stores and retrieves the books a user places in their cart. Since it uses session state and persistent cookies to do this, it remembers a user's cart between sessions.

---

Figure 13-20 The Cart class (part 2)

## The Cart controller

---

Figure 13-21 presents several methods of the Cart controller. While not shown here, this controller's constructor accepts a BookstoreContext object that it uses to initialize a Repository<Book> object that it stores in a private property named `data`.

The private `GetCart()` helper method gets a `Cart` object and loads any cart items from session state or from a persistent cookie. To do that, this code creates a `Cart` object by passing the controller's `HttpContext` property to the `Cart()` constructor. Then, it calls the `Load()` method of the `Cart` object to load the cart items. Finally, it returns the loaded `Cart` object.

The `Index()` action method runs when a user navigates to the Cart page. To start, it uses the `GetCart()` method to get a loaded `Cart` object. Then, it creates a new `BooksGridBuilder` object and passes the controller's `HttpContext.Session` property to the constructor. This constructor retrieves any paging, sorting, and filtering URL route segments in session state. Then, the code uses the `Cart` and `BooksGridBuilder` objects to populate a `CartViewModel` object with the data needed by the `Index` view. Specifically, this is a list of cart items, a subtotal amount for the cart, and a `RouteDictionary` containing the user's paging, sorting, and filtering URL segments.

The `Add()` action method runs when a user clicks an Add to Cart button on the Book Catalog or Book Details page. It accepts an `int` argument and uses it to get the selected book from the database. If that book isn't in the database, it notifies the user.

Otherwise, if the book is in the database, the code creates a new `BookDTO` object and passes the `Book` object to its `Load()` method. Then, it initializes a new `CartItem` object with the `BookDTO` value and a default `Quantity` value of 1. Next, the code gets a `Cart` object, passes the `CartItem` object to its `Add()` method, and calls its `Save()` method. This stores the updated cart information in session state and in a persistent cookie. Finally, the code stores a message in `TempData`.

After the if-else statement, the code creates a new `BooksGridBuilder` object. Then, it uses the `CurrentRoute` value of that object to redirect the user back to the Book Catalog page.

The `Remove()` action method runs when the user clicks a Delete button for a cart item. Then, it gets the `Cart` object, gets the item with the specified ID from the cart, removes that item from the cart, and calls the cart's `Save()` method to update session state and the persistent cookie. Next, it stores a message in `TempData` and redirects to the main Cart page.

The `Clear()` action method isn't shown here due to space considerations. However, it works similarly to the `Remove()` action method. The main difference is that the `Clear()` method removes all items from the cart.

Similarly, the `Edit()` action methods for GET and POST requests aren't shown here due to space considerations. However, they're similar to examples from earlier chapters. If you want, you can review the code for these methods in the download for this book.

## Some methods of the Cart controller

```
private Cart GetCart() {
    var cart = new Cart(HttpContext);
    cart.Load(data);
    return cart;
}

public ViewResult Index() {
    Cart cart = GetCart();
    var builder = new BooksGridBuilder(HttpContext.Session);
    var vm = new CartViewModel {
        List = cart.List,
        Subtotal = cart.Subtotal,
        BookGridRoute = builder.CurrentRoute
    };
    return View(vm);
}

[HttpPost]
public RedirectToActionResult Add(int id) {
    var book = data.Get(new QueryOptions<Book> {
        Include = "BookAuthors.Author, Genre",
        Where = b => b.BookId == id
    });
    if (book == null) {
        TempData["message"] = "Unable to add book to cart.";
    }
    else {
        var dto = new BookDTO();
        dto.Load(book);
        CartItem item = new CartItem {
            Book = dto, Quantity = 1 // default quantity
        };
        Cart cart = GetCart();
        cart.Add(item);
        cart.Save();
        TempData["message"] = $"{book.Title} added to cart";
    }
    var builder = new BooksGridBuilder(HttpContext.Session);
    return RedirectToAction("List", "Book", builder.CurrentRoute);
}

[HttpPost]
public RedirectToActionResult Remove(int id) {
    Cart cart = GetCart();
    CartItem item = cart.GetById(id);
    cart.Remove(item);
    cart.Save();
    TempData["message"] = $"{item.Book.Title} removed from cart.";
    return RedirectToAction("Index");
}

// Edit(), Clear(), and Checkout() methods not shown here
```

---

Figure 13-21 The Cart controller

## The Cart/Index view

---

Figure 13-22 presents the Cart/Index view. This is a strongly-typed view whose model object is an instance of the CartViewModel class.

The HTML for the unordered list at the top of the view file is coded within a <form> tag that posts to the Clear() action method of the Cart controller. In this form, the submit button posts to the server. However, this form doesn't post any data to the server when the user clicks on the submit button. That's because the Clear() action method doesn't have any parameters. As a result, it doesn't need any data from the view.

This form also contains one link to the Checkout() action method of the Cart controller and another to the List() action method of the Book() controller. Here, the link to the List() action method uses the asp-all-route-data tag helper and the BookGridRoute property of the view model to build the URL for the link so it uses any paging, sorting, and filtering route segments stored in session state. That way, any selections you make on the Book Catalog page aren't lost when you navigate to the cart and back. This form also uses the Subtotal property of the view model to display a subtotal value of all items in the cart to the user.

The second <form> tag posts to the Remove() action method of the Cart controller. This form contains an HTML table that displays the cart items in the List property of the view model. The last table cell for each cart item contains an Edit link and a Delete button, though the Edit link is styled so it looks the same as the button.

The Edit link redirects to the Edit() action method in the Cart controller that handles GET requests. To do that, it uses the BookId and Title properties of the Book property of the current CartItem object to add the values for the ID and slug route parameters.

The Remove button posts the form to the server and sends the ID of the book to remove by assigning a value of "id" to its name attribute and by assigning the BookId value to its value attribute. This passes the ID of the current book as the ID parameter of the Remove() action method.

## The Cart/Index view

```
@model CartViewModel
...
<h1>Your Cart</h1>
<form asp-action="Clear" method="post">
    <ul class="list-group mb-4">
        <li class="list-group-item">
            <div class="row">
                <div class="col">Subtotal: @Model.Subtotal.ToString("c")</div>
                <div class="col">
                    <div class="float-right">
                        <a asp-action="Checkout" class="btn btn-primary">Checkout</a>
                        <button type="submit" class="btn btn-primary">
                            Clear Cart</button>
                        <a asp-action="List" asp-controller="Book"
                            asp-all-route-data="@Model.BookGridRoute">
                            Back to Shopping</a>
                    </div>
                </div>
            </div>
        </li>
    </ul>
</form>
<form asp-action="Remove" method="post">
    <table class="table">
        <thead class="thead-dark">
            <tr><th>Title</th><th>Author(s)</th><th>Price</th><th>Quantity</th>
            <th>Subtotal</th><th></th></tr>
        </thead>
        <tbody>
            @foreach (CartItem item in Model.List) {
                <tr>
                    <td>
                        <a asp-action="Details" asp-controller="Book"
                            asp-route-id="@item.Book.BookId"
                            asp-route-slug="@item.Book.Title.Slug()">@item.Book.Title
                        </a></td>
                    <td>
                        @foreach (var keyValuePair in item.Book.Authors) {
                            <p><a asp-action="Details" asp-controller="Author"
                                asp-route-id="@keyValuePair.Key"
                                asp-route-slug="@keyValuePair.Value.Slug()">
                                @keyValuePair.Value</a></p>
                        }
                    </td>
                    <!-- cells that display book price, quantity, and subtotal -->
                    <td>
                        <div class="float-right">
                            <a asp-action="Edit" asp-controller="Cart"
                                asp-route-id="@item.Book.BookId"
                                asp-route-slug="@item.Book.Title.Slug()"
                                class="btn btn-primary">Edit</a>
                            <button type="submit" name="id" value="@item.Book.BookId"
                                class="btn btn-primary">Remove</button>
                        </div>
                    </td>
                </tr>
            }
        </tbody></table></form>
```

---

Figure 13-22 The Cart/Index view

## The book search of the Admin pages

The Admin pages allow you to add, edit, or delete books, authors, or genres. These pages are in their own area named Admin, which allows them to have controllers named BookController and AuthorController that don't conflict with the controllers of the same name in the main Controllers folder.

### The user interface

Most of the code for the Admin area is similar to the add, edit, and delete code presented earlier in this book. However, the Admin area has a search functionality that's different from what's been presented so far. It allows a user to search for books to add, edit, or delete. To do that, the user can search by title, author, or genre. The page that displays the search results is also used to display the books related to an author or genre that can't be deleted because they're related to books. You'll see how this works in the next two figures.

The first screen in figure 13-23 shows the Manage Books tab of the Admin page when it's first displayed. The second screen shows this tab after a user has entered the search term "pride", selected the search option of Title, and clicked the Find button. This displays two books that have the word "pride" in their titles.

The search results for each book includes title, genre, and author information. Each title is a link that the user can click to navigate to the Book Details page. In addition, Edit and Delete buttons are available for each book. This chapter doesn't show the pages that these buttons navigate to, but they're similar to the pages presented in earlier chapters.

This book search functionality uses TempData to store the search term. That's because the search term is only needed for a short time and doesn't need to persist throughout the user's entire session. However, there may be times when it should persist after being read.

For instance, let's say the user performs this search and then wants to edit *Pride and Prejudice* and delete *Pride and Prejudice and Zombies*. With a normal read of TempData, TempData is cleared once these search results are displayed. Because of that, after the user edited the first book, the search would need to be done again before the second book could be deleted. As a result, the search code uses the Peek() method to read TempData instead. That way, the user can take more than one action on the search results.

However, that means the code must at some point explicitly remove the search term from TempData. This makes it work more like session state, and you could easily use session state instead of TempData here. However, since this search data should be temporary, TempData is a better fit.

## The Manage Books tab of the Admin page

A screenshot of a web browser window titled "Tuxedo Books | Manage Books". The URL is https://localhost:5001/admin. The page features a header with "Home", "Books", "Authors", "Cart 2", "Register", and "Admin" links. A logo of a black cat is visible. Below the header, the title "Tuxedo Books" is displayed next to the cat logo. A navigation bar at the top includes "Manage Books", "Manage Authors", and "Manage Genres". A sub-section titled "Manage Books" contains a "Add a book" button and a search form with fields for "Find a book", "Search Term:", "Search By: Title", "Author", "Genre", and a "Find" button.

## The Manage Books tab with search results

A screenshot of a web browser window titled "Tuxedo Books | Search Results" with the URL localhost:44364/admin/book/search. The page layout is identical to the previous screenshot, featuring the same header, logo, and navigation links. The main content area displays the results of a search for the book title 'pride'. The results table shows two entries:

Book Title	Genre	Author(s)	Action
Pride and Prejudice	Novel	Jane Austen	Edit Delete
Pride and Prejudice and Zombies	Novel	Jane Austen Seth Grahame-Smith	Edit Delete

Figure 13-23 The user interface for the Admin page

## The **SearchData** and **SearchViewModel** classes

---

The first code example in figure 13-24 presents the **SearchData** class. The Bookstore website uses this class to store and retrieve search data from **TempData**.

The **SearchData** class starts by declaring two private string constants. Then, it declares a private property for a **TempData** object. The constructor for this class accepts a **TempData** object and stores it in this property.

The **SearchData** class has six public properties. The **SearchTerm** and **Type** properties have setters that use the private constants as keys to store the search term and search type in **TempData**. In addition, they have getters that use the private constants as keys to retrieve these items from **TempData**.

For these properties, the getters use the **Peek()** method of the **TempData** class to retrieve the items, rather than reading them directly. As a result, an item isn't automatically deleted when the request ends. This way, a user can edit or delete a book from the list of search results, and the app remembers the search term and type when the search results are displayed again. Otherwise, the user would have to re-enter the search term and type each time.

The four read-only properties are Boolean flags that provide information about the search term and type that are stored in **TempData**. Specifically, they indicate whether there's a search term and if the type of search is by book title, author, or genre. Finally, the **Clear()** method removes the search term and type from **TempData**.

The second code example presents the **SearchViewModel** class that's used to pass data to the search results view. This view model includes a collection of **Book** objects to display and three string properties that hold the search term, the type of search, and the header text for the view. In addition, the **SearchTerm** property is decorated with the **Required** data attribute. This way, the app makes sure that the user enters a search term.

## The SearchData class

```
using Microsoft.AspNetCore.Mvc.ViewFeatures;
...
public class searchData
{
    private const string SearchKey = "search";
    private const string TypeKey = "searchtype";

    private ITempDataDictionary tempData { get; set; }
    public searchData(ITempDataDictionary temp) =>
        tempData = temp;

    // use Peek() rather than a straight read so value will persist
    public string SearchTerm {
        get => tempData.Peek(SearchKey)?.ToString();
        set => tempData[SearchKey] = value;
    }
    public string Type {
        get => tempData.Peek(TypeKey)?.ToString();
        set => tempData[TypeKey] = value;
    }

    public bool HasSearchTerm => !string.IsNullOrEmpty(SearchTerm);
    public bool IsBook => Type.EqualsNoCase("book");
    public bool IsAuthor => Type.EqualsNoCase("author");
    public bool IsGenre => Type.EqualsNoCase("genre");

    public void Clear() {
        tempData.Remove(SearchKey);
        tempData.Remove(TypeKey);
    }
}
```

## The SearchViewModel class

```
using System.ComponentModel.DataAnnotations;
...
public class SearchViewModel
{
    public IEnumerable<Book> Books { get; set; }
    [Required(ErrorMessage = "Please enter a search term.")]
    public string SearchTerm { get; set; }
    public string Type { get; set; }
    public string Header { get; set; }
}
```

### Description

- The `SearchData` class gets search terms and search types into and out of `TempData`.
- The property getters use the `Peek()` method rather than a straight read so the value will persist in `TempData`. That way, if a user navigates back to the search results page from the edit or delete page, their search data isn't lost.
- The `SearchViewModel` class contains data needed by the `Search` view. The `SearchTerm` property is decorated with a `Required` attribute to make sure the user enters a search value.

---

Figure 13-24 The `SearchData` and `SearchViewModel` classes

## The Search() action methods of the Book controller

---

Figure 13-25 shows the overloaded Search() action methods in the Book controller of the Admin area. The Search() action method for POST requests starts by making sure the user entered valid data. In this case, that means the user has entered a search term.

If the model is valid, the code initializes a new SearchData object. It passes the controller's TempData property to the constructor and uses the values in the SearchViewModel parameter to set the SearchTerm and Type properties. Then, it redirects to the Search() action method for GET requests.

The Search() action method for GET requests starts by creating a new SearchData object and passing it the controller's TempData property. Then, it uses the HasSearchTerm flag to check if there's a search term in TempData. If not, it displays the Manage Books page again. Otherwise, it uses the search term to initialize a new SearchViewModel object.

After creating the view model, this code creates an options object to query the database for books that match the search criteria. For the Where expression, the code checks the flag properties that indicate the type of search. As it adds the correct Where expression, it assigns a message to the Header property of the view model.

For a search by book title or genre, the Where expressions are straightforward. They each pass the search term to the Contains() method of the property being searched on. For a book, that's the Title property. For a genre, that's the GenreId property. This works because the GenreId is a short string. As a result, the user can search for all books in a genre by entering all or part of this short string. For example, the user can enter "hist" to get all books in the History genre or "scifi" to get all books in the Science Fiction genre.

On the other hand, a search by author results in a more complicated Where expression. That's because it works with nested objects and attempts to determine whether the search term is a single word or multiple words. To do that, it looks for a space in the search term. If there isn't one, the user has entered a single word. In this case, the code requests any book with an author whose first name or last name contains that word.

If there is a space, though, the code splits the search term into two words. Then, it requests any book with an author whose first name contains the first word and whose last name contains the second word.

Once the options object is built, the code uses it to query the database and assigns the returned list of Book objects to the Books property of the view model. Finally, it passes the view model to the view, where the search results are displayed to the user.

## The Search() action methods of the Book controller

```
[HttpPost]
public RedirectToActionResult Search(SearchViewModel vm) {
    if (ModelState.IsValid) {
        var search = new searchData(TempData) {
            SearchTerm = vm.SearchTerm,
            Type = vm.Type
        };
        return RedirectToAction("Search");
    }
    else {
        return RedirectToAction("Index");
    }
}

[HttpGet]
public ViewResult Search() {
    var search = new searchData(TempData);

    if (search.HasSearchTerm) {
        var vm = new SearchViewModel {
            SearchTerm = search.SearchTerm
        };
        var options = new QueryOptions<Book> {
            Include = "Genre, BookAuthors.Author"
        };
        if (search.IsBook) {
            options.Where = b => b.Title.Contains(vm.SearchTerm);
            vm.Header = $"Search results for book title '{vm.SearchTerm}'";
        }
        if (search.IsAuthor) {
            int index = vm.SearchTerm.LastIndexOf(' ');
            if (index == -1) {
                options.Where = b => b.BookAuthors.Any(
                    ba => ba.Author.FirstName.Contains(vm.SearchTerm) ||
                    ba.Author.LastName.Contains(vm.SearchTerm));
            }
            else {
                string first = vm.SearchTerm.Substring(0, index);
                string last = vm.SearchTerm.Substring(index + 1);
                options.Where = b => b.BookAuthors.Any(
                    ba => ba.Author.FirstName.Contains(first) &&
                    ba.Author.LastName.Contains(last));
            }
            vm.Header = $"Search results for author '{vm.SearchTerm}'";
        }
        if (search.IsGenre) {
            options.Where = b => b.GenreId.Contains(vm.SearchTerm);
            vm.Header = $"Search results for genre ID '{vm.SearchTerm}'";
        }
        vm.Books = data.Books.List(options);
        return View("SearchResults", vm);
    }
    else {
        return View("Index");
    }
}
```

---

Figure 13-25 The Search() action methods of the Book controller

## The Delete() action method of the Genre controller

---

Figure 13-26 shows how the search functionality is used by the Delete() action method of the Genre controller that handles GET requests. This action method uses the ID argument it receives to retrieve a Genre object from the database. This object includes all related books.

After getting the Genre object, the code checks whether the genre contains any related books. If so, this genre can't be deleted. In that case, the GoToBookSearchResults() helper method is called. The code for this method uses the SearchData class to store the genre ID as a search by genre. Then, it returns a RedirectToActionResult object that redirects to the Search() action method of the Book controller. The Delete() action method, in turn, returns that type. This will display the related books to the user and give them a chance to delete them if they wish. The delete functionality for an author works similarly.

## The Delete() action method of the Genre controller

```
[HttpGet]
public IActionResult Delete(string id) {
    var genre = data.Get(new QueryOptions<Genre> {
        Include = "Books",
        Where = g => g.GenreId == id
    });

    if (genre.Books.Count > 0) {
        TempData["message"] = $"Can't delete genre {genre.Name} "
            + "because it's associated with these books.";
        return GoToBookSearchResults(id);
    }
    else {
        return View("Genre", genre);
    }
}

// private helper method
private RedirectToActionResult GoToBookSearchResults(string id)
{
    // display search results of all books in this genre
    var search = new searchData(TempData) {
        SearchTerm = id,
        Type = "genre"
    };
    return RedirectToAction("Search", "Book");
}
```

---

Figure 13-26 The Delete() action method of the Genre controller

## Perspective

The Bookstore website presented in this chapter uses professional coding techniques that are fairly abstract and may be difficult to understand at first. If you don't understand all of the code presented in this chapter right away, don't be discouraged. At first, it's OK if you only get the general idea of how this website works.

To help you understand this website, you can start by running it to see how it works. As you do this, you can study the source code that makes it work the way it does. At times, you may need to refer to earlier chapters to refresh your memory about how something works. But that's how you learn.

As you study this website, you will probably see many ways in which it can be improved. If you do, you should try to make some of those improvements. That gives you a chance to modify or enhance someone else's code, which is a common practice in the real world. In addition, it demonstrates the value of a logical folder structure and the use of the MVC pattern. And that's a great way to learn.

### Exercise 13-1 Run and test the Bookstore web app

In this exercise, you'll run the Bookstore web app and test it to see how it works. Then, you'll add the ability to change the number of books that display per page.

#### Run the Bookstore web app and test it

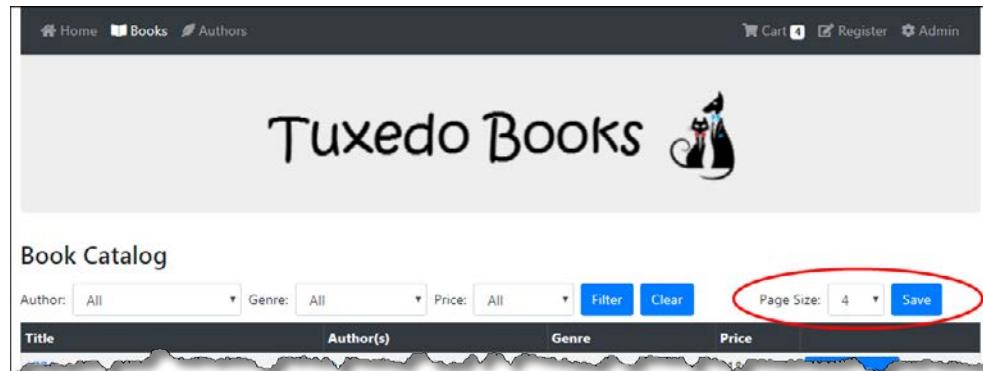
1. Open the Ch13Ex1Bookstore app in the ex\_starts directory.
2. If you haven't already created the database for the Bookstore app, open the Package Manager Console and enter the Update-Database command.
3. Run the app. When the Home page loads, it should display a "Staff Selection" book title. Refresh the page or click the Home nav link once or twice and note that the "Staff Selection" changes.
4. Click on each of the nav links. With the exception of the Registration link, which isn't implemented yet, the app should mark the current link as active.
5. Navigate to the Author Catalog page and click on the Last Name column header to sort the authors by last name. Click that header again to reverse the sort order. Then, click the paging links to page through the authors.
6. Click on an author's first or last name to go to the author details page. From there, click on a book name to go to the book details page.
7. Navigate to the Book Catalog page and experiment with sorting and paging there too. In addition, experiment with the drop-down lists for filtering.
8. Add two or more books to the cart. Note that adding a book displays a notification message on the Book Catalog page. Then, click on a sorting or paging link and note that the notification message disappears.

9. Navigate to the Cart page and review your cart. Click on the Edit button for a book and increase its quantity. Then, delete that book. Finally, click the Clear Cart button to clear the cart.
10. Navigate to the Admin area. In the Manage Books tab, enter “pride” in the Find text box, leave the radio button at its default of searching by title, and click Find. From the search results, select a book, click Edit, change the price, and click Save.
11. Navigate to the Manage Books tab again. Enter the term “Novel”, select Genre from the radio buttons to search by genre, and click Find. This should display seven books.
12. Navigate to the Manage Genres tab and attempt to delete the Novel genre. Note that you can't delete this genre because it contains the same seven books displayed in the previous step.
13. Experiment in the Admin area by adding, editing, and deleting a book, an author, and a genre.

### Use the URL to change the number of books per page

14. Navigate to the Book Catalog page and click one of the paging links.
15. In the browser, review the URL and find the size route segment. Then, change its value from 4 to 10 and press the Enter key. This should display 10 books per page.

### Add a drop-down list for changing the number of books per page



16. In the Models/ViewModels folder, open the BookListViewModel class. Add a read-only property named PageSizes that returns an int array that contains the values 1 through 10.
17. In the Controllers folder, open the BookController class and add an action method for POST requests named PageSize(). This action method should return a RedirectToAction object and accept an int named pagesize.
18. In the PageSize() action method, create a new BooksGridBuilder object. To do that, you can use the code in the Filter() action method as a guide.

19. Use the BooksGridBuilder object to update the PageSize property of the CurrentRoute property with the pagesize parameter. Call the SaveRouteSegments() method of the BooksGridBuilder object. And use the CurrentRoute value to redirect to the List() action method. Again, you can use the code in the Filter() action method as a guide.
20. In the Views folder, open the Book/List view and add a <form> element that posts to the PageSize() action method.
21. Within the <form> element, add a <select> element and a submit button. Make sure the name of the <select> element matches the name of the parameter for the PageSize() action method.
22. Use the asp-items tag helper to add the options available from the PageSizes array of the view model. This should include an argument that marks the current page size as selected. When you're done, the code should look like this:

```
asp-items="@new SelectList(Model.PageSizes, current.PageSize)"
```

23. To display the Page Size drop-down list next to the Filter drop-down lists, you can add a Bootstrap grid with the first column 9 units wide. Then, you can place the <form> element for filtering in the first column, and the <form> element for the page size in the second column like this:

```
<div class="row">
    <div class="col-9">
        @* filter form *@
    </div>
    <div class="col">
        @* page-size form *@
    </div>
</div>
```

24. Run the app and use the new drop-down list to change the page size.

# Section 3

## Add more skills as you need them

Sections 1 and 2 of this book presented the ASP.NET Core MVC skills that you need to build a complex database-driven website such as the Bookstore website presented in chapter 13. Now, this section presents more ASP.NET Core MVC skills that are commonly needed for real-world websites. To make it easy to learn these skills as you need them, each chapter in this section is written as an independent training module. As a result, you can read them in whatever sequence you prefer.

To start, chapter 14 shows how to use dependency injection (DI) to make your code more flexible and easier to test. In addition, it shows how to automate the testing of an app with unit testing.

Chapter 15 shows how to create and use custom tag helpers, partial views, and view components. These features provide a way to reduce code duplication in your views, which makes them more flexible and easier to maintain.

Chapter 16 shows how to restrict access to parts of a website such as the Admin area of a website. Then, it shows how to authenticate users and allow authorized users with the appropriate privileges to access restricted parts of the website.

To finish, chapter 17 shows how to use Visual Studio Code. This source code editor is a popular alternative to the Visual Studio IDE that has been presented so far in this book.



# How to use dependency injection and unit testing

Dependency injection (DI) is a design pattern that can make your code more flexible and easier to change. In addition, DI makes it possible to automate testing of a web app with unit testing. However, dependency injection also increases the complexity of an app. Because of that, it's less useful if you don't use unit testing or if your app is small and unlikely to change much.

In this chapter, you'll learn how to use DI to improve the Bookstore website presented in the previous chapter. This includes how to use unit testing to automate testing of that website.

<b>How to use dependency injection (DI) .....</b>	<b>560</b>
How to configure your app for DI.....	560
How to use DI with controllers.....	562
How to use DI with an HttpContextAccessor object.....	564
How to use DI with action methods .....	566
How to use DI with views.....	568
<b>How to get started with unit testing .....</b>	<b>570</b>
How unit tests work .....	570
How to add an xUnit project to a solution .....	572
How to write a unit test.....	574
How to run a unit test .....	576
<b>How to test methods that have dependencies .....</b>	<b>578</b>
How to use a fake repository object .....	578
How to use a fake TempData object .....	580
<b>How to create fake objects with Moq .....</b>	<b>582</b>
How to work with mock objects .....	582
How to mock a repository object.....	584
How to mock a TempData object .....	584
How to mock an HttpContextAccessor object .....	586
<b>The Bookstore.Tests project .....</b>	<b>588</b>
The Test Explorer.....	588
The BookControllerTests class .....	590
The AdminBookControllerTests class.....	592
The CartTests class .....	596
<b>Perspective .....</b>	<b>600</b>

## How to use dependency injection (DI)

Often, an object depends on another object to do its work. For example, a controller that works with a database depends on a DbContext object. You can code that other object, known as a *dependency*, within the containing object. However, if you do that and the dependency changes, you need to modify the containing object.

A better way to do this is to use a design pattern called *dependency injection (DI)*. With DI, you code the containing object with a constructor that accepts an interface. Then, you can *inject* the dependency by passing an object that implements the interface to the constructor. That way, you can easily change the dependency by passing a different object to the constructor. This will work as long as the object implements the correct interface. And, as you'll see later in this chapter, coding an object this way also makes it easier to test.

### How to configure your app for DI

To use DI with MVC, you must register, or *map*, your dependencies. In other words, you must tell MVC what object to inject for an interface parameter. To do that, you add code to the ConfigureServices() method in the Startup.cs file.

The ConfigureServices() method has a parameter named services that's of the IServiceCollection type. The table in figure 14-1 presents three methods of this object that you can use to map dependencies.

When you map a dependency, you must choose a *life cycle* for the dependency. With the transient life cycle, MVC creates a new object every time it needs to inject a dependency.

With the scoped life cycle, MVC creates a new object the first time it needs to inject a dependency. Then, for the rest of the current scope, MVC reuses that object for each subsequent injection. In a web app, the current scope is usually the current HTTP request.

With the singleton life cycle, MVC creates a new object the first time it needs to inject a dependency. Then, MVC reuses that object for each subsequent injection.

Most of the time, you'll use the transient lifecycle. If you use the longer life cycles, you should make sure that the object being injected can handle concurrency.

The first code example in this figure shows how to map a dependency for a class that doesn't use generics. Here, the AddTransient() method tells MVC to create and inject an instance of the BookstoreUnitOfWork class every time it encounters an IBookstoreUnitOfWork parameter. To do that, this example specifies the interface and class types within the angle brackets of the method call, not within the parentheses.

The second code example shows how to map a dependency for a generic class. Here, the AddTransient() method tells MVC to create and inject an instance of the Repository<T> class every time it encounters an IRepository<T> parameter. This time, the interface and class types are passed within the parentheses of the method call, not within the angle brackets.

### Three methods of the `IServiceCollection` object that map dependencies

Method	Description
<code>AddTransient&lt;interface, class&gt;()</code>	Transient life cycle. MVC creates a new instance of the class every time it needs to inject a dependency.
<code>AddScoped&lt;interface, class&gt;()</code>	Scoped life cycle. MVC creates a new instance of the class the first time it needs to inject a dependency and reuses that instance for all subsequent injections in the scope. Usually, a scope is an HTTP request.
<code>AddSingleton&lt;interface, class&gt;()</code>	Singleton life cycle. MVC creates a new instance of the class the first time it needs to inject a dependency and reuses that instance for all subsequent injections.

### How to configure DI for a class that doesn't use generics

```
public void ConfigureServices(IServiceCollection services)
{
    ...
    services.AddTransient<IBookstoreUnitOfWork, BookstoreUnitOfWork>();
    ...
}
```

### How to configure DI for a generic class

```
services.AddTransient(typeof(IRepository<>), typeof(Repository<>));
```

### How to configure DI for the HTTP context accessor

#### Manually

```
services.AddSingleton< IHttpContextAccessor, HttpContextAccessor>();
```

#### With a method of the `IServiceCollection` object

```
services.AddHttpContextAccessor();
```

### Description

- *Dependency injection (DI)* is a design pattern in which the *dependencies* needed by an object are passed as parameters rather than being coded as part of the object.
- To use DI, you code the constructor of the object so it accepts an interface. Then, you can *inject* the dependency by passing any object that implements the interface to the constructor.
- DI makes code easier to change and facilitates unit testing.
- To use DI with MVC, you must register, or *map*, your dependencies in the Startup.cs file.
- The `IServiceCollection` parameter of the `ConfigureServices()` method has methods you can use to map dependencies.
- When you map dependencies, you must decide which *dependency life cycle* to use.
- You can map a dependency for the `HttpContextAccessor` class manually or with the `AddHttpContextAccessor()` method of the `IServiceCollection` object.

---

Figure 14-1 How to configure your app for DI

The third code example shows how to map a dependency for the HTTP context accessor. This allows you to use DI to work with objects like session state or cookies. To do that, you can manually map the `IHttpContextAccessor` interface and the `HttpContextAccessor` class as a singleton dependency. Or, you can call the `AddHttpContextAccessor()` method of the `IServiceCollection` object.

## How to use DI with controllers

---

Once you've mapped your dependencies, you can request instances of the mapped classes throughout your app. The code in figure 14-2 shows how this works. In particular, these examples show how DI makes the controllers in the Bookstore app from the previous chapter more flexible. Later, this chapter shows how DI also makes these controllers easy to test.

The first code example shows a controller that receives a `DbContext` parameter via DI. More specifically, it shows the `Author` controller from the previous chapter that accepts a `BookstoreContext` object. MVC uses DI to pass an instance of the `BookstoreContext` class to this controller.

This makes it easy to use classes that inherit the `DbContext` class of EF Core in controllers. However, it also makes those controllers *tightly coupled* with EF Core. As a result, if you ever wanted to change your data access to something other than EF Core, you'd need to change your `Repository` class, *and* you'd need to change every controller that receives a `DbContext` class. In a large app, that could be quite an undertaking!

The second code example shows the controller from the first example after it has been modified to receive a `Repository` object via DI. To do that, MVC uses the `IRepository<T>/Repository<T>` mapping shown in the previous figure. Then, when MVC encounters the `IRepository<Author>` parameter, it injects an object created from the `Repository<Author>` class. For this to work, the data type of the private data property must also be `IRepository<Author>`, not `Repository<Author>`. In addition, the `Repository<T>` class must implement the `IRepository<T>` interface, and it must accept a `DbContext` object in its constructor as shown in chapter 12.

Since the controller in the second example only accesses the `IRepository` interface, there's no tight coupling with the `DbContext` class of EF. In other words, it's now *loosely coupled* with EF. As a result, you can change data access from EF to another framework without affecting this controller.

So, in this second example, how does the `Repository<Author>` class get the `BookstoreContext` class it needs? Well, when MVC creates an object to inject, it inspects the constructors of that class, and it injects objects according to its mappings. This is called *dependency chaining*.

The third and fourth examples work like the first two examples. However, they work with a unit of work class, not a repository class. In particular, they work with the `BookstoreUnitOfWork` class that's used by the Book controllers described in the previous chapter.

## An Author controller

### That injects a DbContext object

```
public class AuthorController : Controller
{
    private Repository<Author> data { get; set; }

    public AuthorController(BookstoreContext ctx) =>
        data = new Repository<Author>(ctx);
    ...
}
```

### That injects a repository object

```
public class AuthorController : Controller
{
    private IRepository<Author> data { get; set; }

    public AuthorController(IRepository<Author> rep) =>
        data = rep;
    ...
}
```

## A Book controller

### That injects a DbContext object

```
public class BookController : Controller
{
    private BookstoreUnitOfWork data { get; set; }

    public BookController(BookstoreContext ctx) =>
        data = new BookstoreUnitOfWork(ctx);
    ...
}
```

### That injects a unit of work object

```
public class BookController : Controller
{
    private IBookstoreUnitOfWork data { get; set; }

    public BookController(IBookstoreUnitOfWork unit) => data = unit;
    ...
}
```

## Description

- Controllers that receive DbContext objects via dependency injection are *tightly coupled* with EF Core because they must specify a class derived from the DbContext base class.
- Controllers that receive repository objects or unit of work objects via dependency injection are *loosely coupled* with EF Core because they only need to specify the appropriate interface, which may or may not use EF Core.
- When MVC creates an object to inject, it inspects the constructors of the class and injects objects according to the mappings in the Startup.cs file. This is called *dependency chaining*.

---

Figure 14-2 How to use DI with controllers

## How to use DI with an **HttpContextAccessor** object

Earlier in this chapter, you learned how to map a dependency for the `IHttpContextAccessor` interface, which provides access to the `HttpContextAccessor` object for the app. Now, you'll learn how to use DI to work with an `HttpContextAccessor` object.

The Bookstore app in the previous chapter uses a `Cart` object that accepts an `HttpContext` object in its constructor so it can work with session state and cookies. Then, the `Cart` controller uses this `Cart` object. Since several of the `Cart` controller's action methods use a `Cart` object, it would make sense to create a `Cart` object in the constructor of the controller.

However, because the `HttpContext` property of a controller is null when the constructor runs, the constructor can't create a `Cart` object. As a result, each action method that needs a `Cart` object has to create its own `Cart` object and call its `Load()` method. To reduce code duplication, the controller does that in a private helper method named `GetCart()` that the action methods call. Still, this isn't optimal.

One way to fix this issue is to change the `Cart` controller so it gets an `HttpContextAccessor` object via DI. This allows you to move the code that creates a `Cart` object from the action methods to the constructor, which reduces code duplication.

Figure 14-3 shows how this works. The first example shows a `Cart` class that accepts an `IHttpContextAccessor` object in its constructor. This makes it possible to pass this argument by dependency injection.

Within the constructor, the code initializes the private variables that hold session and cookie objects. To do that, it uses the `HttpContext` property of the `HttpContextAccessor` object.

The second example shows a `Cart` controller class that injects an `HttpContextAccessor` object. To do that, this constructor specifies a second parameter in its constructor. The first parameter specifies the `IRepository<Book>` type, and the second specifies the `IHttpContextAccessor` type. When this constructor is called, the DI provider consults its dependency mappings and passes a `Repository<Book>` object and an `HttpContextAccessor` object to the controller's constructor.

Within the constructor, the first two statements assign the objects injected by DI to private properties. Then, the third statement uses the `HttpContextAccessor` property to create a new `Cart` object and assign it to a private property. Finally, the fourth statement calls the `Load()` method of the `Cart` object. For this to work, the `Load()` method is updated to accept an `IRepository<Books>` argument as shown in the first example.

At this point, the `Cart` object is ready to be used by the action methods of the controller. Because of that, those action methods no longer need to create and load their own `Cart` objects. However, any action method that previously used the `HttpContext` property of the controller, like the `Index()` action method shown here, must be updated to use the `HttpContext` property of the `HttpContextAccessor` property instead. This has the added benefit of making action methods that use the `HttpContext` property easier to test.

## A Cart class that injects an HttpContextAccessor object

```
...
using Microsoft.AspNetCore.Http;
...
public class Cart {
    ...
    private ISession session { get; set; }
    private IRequestCookieCollection requestCookies { get; set; }
    private IResponseCookies responseCookies { get; set; }

    public Cart(IHttpContextAccessor ctx) {
        session = ctx.HttpContext.Session;
        requestCookies = ctx.HttpContext.Request.Cookies;
        responseCookies = ctx.HttpContext.Response.Cookies;
    }
    ...
    public void Load(IRepository<Book> data) {
        // code that uses the repository and HttpContext to load cart items
    }
    ...
}
```

## A Cart controller that injects an HttpContextAccessor object

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Http;
using Bookstore.Models;
...
public class CartController : Controller {
    private IRepository<Book> data { get; set; }
    private IHttpContextAccessor accessor { get; set; }
    private Cart cart { get; set; }

    public CartController(IRepository<Book> rep, IHttpContextAccessor http)
    {
        data = rep;
        accessor = http;           // store IHttpContextAccessor object
        cart = new Cart(accessor); // create model-level Cart object
        cart.Load(data);
    }

    public ViewResult Index()
    {
        var builder = new BooksGridBuilder(accessor.HttpContext.Session);
        // rest of action method code
    }
    ...
    [HttpPost]
    public RedirectToActionResult Edit(CartItem item) {
        cart.Edit(item);          // no need to create its own Cart object
        cart.Save();
        // rest of action method code
    }
    ...
}
```

---

Figure 14-3 How to use DI with an HttpContextAccessor object

## How to use DI with action methods

---

When working with DI, you typically want to use dependency injection with the constructor of a controller as shown in the previous two figures. Sometimes, though, it makes sense to inject a dependency directly into an action method. For example, you may want to do this when only one action method in a controller needs that dependency.

For example, the code at the top of figure 14-4 presents the Validation controller that the Bookstore app from the previous chapter uses to perform remote validation. This controller has two action methods that check whether a value is already in the database. To do that, these methods use Repository objects. However, each action method needs a different Repository object. The CheckGenre() action method needs a Repository<Genre> object, and the CheckAuthor() action method needs a Repository<Author> object.

To accomplish this, the ValidationController class has a private property for each Repository. Then, the constructor uses the BookstoreContext object it receives to create each of the Repository objects. However, each action method only needs one of the Repository objects. But, each time one of these action methods runs, the other Repository object is also created and never used. Obviously, this is not optimal.

You could update this controller to accept the two Repository objects via dependency injection, which would fix the problem of being tightly coupled with the BookstoreContext class. However, you'd still create two Repository objects when you only need one.

In a case like this, it's better to inject the required Repository object directly into the action method. To do that, you add an interface parameter to the action method and decorate it with the FromServices attribute. This tells MVC to use its DI mappings to get the value for this parameter.

The second code example in this figure shows how this works. Here, the Validation controller has been updated to use action method injection. This means it no longer needs a constructor or private properties for the repositories. Instead, each action method directly accepts the Repository<T> object it needs from MVC's DI mappings.

## A controller that doesn't use DI

```
using Microsoft.AspNetCore.Mvc;
using Bookstore.Models;
...
public class ValidationController : Controller
{
    private Repository<Author> authorData { get; set; }
    private Repository<Genre> genreData { get; set; }

    public ValidationController(BookstoreContext ctx) {
        authorData = new Repository<Author>(ctx);
        genreData = new Repository<Genre>(ctx);
    }

    public JsonResult CheckGenre(string genreId) {
        validate.CheckGenre(genreId, genreData);
        ...
    }

    public JsonResult CheckAuthor(string firstName, string lastname,
        string operation) {
        validate.CheckAuthor(firstName, lastName, operation, authorData);
        ...
    }
}
```

## The same controller with DI in its action methods

```
using Microsoft.AspNetCore.Mvc;
using Bookstore.Models;
...
public class ValidationController : Controller
{
    // private properties and constructor no longer needed

    public JsonResult CheckGenre(
        string genreId, [FromServices] IRepository<Genre> data) {
        validate.CheckGenre(genreId, data);
        ...
    }

    public JsonResult CheckAuthor(string firstName, string lastName,
        string operation, [FromServices] IRepository<Author> data) {
        validate.CheckAuthor(firstName, lastName, operation, data);
        ...
    }
}
```

## Description

- To inject an object into an action method, you can use the `FromServices` attribute.
- For this to work, the `CheckGenre()` and `CheckAuthor()` methods define their repository parameter as an interface type, not a class type.

---

Figure 14-4 How to use DI with action methods

## How to use DI with views

---

If necessary, you can use dependency injection with views. Most of the time, you don't want to do that because a view should get what it needs from the controller. However, when a layout for a view has a dependency, there's no controller to pass the required object to the layout. In that case, you may want to inject the object into the layout.

The first code example in figure 14-5 shows a Razor code block in a layout that uses the Cart class from the Bookstore app in the previous chapter. Here, the code block creates a new Cart object and passes it the Context property of the layout, which is of the `HttpContext` type. For this to work, the constructor of the Cart class must accept an `HttpContext` object as described in the previous chapter, not an `HttpContextAccessor` object as described in this chapter.

The second example shows a navigation link in the layout that uses the Cart object to get and display the number of items in the cart. To do that, this code just uses a Razor expression to call the `Count` property of the Cart object.

Another approach is to inject a Cart object that has a constructor that accepts an `HttpContextAccessor` as shown in this chapter. For this to work, you need to create an interface for the Cart class.

The easiest way to generate an interface for an existing class is to use Visual Studio's refactoring feature. To do that, open the C# file that contains the class and right-click the class name. In the context menu, select the Quick Actions and Refactoring item, and select the Extract Interface item. This generates the interface and adds code to the class declaration that implements the new interface.

The third example shows the declarations for the `ICart` interface and the Cart class that implements the interface. After you've created the interface and updated the Cart class to implement it, you need to map that dependency in the `Startup.cs` file, just like you learned in figure 14-1. This is also shown in the third example.

After you've set up the Cart object for DI, you can use the `@inject` directive to inject the Cart object into the layout as shown in the fourth example. This `@inject` directive specifies a variable named `cart` of the `ICart` type. When MVC encounters the `@inject` directive, it consults its mappings and injects an object created from the class that's mapped to the interface. In this case, the `ICart` interface is mapped to the Cart class, so MVC injects a Cart object. As before, MVC uses dependency chaining to make sure it also injects the `HttpContextAccessor` object that the Cart object needs.

As a result of these changes, you no longer need to create a Cart object in the Razor code block at the beginning of the layout. Instead, the navigation link can use the injected Cart object named `cart`. Because of that, this code works the same as before.

## Code in a layout that doesn't inject a Cart object

```
@{  
    var cart = new Cart(Context);  
    ...  
}
```

## A navigation link that uses the Cart object

```
<a class="nav-link" asp-action="Index" asp-controller="Cart" asp-area="">  
    <span class="fas fa-shopping-cart"></span>&ampnbspCart  
    <span class="badge badge-light">@cart.Count</span>  
</a>
```

## How to set up the Cart object for DI

### The ICart interface

```
public interface ICart {  
    // declarations for all properties and methods of the Cart class  
}
```

### The Cart class updated to implement the interface

```
public class Cart : ICart {  
    // properties and methods that now implement the interface  
}
```

### The dependency mapping in the Startup.cs file

```
public void ConfigureServices(IServiceCollection services) {  
    ...  
    // other dependency mappings  
    services.AddTransient<ICart, Cart>();  
    ...  
}
```

## Code in a layout that injects a Cart object

```
@inject ICart cart
```

## Description

- You can use the @inject directive to inject an object into a view. To do that, the data type for the object is the interface type, not the class type.

---

Figure 14-5 How to use DI with views

## How to get started with unit testing

So far, you've learned how to manually test a project by running it and entering valid and invalid input data to be sure that it works in every case. Although this works for simple apps, testing becomes more difficult and tedious as your apps become larger and more complex. Instead of testing these apps manually, you can automate testing with a process called unit testing. This process helps you improve the quality of your apps and leads to many other benefits. As mentioned earlier, using dependency injection makes it easier to automate testing with unit tests.

### How unit tests work

As figure 14-6 shows, *unit testing* provides a way of isolating individual units of code (usually methods) and verifying that they work as expected. Typically, you need to write a series of unit tests for each method to be tested so you can test every possible outcome of the method. Each unit test is a method that calls the method being tested and determines if the return value matches what you expect.

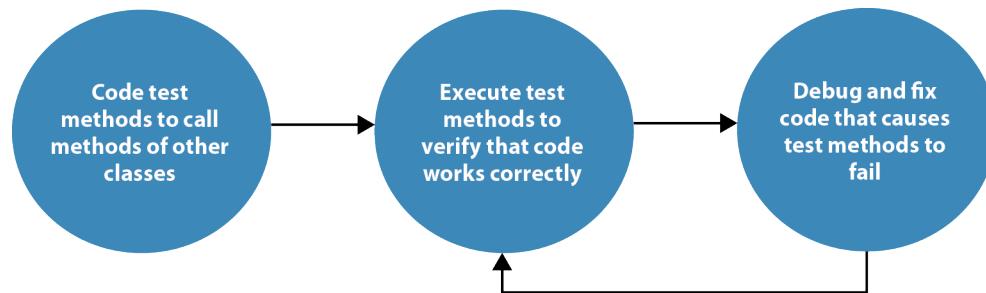
Many unit testing frameworks are available, and some of them integrate seamlessly with Visual Studio. This chapter presents the xUnit framework because it's included with Visual Studio, and it's one of the most popular unit testing frameworks for ASP.NET Core apps. However, other unit testing frameworks such as MSTest and NUnit work much the same way.

Each unit test is a method in a unit test project that you typically include in your Visual Studio solution. Typically, each Visual Studio solution has only one unit test project, but this project may contain one or more classes. Each class in the unit test project tests a different class elsewhere in the Visual Studio solution. You can write unit tests for model classes, utility classes, and controller classes.

Unit tests should be executed often. For example, you can execute them every time you make a change to your code. To do that, you can use the unit testing framework to automatically re-execute the unit tests. This allows you to quickly identify any defects that may be introduced in your code. In this way, unit testing helps you find problems earlier in the development cycle than would be possible by manually testing all your methods.

Unit testing can help you save development time. You save time on data entry because you don't need to load the app and manually enter data into your app every time you make a code change. Since the unit testing framework lets you know when a test fails, you also save time because only the most recent code changes need to be debugged, instead of having to debug the entire app.

## The unit testing process



## Advantages of unit testing

- Unit testing reduces the amount of time you have to spend manually testing an app by running it and entering data.
- Unit testing makes it easy to test an app after each significant code change. This helps you find problems earlier in the development cycle than you typically would when using manual testing.
- Unit testing makes debugging easier because you typically test an app after each significant code change. As a result, when a test fails, you only need to debug the most recent code changes.

## Description

- *Unit testing* provides a way to write code that automatically tests individual methods, called *units*, to verify that they work properly in isolation.
- You code unit tests within a separate project that's part of the solution that contains the classes to be tested.
- Visual Studio provides for creating three types of test projects for .NET Core apps: xUnit, MSTest, and NUnit. This chapter shows how to use xUnit.

---

Figure 14-6 How unit tests work

## How to add an xUnit project to a solution

---

To get started with unit testing, you typically add a new xUnit project to the solution that contains the project with the methods you want to test. To do this, you can follow the steps presented in figure 14-7. In short, you display the Add New Project dialog and use it to select the xUnit Test Project (.NET Core) template for C#. Then, you enter a name for your unit test project and specify a location where the files for the project will be stored.

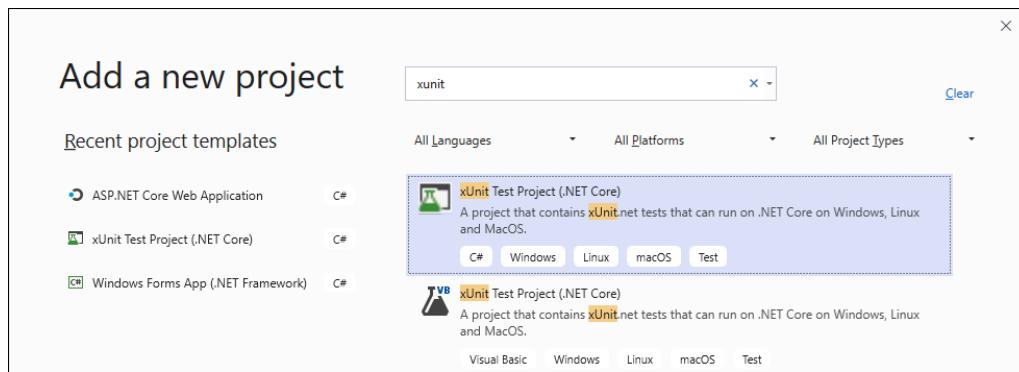
Although you can give your unit test project any name you want, unit tests are usually created to mirror the code being tested. As a result, it's considered a best practice to give the unit test project a name that's related to the project it is testing. For example, if you are testing a project named Bookstore, you can name your unit test project Bookstore.Tests. It's also considered a best practice to store your unit test project in the same solution as the project that you're testing, although that isn't required.

After you create the unit test project, you must add a reference to the project that you're testing as described in this figure. Otherwise, the code in your unit test project won't be able to access the code in that project.

When you create an xUnit project, Visual Studio adds a single class named UnitTest1 by default. Because this isn't a very meaningful name, you'll want to change it to something that is more meaningful. For example, if the class will be used to test the methods in a class named Nav, you might rename this class to NavTests.

Later, when you need to add other classes to store your unit tests, you can add them by right-clicking on the unit test project and selecting the Add ➔ Class item. For example, you can add a class named AuthorControllerTests that contains the unit tests for the Author controller.

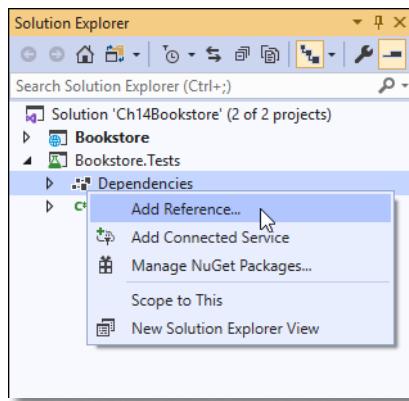
## The Add New Project dialog



## How to add a unit test project to a solution

1. Right-click the solution and select Add→New Project.
2. In the Add New Project dialog, select the xUnit Test Project (.NET Core) template and click Next. To help find the template, you can use the dialog to search for “xunit”.
3. Enter a name and location for the project and click Create.

## The web app project and the unit test project in the Solution Explorer



## Description

- By convention, a unit test project is named after the web app project it's testing.
- For the unit test project to work correctly, it must include a reference to the web app project. To add a reference, right-click Dependencies and select Add Reference. In the resulting dialog, click Projects, select the web app project, and click OK.
- To add new classes to a unit test project, right-click the project and select Add→Class.

---

Figure 14-7 How to add an xUnit project to a solution

## How to write a unit test

---

After you create a unit test class, you can start writing your unit tests. Figure 14-8 presents the basic skills for doing that.

To start, the first table in this figure presents some of the static methods of the Assert class that you can use to test various conditions. Then, the second table presents three attributes that you can use to decorate your test methods. Because the Assert class and the attributes that are used with unit tests are included in the Xunit namespace, you'll typically include a using directive for this namespace in each unit test class as shown in the first code example.

The second code example below the tables presents a test class named NavTests with two test methods. These methods test the Active() method in the Nav class of the Bookstore app. Although it's not shown here, you should know that in addition to the using directive for the Xunit namespace, the file that contains the test class also includes a using directive for the namespace that contains the class that you are testing.

When you code your test methods, it's good to use a consistent naming convention. In this case, both test method names consist of the name of the method to be tested, an underscore, and a description of the behavior the test expects. Names like these can help you keep your unit tests organized.

It's also common to use the *Arrange/Act/Assert (AAA) pattern* for each unit test you write. The code in the Arrange section initializes any arguments needed by the method being tested. Then, the code in the Act section calls the method being tested. Finally, the code in the Assert section checks whether the method being tested behaved as expected.

In this figure, the first test method is decorated with the Fact attribute. As a result, Visual Studio executes this test method when the unit tests run. In the Arrange section, the code creates two string variables. In the Act section, the code passes these variables to the static Active() method of the Nav class and assigns the return value to a variable name result. Finally, in the Assert section, the code uses the static IsType() method of the Assert class to check that the return value is of the string type. If so, the test passes.

The second test method is decorated with the Theory attribute. This tells Visual Studio that this test method has parameters. When you use a test method that has parameters, you must also use theInlineData attribute to provide the values for those parameters. Each attribute represents a separate test. So, this test method runs twice, once with the values "Home" and "Home", and once with the values "Books" and "Books". The Assert section of this test uses the Equal() method of the Assert class to test that it returns the string "active" when the values passed to the Active() method match.

When you write unit tests for a method, you should try to cover as many code paths as possible. For example, in addition to testing that the Active() method returns what you expect when the values match, you should add a test for when the values don't match. Although a unit test like this isn't presented in this figure, this unit test is included in the downloadable app for this chapter.

## Some static methods of the Assert class

Method	Description
<code>Equal(expected, result)</code>	Tests whether the specified objects are equal.
<code>NotEqual(expected, result)</code>	Tests whether the specified objects are not equal.
<code>False(Boolean)</code>	Tests whether the specified condition is false.
<code>True(Boolean)</code>	Tests whether the specified condition is true.
<code>IsType&lt;T&gt;(result)</code>	Tests whether the specified object is of the specified type.
<code>IsNull(result)</code>	Tests whether the specified object is null.

## Three attributes of the Xunit namespace

Attribute	Description
<code>Fact</code>	Identifies a test method.
<code>Theory</code>	Identifies a test method that has parameters.
<code>InlineData(p1, p2, ...)</code>	Provides parameter values to test.

## The using directive for the Xunit namespace

```
using Xunit;
```

## A test class with two test methods

```
public class NavTests
{
    [Fact]
    public void ActiveMethod_ReturnsAString()
    {
        string s1 = "Home";           // arrange
        string s2 = "Books";

        var result = Nav.Active(s1, s2); // act

        Assert.IsType<string>(result); // assert
    }

    [Theory]
    [InlineData("Home", "Home")]
    [InlineData("Books", "Books")]
    public void ActiveMethod_ReturnsValueActiveIfMatch(string s1,
        string s2)
    {
        string expected = "active"; // arrange

        var result = Nav.Active(s1, s2); // act

        Assert.Equal(expected, result); // assert
    }
}
```

## Description

- Code in a unit test is often organized using the *Arrange/Act/Assert (AAA) pattern*.

---

Figure 14-8 How to write a unit test

## How to run a unit test

---

After you have written your unit tests, you can use the Visual Studio Test Explorer shown in figure 14-9 to run them. To open this window, use one of the techniques described in this figure.

The Test Explorer displays the name of your test project, each test class within the project, and each test method within each test class. In addition, for parameterized test methods, it displays a line for each `InlineData` attribute. In other words, it shows you how many times the parameterized test method runs, and the data that's passed to it each time it runs. For example, the first screen shows that the `ActiveMethod_ReturnsValueActiveIfMatch()` method runs twice, once for each pair of string values.

If your tests don't appear in the Test Explorer, you need to build your test project. To do that, you can select the `Build→Build Solution` item from the menu system. Or, you can click the Run All button in the Test Explorer toolbar, which is the first green triangle on the far left side of the toolbar.

The Test Explorer toolbar also displays the number of total tests in the project, how many have passed, how many have failed, and how many haven't run yet. The total tests are marked with a lab test bottle, the successful tests with a green check mark, the failed tests with a red X, and the un-run tests with a blue exclamation point.

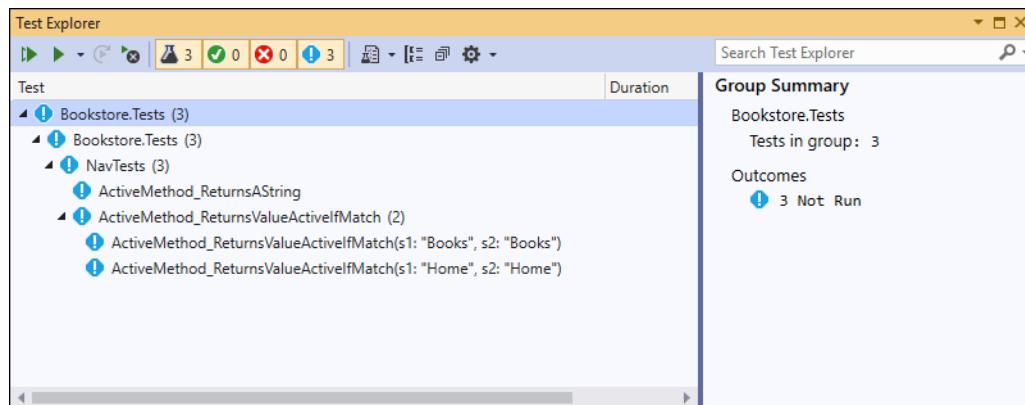
To run your tests, click the Run All button. After your tests run, the Test Explorer indicates how long it took for each unit test to complete. This can be useful because you may need to fix some code if the unit test is taking too long to run, even if the unit test passes. You can display more details of any group or individual unit test by clicking on it in the Test Explorer. This displays details about the item on the right side of the Test Explorer.

Another way to run your tests is to click the Run button, which is the second green triangle on the left side of the Test Explorer toolbar. You can click this button's down arrow to display the menu shown in the second screen. From this menu, you can run your tests in several different ways: you can run all the tests, only the selected tests, only the tests that have failed, or only the tests that haven't been run yet. You can also set a breakpoint in one or more unit tests and debug them just as you would debug a regular app. Stepping through your code with the Visual Studio debugger takes you seamlessly back and forth between the unit tests and the methods that you are testing.

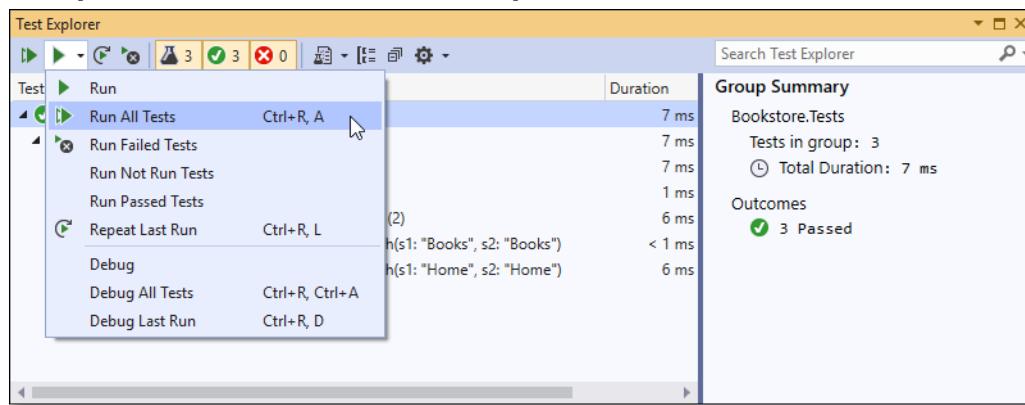
If one or more unit tests fail, you know that there's either a problem in the method that you're testing or a problem with the unit test. In that case, you need to debug your code until you can get your unit tests to pass.

Once you have your unit tests set up, you should run them often. In fact, many programmers like to run their unit tests after every significant code change. If you want, you can configure xUnit to automatically run the unit tests after each build by clicking on the Settings button in the Test Explorer toolbar and selecting the Run Tests After Build item.

## The Test Explorer in Visual Studio



## Some options to run tests in Test Explorer



## Two ways to open the Test Explorer

- From the menu system, select Test→Test Explorer.
- In the Solution Explorer, right-click on the test class and select Run Tests.

## Description

- The Test Explorer shows whether or not tests have been run, as well as which tests passed or failed. It also shows how long it took to execute each unit test.
- The Test Explorer marks tests that succeed with a green check mark, tests that fail with a red X, and tests that have not run yet with a blue exclamation point.
- You have many options to run tests from the Test Explorer. You can also set breakpoints and debug unit tests the same way you would debug regular methods.

Figure 14-9 How to run a unit test

## How to test methods that have dependencies

So far, you've learned how to test methods that accept simple arguments. For example, the Active() method described earlier accepts two strings. In that case, the test just hardcoded two strings and passed them to that method. However, many methods have more complex dependencies, and these dependencies themselves can fail.

For example, suppose you want to test an action method of a controller class whose constructor accepts an IRepository<Book> object. Then, if you pass a Repository<Book> object to the constructor and the test fails, it may be difficult to determine why. Although it could be an error in the action method code, it could also be a problem with the database or with the network that communicates with the database.

To test just the code in an action method, you can use a fake, or *mock*, version of the dependency. That makes it easier to pinpoint the problem if the test fails.

### How to use a fake repository object

Figure 14-10 starts by presenting a controller that depends on a repository class. Specifically, it presents a Home controller that depends on an IRepository<Book> object that it stores in a private property named data. Then, its Index() action method returns a Book object by calling the Get() method of the IRepository<Book> interface and passing it a QueryOptions<Book> object.

To test the Index() action method of this controller, you can code a new implementation of the IRepository<Book> interface. Then, you can code its Get() method so it returns a Book object without accessing the database. That's what the FakeBookRepository class shown in the second example does.

An easy way to code an implementation of an interface is to use Visual Studio's code generation feature. This generates a stub for each property and method of the interface that throws a NotImplementedException object. Then, you implement the properties and methods used by the unit tests so they don't throw an exception. Here, the second example implements the Get() method so it returns a Book object. This is the only method needed by the unit test in the third example.

This test method creates an object from the FakeBookRepository class and passes it to the constructor of the Home controller. Then, it calls the controller's Index() action method and tests that its return value is of the ViewResult type. If so, the test passes.

## A controller that depends on a repository class

```
public class HomeController : Controller
{
    private IRepository<Book> data { get; set; }
    public HomeController(IRepository<Book> rep) => data = rep;

    public ViewResult Index()
    {
        var random = data.Get(new QueryOptions<Book> {
            OrderBy = b => Guid.NewGuid()
        });
        return View(random);
    }
}
```

## A fake repository class that implements the Get() method

```
using Bookstore.Models;
...
public class FakeBookRepository : IRepository<Book>
{
    public int Count => throw new NotImplementedException();
    public void Delete(Book entity) => throw new NotImplementedException();
    public Book Get(QueryOptions<Book> options) => new Book();
    ...
}
```

## A unit test that passes an instance of the fake repository to the controller

```
[Fact]
public void IndexActionMethod_ReturnsAViewResult()
{
    // arrange
    var rep = new FakeBookRepository();
    var controller = new HomeController(rep);

    // act
    var result = controller.Index();

    // assert
    Assert.IsType<ViewResult>(result);
}
```

## Description

- To test a method that depends on a repository class, you can create a fake repository class that implements the interface for the repository class. Then, you can implement the method used by the method you're testing so it doesn't access the database.
- When you use a fake repository class to test a method, you can be sure that if the test fails, it isn't due to problems with the database or the network connection to the database.
- To use Visual Studio to generate stubs for an interface, create a new class that implements the interface. Then, hover the mouse pointer over the red squiggle that indicates that the interface hasn't been implemented, click the lightbulb icon, and select Implement Interface.

---

Figure 14-10 How to use a fake repository object

## How to use a fake TempData object

---

In the previous figure, you learned how to pass a fake version of an object to the constructor of a controller. This works because the controller uses dependency injection, and it uses interface parameters to specify the object it needs. As a result, it's easy to swap in any object that implements the correct interface.

However, controllers that use TempData don't receive a TempData object by dependency injection. Instead, they use the TempData property of the Controller class, as shown by the Edit() action method in the first example of figure 14-11. Fortunately, you can use a variation of the technique from the previous figure to handle this situation.

As before, you need to create a fake version of the object that the action method depends on. This time, though, you create a fake version of the TempData object by implementing the ITempDataDictionary interface. This interface has several properties and methods as well as an indexer. As before, if you use Visual Studio to generate a class for this interface, it generates stubs that throw exceptions. Then, you can write code for the members that the action method you're testing uses. In this figure, the Edit() action method only needs to add a value to TempData, so you only need to write code for the indexer.

The second example shows the FakeTempData class generated by Visual Studio after the indexer is implemented. This implementation doesn't actually do anything because the test only needs the Edit() action method to be able to set a TempData value without throwing an exception.

The Author controller that contains the Edit() action method accepts an IRepository<Author> object in its constructor. To test this method, you can use a FakeAuthorRepository class that works like the FakeBookRepository class from the previous figure. In this case, however, you need to implement the Update() and Save() methods of the FakeAuthorRepository class because the Edit() action method calls these methods.

When you implement a property or method that's used by a method being tested, you should do the most minimal implementation possible. This can be as simple as returning null as shown by the get accessor in the second example, or removing all the code from a method with a void return type as shown by the Update() method in the third example.

The fourth example presents a method that tests the Edit() action method. As before, it creates a fake repository object and passes it to the constructor of the controller. This time, however, it also creates a fake TempData object and assigns it to the TempData property of the controller. Then, the test method calls the controller's Edit() action method and tests that its return value is of the RedirectToActionResult type. If so, the test passes.

## An action method that accesses TempData

```
[HttpPost]
public IActionResult Edit(Author author)
{
    if (ModelState.IsValid) {
        data.Update(author);
        data.Save();
        TempData["message"] = $"{author.FullName} updated.";
        return RedirectToAction("Index");
    }
    else {
        return View("Author", author);
    }
}
```

## The FakeTempData class with an indexer that does nothing

```
using Microsoft.AspNetCore.Mvc.ViewFeatures; // for ITempDataDictionary
...
public class FakeTempData : ITempDataDictionary
{
    public object this[string key] { get => null; set { } }
    public ICollection<string> Keys => throw new NotImplementedException();
    ...
}
```

## Two methods of the FakeAuthorRepository class that do nothing

```
public void Update(Author entity) { } // does nothing
public void Save() { } // does nothing
```

## A test method that tests the action method that uses TempData

```
[Fact]
public void Edit_POST_ReturnsRedirectToActionResultIfModelStateIsValid() {
    // arrange
    var rep = new FakeAuthorRepository();
    var controller = new AuthorController(rep) {
        TempData = new FakeTempData()
    };

    // act
    var result = controller.Edit(new Author());

    // assert
    Assert.IsType<RedirectToActionResult>(result);
}
```

## Description

- To test an action method that accesses TempData, you must create a fake TempData object. Otherwise, the test will fail due to a NullReferenceException.
- To create a fake TempData class, create a new class that implements the ITempDataDictionary interface from the Microsoft.AspNetCore.Mvc.ViewFeatures namespace.

---

Figure 14-11 How to use a fake TempData object

## How to create fake objects with Moq

In the last two figures, you learned how to create mock objects to test a method. That way, your unit tests only test the code in the method being tested, not the code that accesses a database or works with TempData.

As you've seen, you can use the code generation features of Visual Studio to create the mock objects you need. However, that can lead to problems setting up your mock objects for each individual test. For instance, the test method in figure 14-10 needs the implementation of the Get() method of the FakeBookRepository class shown in that figure. If you had tests that needed different implementations of that method, though, you might need to generate different versions of that class. As you can imagine, that could get messy fast.

To solve this issue, many programmers prefer to use a third-party tool to create mock objects. That's why the next few figures show how to use a popular mocking tool called Moq.

## How to work with mock objects

Before you can use the Moq framework, you need to add its NuGet package to your test project as described in figure 14-12. Then, you can use the Mock<T> and It classes to create mock objects. The two tables present some of the properties and methods of these classes, and the code examples below the tables show how to use them.

The first code example presents the using directive for the Moq namespace. Then, the second example shows how to create an object from the generic Mock<T> class. To do that, this example specifies the IRepository<Author> type as the data type to mock.

The third code example shows how to implement a method of the mock object. To do this, the code uses the Setup() method of the Mock<T> class. The argument passed to the Setup() method is a lambda expression that identifies the method to implement. In this case, the code implements the Get() method, but you can use the same technique for other methods.

If the method you're implementing accepts arguments, you configure the arguments with the static methods of the It class. Here, the Get() method is configured to accept any int. This means that your test method can pass any int value to the method that you're testing. You can also specify acceptable values as shown by the fourth example.

If the method you're implementing doesn't return any data, you don't need to do anything else. If, however, the method returns data, you need to tell the mock object what to return. To do this, you use the Returns() method of the Mock<T> class. In this figure, the code in the third and fourth examples returns a new Author object because the Get() method being mocked should return an Author object.

The fifth example shows how to pass a mock object to a controller. To do that, you just use the Object property of the Mock<T> class to access the mock object. Here, the code passes an IRepository<Author> object to the Author controller.

## How to add the Moq framework to your test project

1. In the Solution Explorer, right-click the test project and select Manage NuGet Packages from the resulting menu.
2. In the NuGet Package Manager, click Browse, search for “moq”, select the Moq package, and click Install.
3. In the resulting dialogs, click OK and I Accept.

## Two methods and one property of the Mock<T> class

Method	Description
<b>Setup</b> (lambda)	Lambda expression identifies the method to mock.
<b>Returns</b> (value)	Identifies the return value of the method identified by Setup().
Property	Description
<b>Object</b>	Returns the instance of the fake object.

## Two static methods of the It class

Method	Description
<b>IsAny&lt;T&gt;()</b>	Identifies an argument to be passed to the method under test.
<b>Is&lt;T&gt;(lambda)</b>	Identifies and further specifies an argument to be passed to the method under test.

## The using directive for the Moq namespace

```
using Moq;
```

## Code that creates a Mock< IRepository<Author>> object

```
var rep = new Mock< IRepository<Author>>();
```

## Code that sets up Get() to accept any int and return an Author object

```
rep.Setup(m => m.Get(It.IsAny<int>())).Returns(new Author());
```

## The same code statement adjusted to accept any int greater than zero

```
rep.Setup(m => m.Get(It.IsAny<int>(i => i > 0))).Returns(new Author());
```

## Code that passes the mock object to a controller

```
var controller = new AuthorController(rep.Object);
```

## Description

- It can be time consuming to create your own fake objects. Because of that, many developers prefer to use a mocking tool instead. One popular mocking tool for .NET is the Moq framework.

---

Figure 14-12 How to work with mock objects

## How to mock a repository object

---

Figure 14-10 showed how to code a fake repository class that you can use to test an action method that depends on a repository object. Now, the first example in figure 14-13 shows how to perform the same task using Moq.

Within the test method, the Arrange section starts by creating a `Mock< IRepository< Book >>` object. Then, it passes that mock repository object to the constructor of the Home controller.

After the Arrange section, the Act section calls the `Index()` action method of the Home controller and stores the return value in a variable named `result`. Then, the Assert section checks whether the result is of the `ViewResult` type. If so, the test passes.

## How to mock a TempData object

---

Figure 14-11 showed how to code a fake TempData class that you can use to test an action method that depends on a TempData object. Now, the second example in figure 14-13 shows how to perform the same task using Moq.

Just like the fake TempData class, the class that uses Moq includes a using directive for the `Microsoft.AspNetCore.Mvc.ViewFeatures` namespace that contains the `ITempDataDictionary` interface. Then, within the test method, the Arrange section begins by creating a `Mock< IRepository< Author >>` object and a `Mock< ITempDataDictionary >` object. Next, it creates an instance of the controller that contains the action method that's being tested. To do that, it passes the mock repository object to the controller's constructor and assigns the mock TempData object to the controller's `TempData` property.

With Moq, you don't need to explicitly set up the `Update()` and `Save()` methods of the repository. Similarly, you don't need to set up the `TempData` indexer. This is one of the benefits of using the Moq framework.

The Act section calls the `Edit()` action method and stores its result. Then, the Assert section checks whether the result is of the `RedirectToActionResult` type. If so, the test passes. This makes sure the action method returns a `RedirectToActionResult` object when the controller's `ModelState` property is valid.

Of course, to test all code paths, you should also code a test method that makes sure the action method returns a `ViewResult` object when the controller's `ModelState` property is invalid. A test method like this is shown later in this chapter. In addition, one is included in the downloadable app for this chapter. The key to this test method is that it includes a line of code that adds an error to the model like this:

```
controller.ModelState.AddModelError("", "Test error message.");
```

### A test method that mocks a repository object

```
using Bookstore.Controllers; // for HomeController  
...  
[Fact]  
public void IndexActionMethod_ReturnsViewResult()  
{  
    // arrange  
    var rep = new Mock< IRepository<Book>>();  
    var controller = new HomeController(rep.Object);  
  
    // act  
    var result = controller.Index();  
  
    // assert  
    Assert.IsType< ViewResult >(result);  
}
```

### A test method that mocks a TempData object

```
using Bookstore.Areas.Admin.Controllers; // for Admin.AuthorController  
using Microsoft.AspNetCore.Mvc.ViewFeatures; // for ITempDataDictionary  
...  
[Fact]  
public void Edit_POST_ReturnsRedirectToActionResultIfModelStateIsValid()  
{  
    // arrange  
    var rep = new Mock< IRepository< Author >>();  
    var temp = new Mock< ITempDataDictionary >();  
    var controller = new AuthorController(rep.Object)  
    {  
        TempData = temp.Object  
    };  
  
    // act  
    var result = controller.Edit(new Author());  
  
    // assert  
    Assert.IsType< RedirectToActionResult >(result);  
}
```

### Description

- You can use the Moq framework to mock repository and TempData objects.
- To mock a TempData object, use the ITempDataDictionary interface from the Microsoft.AspNetCore.Mvc.ViewFeatures namespace as the type argument.
- When you use Moq, you don't need to manually implement indexers, properties, or most void methods of the repository or TempData interfaces.

---

Figure 14-13 How to mock repository and TempData objects

## How to mock an IHttpContextAccessor object

The past few figures have shown how to use Moq for a relatively simple setup. However, Moq can also help you with a more complex setup.

To illustrate a more complex setup, the first example in figure 14-14 shows a constructor for a Cart class that accepts an IHttpContextAccessor object. Then, in the body of the constructor, the first three statements use the HttpContext property of that interface to get objects needed to work with session state and cookies. As a result, to test the Cart class, you need to set up all these properties.

This constructor ends with a statement that initializes the private items variable with a list of CartItem objects. The web app project doesn't need this statement to run properly. However, the test project does need this statement. This shows that it's common to need to refactor code that wasn't built with testing in mind so you can test it. So, if you find a class or method hard to test, you may need to refactor it so it's easier to test.

The second example presents the Subtotal property of the Cart class that's tested by the test method shown in the third example. This Subtotal property uses the Sum() method that's available from LINQ to sum the values of the Subtotal property of every item in the cart.

The third example shows the test method that checks whether the Subtotal property returns a double value. The class that contains it includes a using directive for the Microsoft.AspNetCore.Http namespace that contains the IHttpContextAccessor interface.

Within the test method, the Arrange section starts by creating a new Mock<HttpContextAccessor> object. Then, it creates a new DefaultHttpContext object. This object allows you to create an empty HttpContext object for testing.

Next, the Arrange section uses the DefaultHttpContext object and the Setup() method of the Mock<T> class to implement most of the properties that the Cart constructor needs. To implement the Session property, though, it needs to create a Mock<ISession> object and pass it to the Returns() method.

Finally, the Arrange section creates a new Cart object and passes the mock HttpContextAccessor object to its constructor. Then, it uses the Add() method of the Cart class to add an item to the cart. This is necessary because the Cart's items collection is private, so you can't populate it directly.

After the Arrange section, the Act section gets the value that's returned by the Subtotal property. Then, the Assert section checks that this value is of the double type. If so, the test passes.

## The constructor of the Cart class

```
public Cart(IHttpContextAccessor ctx)
{
    // assign private variables
    session = ctx.HttpContext.Session;
    requestCookies = ctx.HttpContext.Request.Cookies;
    responseCookies = ctx.HttpContext.Response.Cookies;
    items = new List<CartItem>(); // needed for test method to run
}
```

## The Subtotal property of the Cart class

```
public double Subtotal => items.Sum(i => i.Subtotal);
```

## A test method that tests the Subtotal property of the Cart with Moq

```
...
using Microsoft.AspNetCore.Http;      // for IHttpContextAccessor
...
[Fact]
public void SubtotalProperty_ReturnsADouble()
{
    // arrange
    var accessor = new Mock<IHttpContextAccessor>();
    var context = new DefaultHttpContext();

    accessor.Setup(m => m.HttpContext).Returns(context);
    accessor.Setup(m => m.HttpContext.Request).Returns(context.Request);
    accessor.Setup(m => m.HttpContext.Response).Returns(context.Response);
    accessor.Setup(m => m.HttpContext.Request.Cookies)
        .Returns(context.Request.Cookies);
    accessor.Setup(m => m.HttpContext.Response.Cookies)
        .Returns(context.Response.Cookies);

    var session = new Mock<ISession>();
    accessor.Setup(m => m.HttpContext.Session).Returns(session.Object);

    Cart cart = new Cart(accessor.Object);
    cart.Add(new CartItem { Book = new BookDTO() });

    // act
    var result = cart.Subtotal;

    // assert
    Assert.IsType<double>(result);
}
```

## Description

- You can use the Moq framework to mock an HttpContextAccessor object. To do that, use the IHttpContextAccessor interface of the Microsoft.AspNetCore.Http namespace as the type argument for the Mock constructor.
- The DefaultHttpContext class creates an empty instance of the HttpContext class that you can use to set up the HttpContextAccessor properties and methods you need.

---

Figure 14-14 How to mock an HttpContextAccessor object

## The Bookstore.Tests project

The next few figures present some of the tests in the test project for the Bookstore app. These tests are similar to the ones you've already seen, although they build upon the concepts that have been presented so far.

### The Test Explorer

Figure 14-15 shows the Test Explorer for the Bookstore.Tests project after all test methods have passed. These methods test a few of the Bookstore web app's models and controllers. A real-world project would probably have many more test methods than this, and those test methods would test more complex behaviors. However, these tests should give you a good idea of how to get started with unit testing.

The Test Explorer shows that the Bookstore.Tests project consists of six test classes. To name these test classes, this project adds a suffix of "Tests" to the name of the class being tested. For example, the CartTests and NavTests classes contain test methods for the Cart model and the Nav model, and the HomeControllerTests and BookControllerTests classes contain test methods for the Home and Book controllers. In addition, if a controller is in the Admin area, the project adds a prefix of "Admin". For example, the AdminAuthorControllerTests and AdminBookControllerTests classes contain test methods for the Author and Book controllers that are in the Admin area.

When you use the Test Explorer, it groups each test class so the test methods they contain display together in the window. This makes it easy to run just that group of tests if you want. For example, if you only want to run the tests for the CartTests class, you can select that class and click the Run button in the Test Explorer toolbar.

The Test Explorer also displays the number of test methods in each test class. For example, the HomeControllerTests class has only two tests.

In this project, the name of each test method identifies the method or property that's being tested and the expected results of the test. For example, the HomeControllerTests class has test methods named

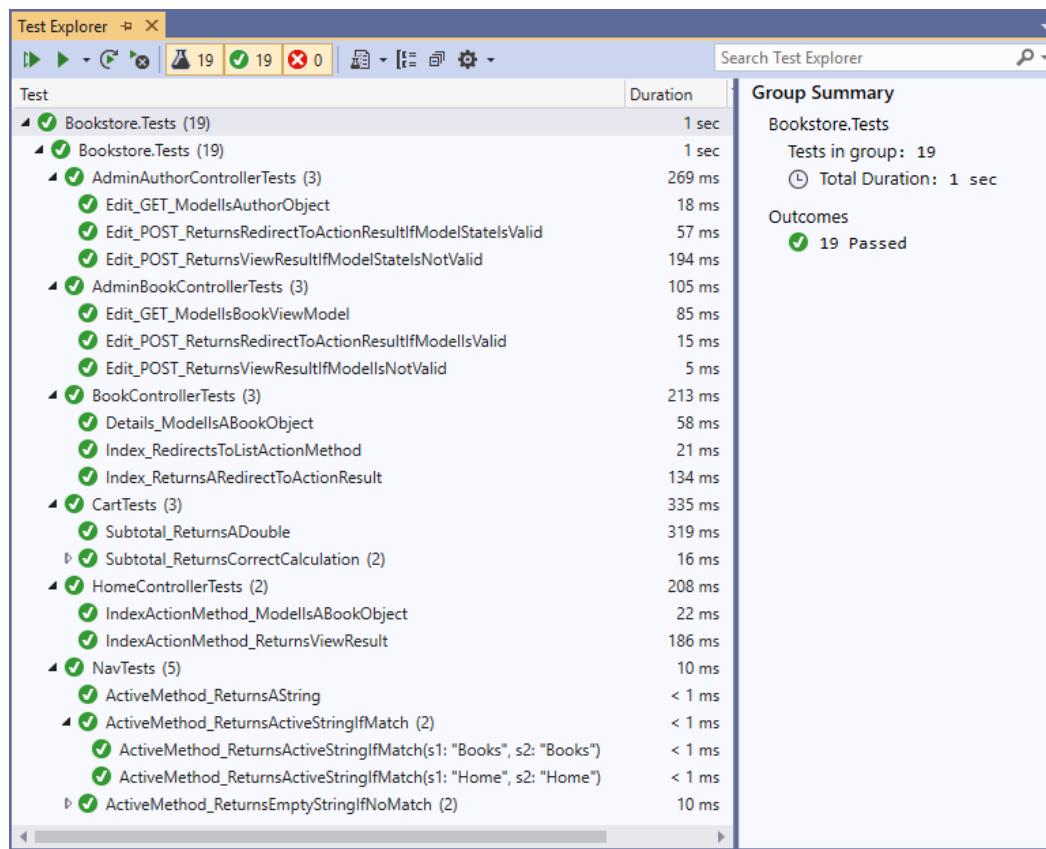
`IndexActionMethod_ModelIsABookObject()`

and

`IndexActionMethod_ReturnsViewResult()`

This naming convention helps organize the tests and makes them self-documenting. When you use this convention, it leads to test method names that are unusually long. However, if these long method names help to organize and document the tests, it's usually worth the extra effort that it takes to create them.

## The Test Explorer for the Bookstore.Tests project



### Description

- The Bookstore.Tests project tests some of the models and controllers of the Bookstore web app.
- In the Bookstore.Tests project, the names of the test classes identify the model or controller that contains the methods being tested. If a controller is in an area, the name of the controller is prefixed with the name of the area.
- In the Bookstore.Tests project, the names of the test methods identify the property or method being tested and the expected result of the test.
- The Test Explorer displays the test methods as children of a class. This makes it easy to run all tests for a class.

Figure 14-15 The Test Explorer for the Bookstore.Tests project

## The BookControllerTests class

---

Figure 14-16 presents the BookControllerTests class. To start, this class includes using directives for the xUnit and Moq namespaces. This allows the class to easily work with those tools. In addition, it includes the Models and Controllers namespaces to work with the model and controller classes of the Bookstore web app.

Within the test class, the first method tests that the Index() action method returns a RedirectToActionResult object. To do that, the Arrange section creates a mock BookstoreUnitOfWork object. Then, it creates a new Book controller by passing the mock unit of work object to the controller's constructor. Since the Index() action method only redirects, the mock unit of work object doesn't need any setup.

After the Arrange section, the Act section calls the controller's Index() action method and stores the result that's returned by the method. Then, the Assert section checks that the result is of the RedirectToActionResult type. If so, the test passes.

The second method tests that the Index() action method redirects to the List() action method. To do that, the Arrange section creates a mock unit of work object. Then, it creates a new Book controller object by passing the mock unit of work object to the controller's constructor. Again, since the Index() action method only redirects, the mock unit of work object doesn't need any setup.

After the Arrange section, the Act section calls the controller's Index() action method and stores the object that's returned in a variable named result. Then, the Assert section checks if the result's ActionName property is equal to the expected value, which is "List". If it is, the test passes.

The third method tests that the Details() action method returns a ViewResult object with a Model property of the Book type. In other words, it checks that the object passed to the View() method in the Details() action method is of the Book type.

To do that, the Arrange section creates a mock IRepository<Book> object and sets up its Get() method to accept any QueryOptions<Book> object and to return a Book object. Then, it creates a mock IBookstoreUnitOfWork object and sets it up so its Books property returns the IRepository<Book> object. Next, it creates a new Book controller by passing this mock unit of work object to its constructor.

After the Arrange section, the Act section calls the Details() action method and passes it an int value. To get the model object, the code calls the Model property of the ViewData property of the ViewResult object that's returned by the Details() action method. Since the Model property is of the object type, the code uses the as keyword to cast this object to the Book type. Then, the Assert section checks that the model object is of the Book type. If so, the test passes.

## The BookControllerTests class

```
using Microsoft.AspNetCore.Mvc;
using Xunit;
using Moq;
using Bookstore.Controllers;
using Bookstore.Models;

namespace Bookstore.Tests
{
    public class BookControllerTests
    {
        [Fact]
        public void Index_ReturnsARedirectToActionResult() {
            // arrange
            var unit = new Mock<IBookstoreUnitOfWork>();
            var controller = new BookController(unit.Object);

            // act
            var result = controller.Index();

            // assert
            Assert.IsType<RedirectToActionResult>(result);
        }

        [Fact]
        public void Index_RedirectsToListActionMethod() {
            // arrange
            var unit = new Mock<IBookstoreUnitOfWork>();
            var controller = new BookController(unit.Object);

            // act
            var result = controller.Index();

            // assert
            Assert.Equal("List", result.ActionName);
        }

        [Fact]
        public void Details_ModelIsABookObject() {
            // arrange
            var bookRep = new Mock< IRepository<Book>>();
            bookRep.Setup(m => m.Get(It.IsAny<QueryOptions<Book>>()))
                .Returns(new Book { BookAuthors = new List<BookAuthor>() });
            var unit = new Mock<IBookstoreUnitOfWork>();
            unit.Setup(m => m.Books).Returns(bookRep.Object);

            var controller = new BookController(unit.Object);

            // act
            var model = controller.Details(1).ViewData.Model as Book;

            // assert
            Assert.IsType<Book>(model);
        }
    }
}
```

---

Figure 14-16 The BookControllerTests class

## The AdminBookControllerTests class

---

Figure 14-17 presents the AdminBookControllerTests class. The using directives of this class are similar to those of the BookControllerTests class presented in the previous figure. However, these using directives include the Microsoft.AspNetCore.Mvc.ViewFeatures namespace. That makes it easy to create a mock TempData object. In addition, these using directives include the namespace for the controllers in the Admin area. That makes it easy to work with the Book controller in the Admin area.

Within the class, the first method is a helper method named GetUnitOfWork() that returns an IBookstoreUnitOfWork object. This method encapsulates the setup code necessary to mock the unit of work object for the Bookstore web app. This helps to avoid duplicating the same setup code in multiple places in your test methods.

Within the GetUnitOfWork() method, the first group of statements creates a Book repository and sets up its Get() and List() methods as well as its Count property. The second group of statements creates an Author repository and sets up its List() method. The third group of statements creates a Genre repository and sets up its List() method. The fourth group of statements creates a unit of work object for the Bookstore web app and sets up its Books, Authors, and Genres properties. Finally, the last statement returns the mock unit of work object.

The first test method checks whether the Edit() action method for GET requests passes a model of the BookViewModel type to the view. To start, the Arrange section calls the helper method to get the unit of work object. Then, it creates a Book controller by passing the unit of work object to the controller's constructor.

After the Arrange section, the Act section calls the Edit() action method of the controller, passes it an int value, gets an object for the view model, and casts that object to the BookViewModel type. Then, the Assert section checks whether this view model is of the BookViewModel type. If so, the test passes.

## The AdminBookControllerTests class

```
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.ViewFeatures;
using Xunit;
using Moq;
using Bookstore.Areas.Admin.Controllers;
using Bookstore.Models;

namespace Bookstore.Tests
{
    public class AdminBookControllerTests
    {
        public IBookstoreUnitOfWork GetUnitOfWork()
        {
            // set up Book repository
            var bookRep = new Mock<IRepository<Book>>();
            bookRep.Setup(m => m.Get(It.IsAny<QueryOptions<Book>>()))
                .Returns(new Book { BookAuthors = new List<BookAuthor>() });
            bookRep.Setup(m => m.List(It.IsAny<QueryOptions<Book>>()))
                .Returns(new List<Book>());
            bookRep.Setup(m => m.Count).Returns(0);

            // set up Author repository
            var authorRep = new Mock<IRepository<Author>>();
            authorRep.Setup(m => m.List(It.IsAny<QueryOptions<Author>>()))
                .Returns(new List<Author>());

            // set up Genre repository
            var genreRep = new Mock<IRepository<Genre>>();
            genreRep.Setup(m => m.List(It.IsAny<QueryOptions<Genre>>()))
                .Returns(new List<Genre>());

            // set up unit of work
            var unit = new Mock<IBookstoreUnitOfWork>();
            unit.Setup(m => m.Books).Returns(bookRep.Object);
            unit.Setup(m => m.Authors).Returns(authorRep.Object);
            unit.Setup(m => m.Genres).Returns(genreRep.Object);

            return unit.Object;
        }

        [Fact]
        public void Edit_GET_ModelIsBookObject()
        {
            // arrange
            var unit = GetUnitOfWork();
            var controller = new BookController(unit);

            // act
            var model = controller.Edit(1).ViewData.Model as BookViewModel;

            // assert
            Assert.IsType<BookViewModel>(model);
        }
    }
}
```

---

Figure 14-17 The AdminBookControllerTests class (part 1)

The second test method tests whether the Edit() action method for POST requests returns a ViewResult if the model state is not valid. Within this method, the Arrange section begins like the previous method. However, after creating the controller object, the Arrange section makes the model state invalid by adding an error to it. To do that, it calls the AddModelError() method of the controller's ModelState property. Then, the Arrange section finishes by creating the BookViewModel object that's required by the Edit() action method for POST requests.

After the Arrange section, the Act section calls the Edit() action method and passes it the BookViewModel object that it requires and stores the result. Then, the Assert section checks whether the result is of the ViewResult type. If so, the test passes. That makes sense because the Edit() action method displays the view again if the model state is not valid.

The third test method tests whether the Edit() action method for POST requests returns a RedirectToActionResult if the model state is valid. This works much like the second method. However, the Arrange section doesn't need to add an error to the model state. Instead, it needs to create a mock TempData object and assign it to the controller's TempData property. This is necessary because the code path for a valid model state uses TempData to display a message that indicates that the book was successfully edited. Conversely, the code path for an invalid model state doesn't use TempData.

After the Arrange section, the Act section calls the Edit() action method, passes it a BookViewModel object, and stores the result. Then, the Assert section checks whether the result is of the RedirectToActionResult type. If so, the test passes. That makes sense because the Edit() action method redirects to the List() action method if the model state is valid.

### The AdminBookControllerTests class (continued)

```
[Fact]
public void Edit_POST_ReturnsViewResultIfModelIsNotValid()
{
    // arrange
    var unit = GetUnitOfWork();
    var controller = new BookController(unit);
    controller.ModelState.AddModelError("", "Test error message.");
    BookViewModel vm = new BookViewModel();

    // act
    var result = controller.Edit(vm);

    // assert
    Assert.IsType<ViewResult>(result);
}

[Fact]
public void Edit_POST_ReturnsRedirectToActionResultIfModelIsValid()
{
    // arrange
    var unit = GetUnitOfWork();
    var controller = new BookController(unit);
    var temp = new Mock<ITempDataDictionary>();
    controller.TempData = temp.Object;
    BookViewModel vm = new BookViewModel { Book = new Book() };

    // act
    var result = controller.Edit(vm);

    // assert
    Assert.IsType<RedirectToActionResult>(result);
}
```

---

Figure 14-17 The AdminBookControllerTests class (part 2)

## The CartTests class

---

Figure 14-18 presents the CartTests class. This class tests a model, not a controller. As a result, the using directives for this class are different from the ones in the previous two figures. For example, this class doesn't need the Microsoft.AspNetCore.Mvc namespace. However, it does need the Microsoft.AspNetCore.Http namespace so it can easily access HttpContext objects. In addition, it needs the System.Linq namespace because the class uses LINQ's Sum() method, and it needs the System namespace because the class uses the Math.Round() method.

Within the class, the first method is a helper method named GetCart() that returns a Cart object. This method encapsulates the setup code presented in figure 14-14 to mock an HttpContextAccessor object. This helps to avoid duplicating the same setup code in multiple places in your test methods.

The first test method performs the same test as the method presented in figure 14-14. Now, though, it uses the GetCart() helper method to get a Cart object with a mock HttpContextAccessor object. This shortens the test code and makes it easier to read.

## The CartTests class

```
using System;
using System.Linq;
using Xunit;
using Moq;
using Microsoft.AspNetCore.Http;
using Bookstore.Models;

namespace Bookstore.Tests
{
    public class CartTests
    {
        private Cart GetCart()
        {
            // create HTTP context accessor
            var accessor = new Mock<IHttpContextAccessor>();

            // setup request and response cookies
            var context = new DefaultHttpContext();
            accessor.Setup(m => m.HttpContext)
                .Returns(context);
            accessor.Setup(m => m.HttpContext.Request)
                .Returns(context.Request);
            accessor.Setup(m => m.HttpContext.Response)
                .Returns(context.Response);
            accessor.Setup(m => m.HttpContext.Request.Cookies)
                .Returns(context.Request.Cookies);
            accessor.Setup(m => m.HttpContext.Response.Cookies)
                .Returns(context.Response.Cookies);

            // setup session
            var session = new Mock<ISession>();
            accessor.Setup(m => m.HttpContext.Session)
                .Returns(session.Object);

            return new Cart(accessor.Object);
        }

        [Fact]
        public void Subtotal_ReturnsADouble()
        {
            // arrange
            Cart cart = GetCart();
            cart.AddItem { Book = new BookDTO() };

            // act
            var result = cart.Subtotal;

            // assert
            Assert.IsType<double>(result);
        }
    }
}
```

---

Figure 14-18 The CartTests class (part 1)

The second test method checks whether the Subtotal property of a Cart object returns the correct calculation. To do that, this test method accepts a double array of prices as its parameter. This parameter uses the params keyword so the array of prices can be passed in a comma-separated list. That way, the test method can work with the comma-separated lists of prices contained in the `InlineData` attributes. This also makes it possible to pass a variable number of prices to the test method.

Within the test method, the `Arrange` section calls the private `GetCart()` method to get a `Cart` object that has a mock `HttpContextAccessor` object. Then, it loops through the `prices` array and creates a cart item for each price. In addition, it specifies a quantity of 1 for each item. Next, it adds each item to the cart. Finally, the code calculates the expected value by calling the `Sum()` LINQ extension method from the array of prices.

After the `Arrange` section, the `Act` section stores the value of the `Subtotal` property of the `Cart` object in a variable named `result`. Then, the `Assert` section compares the expected value to this `result` value. To correct for any imprecision that can occur with double values, it uses the static `Math.Round()` method to round both the expected and `result` values to 2 decimal places.

### The CartTests class (continued)

```
[Theory]
[InlineData(9.99, 6.89, 12.99)]
[InlineData(8.97, 45.00, 9.99, 15.00)]
public void Subtotal_ReturnsCorrectCalculation(
    params double[] prices)
{
    // arrange
    Cart cart = GetCart();
    for (int i = 0; i < prices.Length; i++)
    {
        var item = new CartItem
        {
            Book = new BookDTO { BookId = i, Price = prices[i] },
            Quantity = 1
        };
        cart.Add(item);
    }
    double expected = prices.Sum();

    // act
    var result = cart.Subtotal;

    // assert
    Assert.Equal(Math.Round(expected, 2), Math.Round(result, 2));
}
```

---

Figure 14-18 The CartTests class (part 2)

## Perspective

---

This chapter shows how to use dependency injection (DI) to make your code more flexible. In addition, it shows how to automate the testing of a web app by using DI to make it possible to work with unit testing. This includes using the xUnit and Moq frameworks. These skills should provide a good foundation for working with DI and unit testing, even if you need to use another unit testing framework such as MSTest or NUnit.

### Terms

---

dependency injection (DI)  
dependency  
inject a dependency  
dependency life cycle  
tightly coupled  
loosely coupled

dependency chaining  
unit testing  
unit  
Arrange/Act/Assert (AAA) pattern  
mock

### Summary

---

- *Dependency injection (DI)* is a design pattern in which the *dependencies* needed by an object are passed as parameters rather than being coded as part of the object.
- To use DI, you code the constructor of the object so it accepts an interface. Then, you can *inject* the dependency by passing any object to the constructor that implements the interface.
- When you map dependencies, you must decide which *dependency life cycle* to use. Most of the time, you can use the transient life cycle, which is the shortest life cycle.
- Controllers that receive DbContext objects via dependency injection are *tightly coupled* with EF Core because they must specify a class that's derived from the DbContext base class.
- Controllers that receive repository objects or unit of work objects via dependency injection are *loosely coupled* with EF Core because they only need to specify the appropriate interface, which may or may not use EF Core.
- When MVC creates an object to inject, it inspects the constructors of the class and injects dependencies according to the mappings in the Startup.cs file. This is called *dependency chaining*.
- *Unit testing* provides a way to write code that automatically tests individual methods, called *units*, to verify that they work properly in isolation.
- Code in a unit test is often organized using the *Arrange/Act/Assert (AAA) pattern*.
- One way to test a method that has a dependency is to use a fake version, or *mock*, of the dependency.

## Exercise 14-1 Add dependency injection and some unit tests

In this exercise, you'll modify a Class Schedule web app so it uses dependency injection. Then, you'll add a test project to the solution, write some unit tests with xUnit and Moq, and run the tests.

### Run the app

1. Open the Ch14Ex1ClassSchedule web app in the ex\_starts directory.
2. If you didn't do the exercises for chapter 12, open the Package Manager Console and enter the Update-Database command to create the database for this app.
3. Run the app and make sure it works correctly.

### Add dependency injection for the data layer classes

4. In the Startup.cs file, map dependencies for the following classes:

```
ClassScheduleUnitOfWork  
Repository<T>
```

5. Update the IClassScheduleUnitOfWork interface and the ClassScheduleUnitOfWork class to work with the IRepository<T> interface rather than the Repository<T> class.
6. Update the constructor of each controller to use DI to get the unit of work class or the repository class it needs.
7. Run the app and make sure it still works correctly.

### Add dependency injection for the HttpContextAccessor class

8. In the Startup.cs file, map a dependency for the HttpContextAccessor class.
9. Open the Home controller and add a parameter to its constructor of the IHttpContextAccessor type. Store this parameter value in a private property.
10. Update the controller code to use the HttpContext property of the IHttpContextAccessor interface instead of the HttpContext property of the Controller class.
11. Repeat steps 9 and 10 for the Class controller. Be sure to check all the action methods and helper methods for code that uses the HttpContext property.
12. Run the app. To test your changes, click Monday to only display the classes for Monday. Then, click Edit for a class. On the edit page, click Cancel. The app should “remember” your day filter.

### Add a test project to the solution

13. Add an xUnit Test Project (.NET Core) for C# to the solution. Name the test project ClassScheduleTests.
14. Add a reference to the project that contains the Class Schedule web app.
15. Open the Manage NuGet Packages for Solution window and add Moq to the test project.

**Write a test**

16. Rename the default file named UnitTest1.cs to TeacherControllerTests.cs. Then, rename the class it contains to TeacherControllerTests. (Visual Studio may offer to rename the class for you.)
17. Add using statements for the Xunit, Moq, Microsoft.AspNetCore.Mvc, ClassSchedule.Models, and ClassSchedule.Controllers namespaces.
18. Rename the default test method IndexActionMethod\_ReturnsAViewResult().
19. Within the test method, write code that checks that the Index() action method of the TeacherController class returns a ViewResult object. Use Moq to create the repository object the controller depends on.

**Open the Test Explorer and run your test**

20. Open the Test Explorer. If you don't see your test there, build the solution. If you still don't see your test, make sure it's decorated with the Fact attribute.
21. Run your test. If it fails, debug your test method until it passes.

**Write another test and run it**

22. Add a new class to the test project named HomeControllerTests. Make sure the class is public.
23. Add the using statements listed earlier in step 17. In addition, add a using statement for the Microsoft.AspNetCore.Http namespace.
24. Add a test method named IndexActionMethod\_ReturnsAViewResult().
25. Within that test method, write code that checks that the Index() action method of the HomeController class returns a ViewResult object. Use Moq to create the unit of work and HttpContextAccessor objects the controller depends on.
26. Run both of your tests. If either test fails, debug your test methods until they both pass.

# How to work with tag helpers, partial views, and view components

So far, this book has shown how to use the built-in tag helpers provided by MVC. Now, this chapter reviews those tag helpers and shows how to create custom tag helpers. In addition, it shows how to create partial views and view components. You can use these features to reduce code duplication in your views, which makes your views more flexible and easier to maintain.

<b>An introduction to tag helpers .....</b>	<b>604</b>
How to register and use tag helpers .....	604
How tag helpers compare to HTML helpers .....	606
<b>How to create custom tag helpers .....</b>	<b>608</b>
How to create a custom tag helper .....	608
How to create a tag helper for a non-standard HTML element .....	610
How to use extension methods with tag helpers .....	612
How to control the scope of a tag helper .....	614
How to use a tag helper to add elements .....	616
<b>More skills for custom tag helpers .....</b>	<b>618</b>
How to use properties with a tag helper .....	618
How to work with the model property that an element is bound to .....	620
How to use dependency injection with a tag helper .....	622
How to create a conditional tag helper .....	624
How to generate URLs in a tag helper .....	626
<b>How to work with partial views .....</b>	<b>628</b>
How to create and use a partial view .....	628
How to pass data to a partial view .....	630
<b>How to work with view components .....</b>	<b>632</b>
How to create and use a view component .....	632
How to pass data to a view component .....	634
How view components can simplify an app .....	636
<b>The Bookstore app .....</b>	<b>638</b>
The Book Catalog page .....	638
The updated ActiveNavbar tag helper .....	640
The layout .....	642
The Book>List view .....	644
<b>Perspective .....</b>	<b>646</b>

## An introduction to tag helpers

MVC provides several built-in tag helpers to generate HTML that's sent to a browser. To start, this chapter reviews some of these built-in tag helpers. Then, it compares tag helpers to the HTML helpers that were used prior to tag helpers.

### How to register and use tag helpers

The table in figure 15-1 presents some of the common tag helpers provided by MVC. In this table, the first six tag helpers provide attributes for standard HTML elements like `<form>` and `<input>` tags. The last two, by contrast, provide non-standard HTML elements.

The built-in tag helpers for attributes have a prefix of “asp-”. This makes it clear that the attributes are tag helpers and not standard HTML attributes. When you create custom tag helpers, it’s generally considered a good practice to follow this convention. You can use any letters you like for a prefix, such as the initials of your company. In this chapter, the examples use a prefix of “my-”.

Below the table, the code examples show how to use some of the built-in tag helpers. The first example uses the `asp-for` tag helper to bind an `<input>` tag to a model property. This tag helper generates several attributes for the `<input>` tag, including the data attributes needed for the data validation in the model. Using this tag helper keeps your view markup clean and makes sure these attributes are generated correctly.

The second example uses the `asp-action` tag helper with a `<form>` tag to generate the URL the form should post to. Since there’s no `asp-controller` tag helper, this tag helper uses the current controller and generates a relative URL. Using this tag helper makes the attribute more flexible and reduces the chances of coding an incorrect URL.

The third example uses the `<environment>` tag helper to send different CSS links to the browser for different hosting environments. For development, the view uses the regular Bootstrap CSS file. But in other environments such as a production environment, the app uses the minified version of this CSS file. Unlike the previous examples, this tag helper is an element, not an attribute. That’s why it doesn’t use a prefix.

Before you can use a tag helper in your app, you must register it. The easiest way to do that is to code one or more `@addTagHelper` directives in the `_ViewImports.cshtml` file as shown by the last example.

The first parameter for the `@addTagHelper` directive specifies the name of the tag helper to register. In this example, both directives use the wildcard symbol (\*) to indicate that all tag helpers should be registered.

The second parameter is the name of the assembly that contains the tag helpers. For custom tag helpers, this is usually the name of your web app as shown by the second directive.

## Common built-in tag helpers

Tag helper	Description
<code>asp-action</code>	Attribute that indicates the action method in a route.
<code>asp-controller</code>	Attribute that indicates the controller in a route.
<code>asp-area</code>	Attribute that indicates the area in a route.
<code>asp-for</code>	Attribute that binds an element to a property of a model object. MVC will generate the appropriate attributes for the element.
<code>asp-validation-summary</code>	Attribute that controls the display of validation messages. Usually applied to a <code>&lt;div&gt;</code> tag.
<code>asp-validation-for</code>	Attribute that controls the display of a single validation message. Usually applied to a <code>&lt;span&gt;</code> tag.
<code>environment</code>	Element that uses include and exclude attributes to display HTML based on the current hosting environment.
<code>partial</code>	Element that renders a partial view. See figure 15-13.

### A tag helper that generates attributes for an `<input>` tag

```
<input asp-for="FirstName" class="form-control" />
```

#### The HTML that's sent to the browser

```
<input class="form-control" type="text" data-val="true" data-val-
maxlength="The field FirstName must be a string or array type with a
maximum length of &#x27;200&x27;." data-val-maxlength-max="200" data-
val-required="Please enter a first name." id="FirstName" maxlength="200"
name="FirstName" value="" />
```

### A tag helper that generates a route-based URL in a `<form>` tag

```
<form asp-action="Edit" method="post"></form>
```

#### The HTML that's sent to the browser

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

### Tag helpers that output a different CSS link based on hosting environment

```
<environment include="Development">
    <link rel="stylesheet" href="~/lib/bootstrap/css/bootstrap.css" />
</environment>
<environment exclude="Development">
    <link rel="stylesheet" href="~/lib/bootstrap/css/bootstrap.min.css" />
</environment>
```

### A `_ViewImports.cshtml` file that registers all built-in and custom tag helpers

```
...
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper *, Bookstore
```

### Description

- To use tag helpers, you must register them with the `@addTagHelper` directive.

---

Figure 15-1 How to register and use tag helpers

## How tag helpers compare to HTML helpers

Tag helpers were introduced with ASP.NET MVC Core 2.1. Prior to that release, MVC provided HTML helpers. HTML helpers are Razor methods that you invoke with the standard Razor syntax. Then, the method outputs the correct HTML. For example, to create a `<form>` element, you could call the `Html.BeginForm()` and `Html.EndForm()` methods. Or, to create a `<label>` element, you could call the `Html.LabelFor()` method.

Figure 15-2 presents two examples of a strongly-typed view whose model is a `Book` object. The first example uses HTML helpers to create a `<form>` element with `<label>` elements and `<input>` elements that are bound to the model object. Then, the second example uses tag helpers to create the same form. The third example shows the HTML that the first two examples send to the browser.

The first example consists entirely of Razor code with no HTML. To use the methods of this Razor code, you need to know what arguments they expect. And, if you want to include an additional HTML attribute, like a `class` attribute or a `placeholder` attribute, you must code it as an anonymous object.

Unfortunately, this code makes it hard to visualize the HTML that's sent to the browser. In addition, it's not easy to add CSS classes to your HTML. This is particularly problematic on teams with web designers who don't have a lot of coding experience.

In contrast to the first example, the second example consists almost entirely of HTML with no Razor code. For instance, the `<form>` element is a normal HTML `<form>` element, and it uses the same method and `class` attributes as any other `<form>` element. However, it uses the `asp-action` tag helper to automatically generate the URL for the `action` attribute.

Similarly, the `<input>` elements are normal HTML `<input>` elements, and they use the same `class` attributes as any other `<input>` element. However, they use the `asp-for` tag helper to automatically generate the `type`, `id`, `name`, and `value` attributes of the `<input>` element.

This makes it easy to envision the HTML that's sent to the browser. In addition, it's easy to add CSS classes because you just code normal `class` attributes as needed. Because of this, most developers use tag helpers rather than HTML helpers. However, many online examples still show how to use HTML helpers, so it's good to know what they are and how they work.

## Razor code that uses HTML helpers (old technique)

```
@{ Html.BeginForm("Edit", "Home", FormMethod.Post, new { @class = "form-inline" }); }
@Html.LabelFor(m => m.Title)
@Html.TextBoxFor(m => m.Title, new { @class = "form-control m-2" })
@Html.LabelFor(m => m.Price)
@Html.TextBoxFor(m => m.Price, new { @class = "form-control m-2",
    placeholder = "e.g., $14.99" })
@{ Html.EndForm(); }
```

## HTML that uses tag helpers (new technique)

```
<form asp-action="Edit" method="post" class="form-inline">
    <label asp-for="Title">Title</label>
    <input asp-for="Title" class="form-control m-2" />
    <label asp-for="Price">Price</label>
    <input asp-for="Price" class="form-control m-2"
        placeholder="e.g., $14.99" />
</form>
```

## The HTML that both examples send to the browser

```
<form action="/Home/Edit" method="post" class="form-inline">
    <label for="Title">Title</label>
    <input type="text" id="Title" name="Title" value=""
        class="form-control m-2" />
    <label for="Price">Price</label>
    <input type="text" id="Price" name="Price" value=""
        class="form-control m-2" placeholder="e.g., $14.99" />
</form>
```

## Description

- Prior to version 2.1, ASP.NET MVC Core provided HTML helpers to generate HTML.
- With version 2.1 and later, ASP.NET MVC Core provides tag helpers to generate HTML.
- Tag helpers use markup that works like the markup for standard HTML elements. As a result, they're easy to read and work with.
- HTML helpers use C# code, not markup. As a result, they're more difficult to read and work with, especially for web designers with limited coding experience.

---

Figure 15-2 How tag helpers compare to HTML helpers

## How to create custom tag helpers

By now, you should understand how to use MVC's built-in tag helpers. However, it can be useful to create your own custom tag helpers for snippets of HTML that you use frequently.

### How to create a custom tag helper

To create a custom tag helper, you need to code a class that inherits the TagHelper class of the TagHelpers namespace. The first code example in figure 15-3 shows the using directive for that namespace. This using directive also makes it easy to access other classes for working with tag helpers such as the TagHelperOutput and TagHelperAttributeList classes described in this figure.

When you create a tag helper class, it's common to store it in a folder named TagHelpers, but that isn't required. Similarly, it's common, but not required, to add a suffix of TagHelper to the class name.

If you name your tag helper class so it starts with the same name as an HTML element, MVC automatically applies your tag helper to that element. In this figure, for example, the second example creates a tag helper named ButtonTagHelper. As a result, this tag helper targets the <button> element.

A tag helper class must inherit the TagHelper class, and it typically overrides the virtual Process() method of that class. The Process() method accepts TagHelperContext and TagHelperOutput arguments. This is illustrated by the ButtonTagHelper class shown here.

Within the Process() method, the code adds a class attribute that specifies two Bootstrap classes that style a button. To do that, it uses the Attributes property of the TagHelperOutput parameter to access a list of attributes for the <button> element. From this list of attributes, the code calls the SetAttribute() method to set the class attribute to "btn btn-primary". This overwrites any CSS classes that might already exist in the class attribute.

The third code example shows two <button> elements in a view. Then, the fourth example shows the HTML that MVC sends to the browser. This shows that the tag helper makes sure that each <button> element has both of the Bootstrap classes for styling buttons.

Because the Process() method is a synchronous method, it can only call synchronous methods of the TagHelperOutput class. However, this class also contains an asynchronous method named GetChildContentAsync() that you may want to use from time to time. To call this method, you must override the ProcessAsync() method instead of the Process() method. The signature for the ProcessAsync() method looks like this:

```
public override async Task ProcessAsync(TagHelperContext context,  
TagHelperOutput output)
```

And the code that calls the asynchronous method looks like this:

```
var content = await output.GetChildContentAsync();
```

## One virtual method of the TagHelper class

Method	Description
<code>Process(ctx, out)</code>	Manipulates and outputs an HTML element. Accepts a TagHelperContext object and a TagHelperOutput object. Also has an asynchronous version named ProcessAsync() that returns a Task object.

## One property of the TagHelperOutput class

Property	Description
<code>Attributes</code>	A TagHelperAttributeList object that holds an element's attributes.

## One method of the TagHelperAttributeList class

Property	Description
<code>SetAttribute(name, val)</code>	Sets the attribute in the attribute list with the specified name to the specified value. If the attribute doesn't exist, it's added to the end of the list.

## The using directive for the TagHelpers namespace

```
using Microsoft.AspNetCore.Razor.TagHelpers;
```

## A tag helper that applies to any standard HTML <button> element

```
public class ButtonTagHelper : TagHelper
{
    public override void Process(TagHelperContext context,
        TagHelperOutput output)
    {
        output.Attributes.SetAttribute("class", "btn btn-primary");
    }
}
```

### Two button elements in a view

```
<button type="submit">Submit</button>
<button type="reset">Reset Form</button>
```

### The HTML that MVC sends to the browser

```
<button type="submit" class="btn btn-primary">Submit</button>
<button type="reset" class="btn btn-primary">Reset Form</button>
```

## Description

- To create a custom tag helper, you code a class that inherits the TagHelper class, and you typically override its virtual Process() method.
- A tag helper class automatically applies to the HTML element of the same name.
- By convention, tag helper classes have a suffix of TagHelper, but this is not required.
- The TagHelperContext class represents the current context of an HTML element.
- The TagHelperOutput class represents the state of the HTML element.

---

Figure 15-3 How to create a custom tag helper

## How to create a tag helper for a non-standard HTML element

---

In the last figure, you learned how to apply a custom tag helper to a standard HTML element such as a <button> element. However, it's also possible to apply a custom tag helper to a non-standard HTML element such as a <submit-button> element. For that to work, your tag helper needs to transform the non-standard HTML element to a standard HTML element.

The table at the top of figure 15-4 presents two more properties of the TagHelperOutput class that you can use to transform an HTML element. First, you can use the TagName property to replace the name within the element's start tag as well as its end tag, if the element has an end tag.

Second, you can use the TagMode property to specify the type of start and end tags the element should have. To do that, it uses the TagMode enumeration. As the table indicates, you can use this property to output an element with both start and end tags (<label></label>), a start tag only (<input>), or a self-closing tag (<input />).

The code below the table shows a tag helper class named SubmitButtonTagHelper. MVC automatically applies this tag helper to elements of the same name minus the TagHelper suffix. In addition, it automatically translates between the Pascal casing common to C# classes and the kebab casing common to HTML. So, a tag helper named SubmitButton or SubmitButtonTagHelper targets a non-standard HTML element named <submit-button>.

Within this class, the Process() method starts by assigning “button” to the TagName property. Then, it assigns the StartTagAndEndTag value to the TagMode property. As a result, the tag helper outputs a <button> element with start and end tags. Next, the code adds an attribute to make sure the button is of type submit.

After transforming the non-standard <submit-button> element into a standard <button> element, the code adds two Bootstrap button classes. Unlike the previous figure, though, this code appends the Bootstrap classes to any existing CSS classes. To do that, it uses the Attributes property to get the old CSS classes from the class attribute. Since the class attribute might not exist, this statement uses the ? operator to check for nulls. Then, the code appends the new classes to any old classes and sets the resulting classes as the class attribute.

The second code example shows a <submit-button> element in a view. Then, the third example shows the HTML that MVC sends to the browser for that element. This shows that the tag helper transforms the <submit-button> element to a <button> element, adds a type attribute whose value is “submit”, and appends the “btn” and “btn-primary” class names to the existing class attribute.

Note that this tag helper doesn't change the inner HTML (the value between the start and end tags). This shows that a tag helper works with an existing element and only changes the specified parts of that element.

## More properties of the TagHelperOutput class

Property	Description
TagName	Replaces the current start tag. Also replaces the end tag, if applicable.
TagMode	Uses the TagMode enumeration to indicate the type of tag. Options are SelfClosing, StartAndEndTag, and StartTagOnly.

### A tag helper that applies to any non-standard <submit-button> element

```
public class SubmitButtonTagHelper : TagHelper
{
    public override void Process(TagHelperContext context,
        TagHelperOutput output)
    {
        // make it a button element with start and end tags
        output.TagName = "button";
        output.TagMode = TagMode.StartTagAndEndTag;

        // make it a submit button
        output.Attributes.SetAttribute("type", "submit");

        // append bootstrap button classes
        string newClasses = "btn btn-primary";
        string oldClasses = output.Attributes["class"]?.Value?.ToString();
        string classes = (string.IsNullOrEmpty(oldClasses)) ?
            newClasses : $"{oldClasses} {newClasses}";
        output.Attributes.SetAttribute("class", classes);
    }
}
```

### A submit-button element in a view

```
<submit-button class="mr-2">Submit</submit-button>
```

### The HTML that MVC sends to the browser

```
<button type="submit" class="mr-2 btn btn-primary">Submit</button>
```

## Description

- MVC automatically translates between the Pascal casing of a C# class name such as SubmitButton and the kebab casing of the corresponding HTML element such as <submit-button>.

---

Figure 15-4 How to create a tag helper for a non-standard HTML element

## How to use extension methods with tag helpers

The classes of the TagHelpers namespace are regular C# classes. As a result, you can add extension methods to them to encapsulate code that you use frequently. For instance, you might often need to append one or more classes to a class attribute like the example in the previous figure. Or, you might often need to transform an element to a standard HTML element with start and end tags.

Figure 15-5 shows a static TagHelperExtensions class with three static extension methods. First, the AppendCssClass() method extends the TagHelperAttributeList class. This is the data type of the Attributes property of the TagHelperOutput class. As a result, you can call this method from the Attributes property of the TagHelperOutput parameter of the Process() method.

The AppendCssClass() method accepts a string of CSS classes. Then, rather than overwriting existing CSS classes, this code appends the new classes to any old CSS classes. To do that, it uses code that's similar to the code shown in the previous figure.

Second, the BuildTag() method extends the TagHelperOutput class. This method accepts string arguments that specify the name of a tag and the names of any CSS classes. The code assigns the value of the tagName parameter to the TagName property of the TagHelperOutput object, and sets its TagMode property so the element has a start and end tag. Then, it calls the first extension method in this figure to append the value of the classNames parameter to the element's class attribute.

Note that the classNames parameter of the BuildTag() method is required. As a result, if you don't always want to add one or more CSS classes when you build a tag, you can change this parameter to be optional with a default value of an empty string.

Third, the BuildLink() method also extends the TagHelperOutput class. It accepts one string argument for a URL and another for a CSS class. Its code starts by calling the BuildTag() extension method shown earlier in this figure. This transforms the targeted element to an <a> element with start and end tags and a class attribute. Then, it calls the SetAttribute() method to add an href attribute for the url argument.

After presenting the three extension methods, this figure presents the tag helpers shown in the previous two figures after they have been updated to use these extension methods. Now, the ButtonTagHelper class uses the AppendCssClass() method to add two Bootstrap classes to the element's class attribute. This improves this tag helper because it no longer overwrites any existing CSS classes like the example in figure 15-3 does.

Similarly, the SubmitButtonTagHelper class now uses the BuildTag() method to transform the <submit-button> element to a <button> element with start and end tags and two Bootstrap classes appended to the element's class attribute. The use of this extension method replaces six statements with one statement. Then, it calls the SetAttribute() method of the Attributes property to make the button a submit button.

### Three extension methods for tag helpers

```
using Microsoft.AspNetCore.Razor.TagHelpers;
...
public static class TagHelperExtensions
{
    public static void AppendCssClass(this TagHelperAttributeList list,
        string newCssClasses)
    {
        string oldCssClasses = list["class"]?.Value?.ToString();
        string cssClasses = (string.IsNullOrEmpty(oldCssClasses)) ?
            newCssClasses : $"{oldCssClasses} {newCssClasses}";
        list.SetAttribute("class", cssClasses);
    }

    public static void BuildTag(this TagHelperOutput output,
        string tagName, string classNames)
    {
        output.TagName = tagName;
        output.TagMode = TagMode.StartTagAndEndTag;
        output.Attributes.AppendCssClass(classNames);
    }

    public static void BuildLink(this TagHelperOutput output,
        string url, string className)
    {
        output.BuildTag("a", className);
        output.Attributes.SetAttribute("href", url);
    }
}
```

### Two tag helpers that use the extension methods

```
public class ButtonTagHelper : TagHelper
{
    public override void Process(TagHelperContext context,
        TagHelperOutput output)
    {
        output.Attributes.AppendCssClass("btn btn-primary");
    }
}

public class SubmitButtonTagHelper : TagHelper
{
    public override void Process(TagHelperContext context,
        TagHelperOutput output)
    {
        output.BuildTag("button", "btn btn-primary");
        output.Attributes.SetAttribute("type", "submit");
    }
}
```

### Description

- You can use extension methods to add functionality to the classes of the TagHelpers namespace that you may use frequently.

---

Figure 15-5 How to use extension methods with tag helpers

## How to control the scope of a tag helper

So far, you've learned how to apply a tag helper to a standard or non-standard HTML element by naming your tag helper class to match that element. However, what if you don't want your tag helper to apply to all instances of an HTML element? For instance, you might want the Button tag helper to only apply to submit buttons. Or, what if you want your tag helper to apply to more than one element? For instance, you might want the Button tag helper to apply to `<button>` elements and `<input>` elements of the submit type.

Fortunately, you can control the elements that your tag helper class targets by decorating it with the `HtmlTargetElement` attribute. With this attribute, you can narrow the scope of your tag helper so it only applies to an element under certain conditions. You can also use it to widen the scope of your tag helper so it applies to more than one element. Or, you can use it to name the element you want to match so your class can use a different name.

The `HtmlTargetElement` attribute accepts a constructor argument that specifies the name of the HTML element to target. It also accepts the two properties presented in the table at the top of figure 15-6. The `Attributes` property specifies the attributes an HTML element must have for the tag helper to be applied. If there's more than one attribute, they can be included in a comma-separated list. The `ParentTag` property specifies that the element must be a child of the listed element for the tag helper to be applied.

Below the table, this figure presents several examples of tag helpers that use the `HtmlTargetElement` attribute to control their scope. The first example presents a class named `LabelTagHelper` that doesn't use an attribute. As a result, it applies to all `<label>` elements.

The second example uses the `HtmlTargetElement` attribute to specify that the class named `MyLabelTagHelper` applies to `<label>` elements. This allows the name of the class to be different from the element it targets.

The third example also applies to `<label>` elements. However, it uses the `Attributes` and `ParentTag` properties to narrow the scope of this tag helper so it only applies to `<label>` elements that have an `asp-for` attribute and are coded within a `<form>` element. Since the attribute specifies the element to target, this allows the name of the class to be more descriptive.

The fourth example shows that you can decorate your tag helper class with more than one `HtmlTargetElement` attribute. Here, the first attribute specifies that this tag helper applies to `<input>` elements within a `<form>` element. Then, the second attribute specifies that this tag helper also applies to `<select>` elements within a `<form>` element.

The last example shows another way to apply a tag helper to more than one element. Here, the first attribute skips the argument that specifies an HTML element and uses brackets to specify the name and value of the attribute. As a result, this tag helper applies to any element that has a `type` attribute with a value of "submit". Then, the second attribute applies this tag helper to `<a>` elements with a custom attribute named `my-button`.

## Two properties of the `HtmlTargetElement` attribute

Property	Description
<code>Attributes</code>	The attributes an HTML element must have for the tag helper to be applied.
<code>ParentTag</code>	The parent tag that must contain an HTML element for the tag helper to be applied.

### A tag helper that applies to any `<label>` element

```
public class LabelTagHelper : TagHelper {...}
```

### Another tag helper that applies to any `<label>` element

```
[HtmlTargetElement("label")]
public class MyLabelTagHelper : TagHelper {...}
```

### A tag helper that applies to bound `<label>` elements in a form

```
[HtmlTargetElement("label", Attributes = "asp-for", ParentTag = "form")]
public class FormLabelTagHelper : TagHelper {...}
```

### A tag helper that applies to `<input>` or `<select>` elements in a form

```
[HtmlTargetElement("input", ParentTag = "form")]
[HtmlTargetElement("select", ParentTag = "form")]
public class FormTagHelper : TagHelper {...}
```

### A tag helper that applies to any element of the submit type

#### or any `<a>` element with a my-button attribute

```
[HtmlTargetElement(Attributes = "[type=submit]")]
[HtmlTargetElement("a", Attributes = "my-button")]
public class MyButtonTagHelper : TagHelper {...}
```

## You can use the `HTMLTargetElement` attribute to...

- Allow a tag helper class to have a different name than the HTML element it targets.
- Narrow the scope of a tag helper so it only targets an element under certain conditions.
- Widen the scope of a tag helper so it targets multiple elements.

## Description

- The `HtmlTargetElement` attribute specifies the target HTML element for a tag helper.
- You can apply multiple `HtmlTargetElement` attributes to a tag helper class.
- Within the `HtmlTargetElement` attribute, you can use brackets to specify the name and value of an attribute.

---

Figure 15-6 How to control the scope of a tag helper

## How to use a tag helper to add elements

---

The first table in figure 15-7 shows that the TagHelperOutput class has properties named PreElement, Content, and PostElement. These properties are of the TagHelperContent type. This type provides many methods that you can use to get and set the HTML that's output to the browser. For example, it provides an AppendHtml() method that adds the HTML it receives to the output, and it provides a SetContent() method that replaces the text between the start and end tag of an element.

When you use these methods to set HTML, you could pass a string literal to the method like this:

```
output.PreElement.AppendHtml("<label>First Name</label>");
```

However, this approach tends to be error prone. As a result, it's generally considered a better practice to use the TagBuilder class to build the HTML.

The second table presents two properties of the TagBuilder class. The Attributes property is a dictionary that stores the attributes of the element being built. The InnerHtml property is a builder object that allows you to set the inner HTML of the element being built. In other words, it lets you set the HTML between the element's start and end tags.

Each of the properties in the second table has a method that's useful when building elements with the TagBuilder class. For example, the Add() method that's available from the Attributes property lets you add an attribute to the element, and the Append() method of the InnerHTML property lets you append HTML to the inner HTML of the element.

Below the tables, the first code example shows the using directive for the Rendering namespace that contains the TagBuilder class. Then, the second code example shows a tag helper that targets `<input>` elements that have an attribute named my-required.

Within its Process() method, the code starts by appending a Bootstrap CSS class named form-control to the `<input>` element. Then, it uses the TagBuilder class to create a `<span>` element, add a class attribute with two Bootstrap CSS classes to it, and set its inner HTML to an asterisk. To finish, this code uses the AppendHtml() method of the PostElement property to add the `<span>` element after the targeted `<input>` element.

The third example shows an `<input>` element that includes the my-required attribute. Then, the fourth example shows the HTML that MVC sends to the browser for that element. This shows that the required input tag helper adds a class attribute to the `<input>` element and a `<span>` element after it.

When you use the my-required attribute, you can code it without a value. That's because the value of the attribute isn't used anywhere. Rather, the attribute just needs to be present so MVC applies the tag helper. Also, note that MVC sends the attribute to the browser, but it isn't needed there since it isn't standard HTML. If you want, you can prevent it from being sent to the browser by coding a property in the tag helper with the same name as the attribute. You'll learn how to use properties with tag helpers in the next figure.

### Three properties of the TagHelperOutput class

Property	Description
PreElement	The HTML before an element.
Content	The main content between the start and end tags of an element.
PostElement	The HTML after an element.

### Two properties of the TagBuilder class

Property	Description
Attributes	A dictionary of an element's attributes. Can be used to add attributes to an element.
InnerHtml	A content builder that works with the inner HTML of an element.

### The using directive for the Rendering namespace

```
using Microsoft.AspNetCore.Mvc.Rendering; // for TagBuilder
```

### A tag helper that adds a <span> element after the targeted element

```
[HtmlTargetElement("input", Attributes = "my-required")]
public class RequiredInputTagHelper : TagHelper
{
    public override void Process(TagHelperContext context,
        TagHelperOutput output)
    {
        // add CSS class to input element
        output.Attributes.AppendCssClass("form-control");

        // create a <span> element
        TagBuilder span = new TagBuilder("span");
        span.Attributes.Add("class", "text-danger mr-2");
        span.InnerHtml.Append("*");

        // add span element after input element
        output.PostElement.AppendHtml(span);
    }
}
```

### An <input> element that uses the tag helper

```
<input asp-for="Title" my-required />
```

### The HTML that's sent to the browser

```
<input my-required type="text" id="Title" name="Title" value="" class="form-control" />
<span class="text-danger mr-2">*</span>
```

### The elements displayed in a browser



Figure 15-7 How to use a tag helper to add elements

## More skills for custom tag helpers

So far, this chapter has presented some basic skills for creating and using custom tag helpers. Now, this chapter shows how to take those skills to the next level.

### How to use properties with a tag helper

A tag helper class is a normal C# class. As a result, you can code normal C# properties for it. In a tag helper class, the properties correspond to the attributes for the HTML element that the tag helper targets. Visual Studio even provides IntelliSense support for these attributes. In addition, MVC automatically translates between the Pascal casing of the C# property name and the kebab casing of the HTML attribute.

One reason to code a property in a tag helper class is to provide a way to prevent attributes that you need to apply a tag helper to from being output to the browser. For example, if you added a Boolean Required property to the tag helper in the previous figure, MVC wouldn't output the required attribute as part of the HTML for the targeted element. More often, though, you code properties in a tag helper class to get data that the tag helper needs.

Figure 15-8 presents a tag helper class with properties that get a minimum and a maximum number. This tag helper applies to `<select>` elements that have attributes named `my-min-number` and `my-max-number`. Here, the properties named `Min` and `Max` correspond to those attributes, as indicated by the `HtmlAttributeName` attribute that decorates each property. Without these attributes, the properties would have to be named `MyMinNumber` and `MyMaxNumber`, which would make them more difficult to refer to in the tag helper class.

Within the `Process()` method, the code uses a for loop that starts at the value of the `Min` property and runs until the counter reaches the value of the `Max` property. With each iteration of the loop, the code uses the `TagBuilder` class to create an `<option>` element and set its inner HTML to the number stored by the loop's counter variable. Then, the code appends the `<option>` element to the HTML content of the targeted `<select>` element.

The second example shows a `<select>` element in a view with attributes for the `my-min-number` and `my-max-number` attributes of the tag helper. Then, the third example shows the HTML that MVC sends to the browser for that element. This shows that the tag helper adds `<option>` elements for the numbers 1 through 10, as illustrated by the fourth example.

In the second example, the `asp-for` tag helper binds the `<select>` element to the `Quantity` property of the view's `Model` object. However, the tag helper doesn't mark the option that matches the current `Quantity` value as selected. To do that, you can use the technique described in the next figure.

## A tag helper that generates numeric options for a <select> element

```
[HtmlTargetElement("select", Attributes = "my-min-number, my-max-number")]
public class NumberDropDownTagHelper : TagHelper
{
    [HtmlAttributeName("my-min-number")]
    public int Min { get; set; }

    [HtmlAttributeName("my-max-number")]
    public int Max { get; set; }

    public override void Process(TagHelperContext context,
        TagHelperOutput output)
    {
        for (int i = Min; i <= Max; i++)
        {
            TagBuilder option = new TagBuilder("option");
            option.InnerHtml.Append(i.ToString());
            output.Content.AppendHtml(option);
        }
    }
}
```

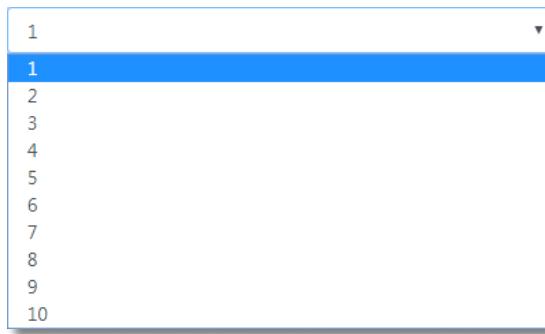
## A view that uses the tag helper

```
@model CartItem
...
<select asp-for="Quantity" class="form-control"
    my-min-number="1" my-max-number="10"></select>
```

## The HTML that's sent to the browser

```
<select class="form-control" id="Quantity" name="Quantity">
    <option>1</option>
    <option>2</option>
    ...
    <option>10</option>
</select>
```

## The <select> element displayed in a browser



## Description

- You can use properties in a tag helper class to get data from the view.
- You can use the HtmlAttributeName attribute to define the name of the HTML attribute if you want it to be different from the property name.

---

Figure 15-8 How to use properties with a tag helper

## How to work with the model property that an element is bound to

---

As you've seen, the Process() method in a tag helper class accepts TagHelperContext and TagHelperOutput objects. So far, you've learned how to use the Attributes property of a TagHelperOutput object to work with the attributes of an element. Now, figure 15-9 shows how to use the AllAttributes property of the TagHelperContext class to work with the attributes of an element.

The difference between these two properties is that the TagHelperContext class allows you to work with the attributes that are coded in the view, and TagHelperOutput class allows you to work with the attributes that are sent to the browser. As a result, if a view includes this HTML:

```
<input asp-for="FirstName" />
```

the TagHelperContext object stores one item for the asp-for attribute, but the TagHelperOutput object stores all attributes generated for the element including items for the type, id, name, and value attributes as well as any data validation attributes.

Since the AllAttributes property stores an item for the asp-for attribute, you can use it to get information about the property that the targeted element is bound to. To do that, you need to convert that item to the ModelExpression type that's available from the ViewFeatures namespace shown in the first code example.

The second table in this figure shows three of the properties of the ModelExpression class. The Name property stores the name of the bound property, and the Model property stores its value. However, the Model property is of the Object type. As a result, you may need to cast it to the actual data type of the property. Finally, the Metadata property contains other attributes of the bound property, including the value of any Display attribute.

The second code example shows the tag helper from the previous figure after it has been updated to use the TagHelperContext object to get the asp-for attribute. To do that, this code casts the value of the asp-for attribute that's available from the AllAttributes item to a ModelExpression type. Then, it casts the value of the Model property to the int type.

Within the loop, the code checks whether the value of the counter matches the value from the Model property. If so, the code adds a selected attribute to the <option> element. This marks the option that matches the current Quantity value as selected.

## One property of the TagHelperContext class

Property	Description
AllAttributes	A list containing all attributes of the targeted element.

## Three properties of the ModelExpression class

Property	Description
Name	The name of the bound property.
Model	The value of the bound property.
Metadata	Other attributes of the bound property including the display name.

## The using statement for the ViewFeatures namespace

```
using Microsoft.AspNetCore.Mvc.ViewFeatures; // for ModelExpression
```

## The updated NumberDropDownTagHelper class

```
[HtmlTargetElement("select", Attributes = "my-min-number, my-max-number")]
public class NumberDropDownTagHelper : TagHelper
{
    // Min and Max properties same as before

    public override void Process(TagHelperContext context,
        TagHelperOutput output)
    {
        // get selected value from view's model
        ModelExpression aspfor =
            (ModelExpression) context.AllAttributes["asp-for"].Value;
        int modelValue = (int)aspfor?.Model;

        for (int i = Min; i <= Max; i++)
        {
            TagBuilder option = new TagBuilder("option");
            option.InnerHtml.Append(i.ToString());

            // mark option as selected if matches model's value
            if (modelValue == i)
                option.Attributes["selected"] = "selected";

            output.Content.AppendHtml(option);
        }
    }
}
```

## Description

- The ModelExpression class represents a model property that's bound to an element with the asp-for tag helper.
- You can use the AllAttributes property of the TagHelperContext class to retrieve a ModelExpression object for a bound element.

---

Figure 15-9 How to work with the model property that an element is bound to

## How to use dependency injection with a tag helper

Dependency injection (DI) provides a way to inject the objects that a class needs into the class without explicitly creating them within that class. Chapter 14 describes how DI works. If you've already read that chapter, you shouldn't have any trouble understanding figure 15-10. If you haven't, you might want to skim that chapter to better understand this figure.

In this figure, the first tag helper class marks an `<a>` element in a Bootstrap navbar as active. To do that, it uses DI to get the `ViewContext` property from the view. This property provides a way to access data for the view such as its route data, `ViewBag`, session state, `TempData`, and so on.

To inject a `ViewContext` object into a tag helper, you can code a property of the `ViewContext` type and decorate that property with the `ViewContext` attribute. Interestingly, this data type and attribute are in different namespaces as shown by the first example. In addition, you should decorate the property with the `HtmlAttributeNotBound` attribute. This tells MVC that the value doesn't come from the HTML of the view.

Once you've coded a property with the `ViewContext` data type and attribute, MVC automatically injects the view's `ViewContext` object into this property. In other words, you don't need to do any further configuration for this to work.

In the `Process()` method, the code uses the `ViewContext` object to get the name of the current area and controller from the `Values` property of the `RouteData` property. Then, it uses the `AllAttributes` property of the `TagHelperContext` object to get the values of the "asp-area" and "asp-controller" tag helpers. Next, it compares the current area and controller with the corresponding tag helper values. If they match, the tag helper appends the Bootstrap's active class to the `class` attribute of the `<a>` element.

The second tag helper class has a constructor that accepts an `ICart` parameter. If you map a dependency for the `Cart` class as described in figure 14-5, MVC automatically passes a `Cart` object to this constructor. Then, the constructor stores this object in the private `cart` variable. As a result, the `Process()` method can use the `Count` property of the `Cart` object to set the text of the targeted `<span>` element.

The tag helpers in this figure remove the need for a layout to use Razor code to set the active link or to display the count in the cart badge. The first tag helper does this without adding any extra attributes to the Bootstrap navbar. However, the second tag helper requires the layout to include a `my-cart-badge` attribute like the one shown in the third example. Note that the `Process()` method for this tag helper doesn't use the corresponding `MyCartBadge` property. Instead, this property is included to prevent the `my-cart-badge` attribute that's coded in the layout from being sent to the browser.

## A tag helper that gets a ViewContext value via dependency injection

```
using Microsoft.AspNetCore.Mvc.ViewFeatures;           // ViewContext attribute
using Microsoft.AspNetCore.Mvc.Rendering;            // ViewContext data type
...
[HtmlTargetElement("a", Attributes = "[class=nav-link]", ParentTag = "li")]
public class ActiveNavbarTagHelper : TagHelper
{
    [ViewContext]
    [HtmlAttributeNotBound]
    public ViewContext ViewCtx { get; set; }

    public override void Process(TagHelperContext context,
        TagHelperOutput output)
    {
        string area = ViewCtx.RouteData.Values["area"]?.ToString() ?? "";
        string ctrlr = ViewCtx.RouteData.Values["controller"].ToString();

        string aspArea = context.AllAttributes["asp-area"]?.Value?
            .ToString() ?? "";
        string aspCtrlr = context.AllAttributes["asp-controller"].Value
            .ToString();

        if (area == aspArea && ctrlr == aspCtrlr)
            output.Attributes.AppendCssClass("active");
    }
}
```

## A tag helper that gets a Cart object via dependency injection

```
[HtmlTargetElement("span", Attributes = "my-cart-badge")]
public class CartBadgeTagHelper : TagHelper
{
    private ICart cart;
    public CartBadgeTagHelper(ICart c) => cart = c;

    public bool MyCartBadge { get; set; }

    public override void Process(TagHelperContext context,
        TagHelperOutput output)
    {
        output.Content.SetContent(cart.Count?.ToString());
    }
}
```

### The tag helper in a layout

```
<span class="fas fa-shopping-cart"></span>&nbsp;Cart
<span class="badge badge-light" my-cart-badge></span>
```

## Description

- If you create a property of the ViewContext type and decorate it with the ViewContext attribute, MVC automatically injects the ViewContext object into the class.
- The HtmlAttributeNotBound attribute tells MVC that a property isn't set in the HTML.
- You can also use dependency injection to manually inject an object into a tag helper.

Figure 15-10 How to use dependency injection with a tag helper

## How to create a conditional tag helper

---

Sometimes, you only want to display an HTML element under certain circumstances. For instance, the first code example in figure 15-11 presents a layout that uses Razor code to only display an `<h4>` element if there's a message in TempData.

Then, the second code example shows how to use a tag helper to perform the same task. To do that, the tag helper can use the `SuppressOutput()` method that's summarized in the table. This method prevents MVC from sending any HTML for the targeted element to the browser.

In the second example, the tag helper class begins by getting the `ViewContext` object from the layout using the technique presented in the previous figure. That way, the `Process()` method can use the `ViewContext` object to access the `TempData` for the layout.

Within the `Process()` method, the code uses the `ViewContext` object to get the `TempData` property and store it in a variable named `td`. Then, the code checks whether the key named `message` is in `TempData`.

If the key is in `TempData`, the code calls the `BuildTag()` extension method of the `TagHelperOutput` class and passes it a tag name of “`h4`” and a string that specifies several Bootstrap CSS classes. This transforms the non-standard `my-temp-message` element to a standard `<h4>` element and sets its `class` attribute. Next, it sets the content of the `<h4>` element to display the message from `TempData`.

However, if the key isn't in `TempData`, the code calls the `SuppressOutput()` method. This tells MVC to not send any HTML for the tag helper.

The third code example shows the layout after it's updated to use this tag helper. This shows that the code that uses the tag helper is more concise than the Razor code. In fact, no Razor code is needed to implement this functionality in the layout.

## A layout that only displays an element if there's a value in TempData

```
<main>
    @if (TempData.Keys.Contains("message"))
    {
        <h4 class="bg-info text-center text-white p-2">
            @TempData["message"]
        </h4>
    }
    @RenderBody()
</main>
```

## A method of the TagHelperOutput class

Method	Description
SuppressOutput()	Prevents MVC from sending any HTML for the targeted element to the browser.

## A tag helper that only outputs HTML if there's a value in TempData

```
using Microsoft.AspNetCore.Mvc.ViewFeatures;           // ViewContext attribute
using Microsoft.AspNetCore.Mvc.Rendering;             // ViewContext class
...
[HtmlTargetElement("my-temp-message")]
public class TempMessageTagHelper : TagHelper
{
    [ViewContext]
    [HtmlAttributeNotBound]
    public ViewContext ViewCtx { get; set; }

    public override void Process(TagHelperContext context,
        TagHelperOutput output)
    {
        var td = ViewCtx.TempData;
        if (td.Keys.Contains("message"))
        {
            output.BuildTag("h4", "bg-info text-center text-white p-2");
            output.Content.SetContent(td["message"].ToString());
        }
        else
        {
            output.SuppressOutput();
        }
    }
}
```

## A layout that uses the conditional tag helper

```
<main>
    <my-temp-message />
    @RenderBody()
</main>
```

## Description

- You can use the SuppressOutput() method of the TagHelperOutput class to create tag helpers that only send an element to the browser under certain conditions.

---

Figure 15-11 How to create a conditional tag helper

## How to generate URLs in a tag helper

---

Sometimes, you need your tag helper to generate a URL. In such a case, it's better to use the routing system to generate that URL rather than hardcoding the URL. MVC provides a LinkGenerator class that allows you to use the routing system to generate URLs. To use this class, you should include a using directive for the Microsoft.AspNetCore.Routing namespace.

Figure 15-12 presents a tag helper class that uses a LinkGenerator object to create a paging link. This class has a constructor that accepts a LinkGenerator object. When you code a constructor this way, MVC automatically injects the LinkGenerator object. You don't have to do any other configuration.

This tag helper class has a property named Number of the int type. This is how the tag helper gets the page number of the link it's creating. In addition, the tag helper has a property named Current of the RouteDictionary type like the one presented in figures 13-7 and 13-12. This is how the tag helper gets the current route segments it needs to build the route-based URL.

Within the Process() method, the code calls the Clone() method of the RouteDictionary property to get a copy of the route segments. Then, it updates the PageNumber route segment to the value for this paging link.

Next, the code uses the ViewContext object it receives via dependency injection to get the name of the current controller and action method. Then, it passes these values and the updated route segments to the GetPathByAction() method of the LinkGenerator class. This method returns a route-based URL, which the code stores in a variable named url.

After creating the URL for the link, the code builds the CSS for the paging link. First, it codes a string named linkClasses with two Bootstrap classes. Then, it checks whether the value in the Number property matches the current page number. If it does, the code appends the active class to the linkClasses string.

Finally, the code calls the BuildLink() extension method of the TagHelperOutput class and passes it the route-based URL and the linkClasses string. This transforms the non-standard my-paging-link element to a standard <a> element and sets its href and class attributes. Then, it sets the content of the <a> element to display the value in the Number property.

The second example shows a view that uses this tag helper. This shows how the Number and Current properties in the class correspond to the number and current attributes in the view. The counter variable for the loop supplies the value for the number attribute, and the CurrentRoute property of the view's Model object supplies the value for the current attribute.

## A tag helper that generates a paging link

```
using Microsoft.AspNetCore.Mvc.ViewFeatures;           // ViewContext attribute
using Microsoft.AspNetCore.Mvc.Rendering;             // ViewContext data type
using Microsoft.AspNetCore.Routing;                // LinkGenerator
using Bookstore.Models;                             // RouteDictionary

...
[HtmlTargetElement("my-paging-link")]
public class PagingLinkTagHelper : TagHelper
{
    private LinkGenerator linkBuilder;
    public PagingLinkTagHelper(LinkGenerator lg) => linkBuilder = lg;

    [ViewContext]
    [HtmlAttributeNotBound]
    public ViewContext ViewCtx { get; set; }

    public int Number { get; set; }
    public RouteDictionary Current { get; set; }

    public override void Process(TagHelperContext context,
        TagHelperOutput output)
    {
        // update routes for this paging link
        var routes = Current.Clone();
        routes.PageNumber = Number;

        // get controller and action method, create paging link URL
        string ctrr = ViewCtx.RouteData.Values["controller"].ToString();
        string action = ViewCtx.RouteData.Values["action"].ToString();
        string url = linkBuilder.GetPathByAction(action, ctrr, routes);

        // build up CSS string
        string linkClasses = "btn btn-outline-primary";
        if (Number == Current.PageNumber)
            linkClasses += " active";

        // create link
        output.BuildLink(url, linkClasses);
        output.Content.SetContent(Number.ToString());
    }
}
```

## A view that uses the tag helper

```
@{
    for (int i = 1; i <= Model.TotalPages; i++)
    {
        <my-paging-link number="@i" current="@Model.CurrentRoute" />
    }
}
```

## Description

- You can use the GetPathByAction() method of the LinkGenerator class to generate a route-based URL. This class is in the Microsoft.AspNetCore.Routing namespace.
- If your tag helper has a constructor with a LinkGenerator parameter, MVC will inject it.

---

Figure 15-12 How to generate URLs in a tag helper

## How to work with partial views

By now, you've learned that a tag helper is a powerful tool that provides many ways to reduce code duplication in a view. However, if you find yourself coding a tag helper that outputs a lot of HTML, it might be better to use a partial view.

### How to create and use a partial view

A *partial view* can contain HTML, Razor code, or a combination of the two. Partial views are useful for blocks of HTML or Razor code that occur in multiple places in an app or in multiple apps.

To create a partial view, you start by following the same procedure for creating a normal view. Then, in the Add MVC View dialog, you check the “Create as a partial view” box as shown in figure 15-13. Since a partial view is just a view, it's OK if you forget to check this checkbox. In that case, you just need to delete the HTML that Visual Studio generates for a normal view.

When you create a partial view, you can name it whatever you want. Many developers prefer to add a prefix of an underscore and a suffix of Partial to the name of a partial view, and that's the approach this book uses. However, this isn't required.

Like other views, MVC expects partial views to be in the Views folder. MVC looks for a partial view in the folder for the current controller and in the Shared folder as shown by the first example.

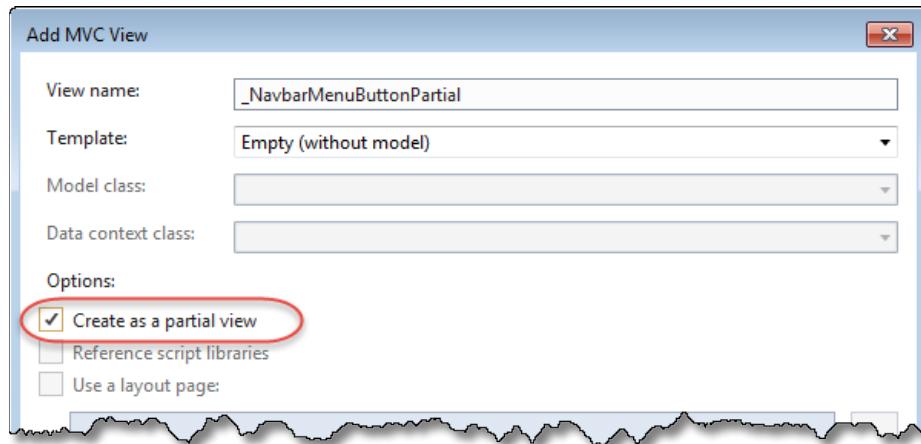
When you create a new project for an ASP.NET Core web app with the MVC template, Visual Studio generates a partial view named \_ValidationScriptsPartial in the Views/Shared folder. This partial view contains <script> tags for the jQuery validation libraries shown in the second example. Then, you can include this partial view in any page that requires validation.

To use a partial view, you need to tell MVC where in your view to insert the partial view. To do that, you can use the partial tag helper as shown by the third example. This renders the partial view asynchronously, which is usually what you want.

The fourth example shows a partial view named \_NavbarMenuButtonPartial. This partial view contains the HTML for the menu button portion of a Bootstrap navbar. This is a good candidate for a partial view because it's needed in many apps, it doesn't usually change from one app to the next, and it's easy to type the HTML incorrectly.

The fifth example shows a layout that uses the partial tag helper to include the \_NavbarMenuButtonPartial view as part of a Bootstrap navbar. This reduces several lines of HTML to a single line.

## The Add MVC View dialog



### The paths that MVC searches for a partial view

/Views/ControllerName/PartialViewName  
/Views/Shared/PartialViewName

### A partial view that loads the jQuery validation libraries

```
<script src="~/lib/jquery-validation/dist/jquery.validate.min.js"></script>
<script src="~/lib/jquery-validation-unobtrusive/
    jquery.validate.unobtrusive.min.js"></script>
```

### A tag helper that includes the partial view in a view

```
<partial name="_ValidationScriptsPartial" />
```

### A partial view that contains the HTML for a Bootstrap navbar menu button

```
<button class="navbar-toggler" type="button" data-toggle="collapse"
    data-target="#menu" aria-controls="menu" aria-expanded="false"
    aria-label="Toggle navigation">
    <span class="navbar-toggler-icon"></span>
</button>
```

### A layout that uses the partial view in a Bootstrap navbar

```
<nav class="navbar navbar-expand-md navbar-dark bg-dark">
    <partial name="_NavbarMenuButtonPartial" />
    <div class="collapse navbar-collapse" id="menu">
        <ul class="navbar-nav mr-auto">...</ul>
    </div>
</nav>
```

### Description

- A *partial view* can contain HTML and Razor code that can be used in multiple views.

---

Figure 15-13 How to create and use a partial view

## How to pass data to a partial view

---

MVC treats a partial view as part of the view it's added to. As a result, the partial view can use the model object of the parent view. That means the model object for a partial view can change, depending on the parent view.

As a result, you may sometimes want to specify the model for a partial view. In that case, you can use the `model` attribute or the `for` attribute of the partial tag helper. These attributes are presented in the table in figure 15-14. Since these attributes are both used to set the model value, you can't use them together. In other words, if you set the `model` property, you can't set the `for` property.

A partial view also uses the same ViewData dictionary as the parent view. However, you can use the `viewData` attribute of the partial tag helper to change that too.

Below the table, the first example shows a partial view that creates a link to the details page for a book. To work properly, this partial view needs a `Book` object as its model object.

If the parent view uses a `Book` object as its model object, you can use the partial tag helper shown in the second example to include the `_BookLinkPartial` view. This works because the parent view uses a `Book` object as its model. As a result, there's no need to specify the model object.

However, if the parent view doesn't use a `Book` object as its model object, the partial tag helper needs to set the model for the partial view. In the third example, for instance, the parent view uses a `BookListViewModel` object as its model. As a result, the partial tag helper for the `_BookLinkPartial` view uses the `model` attribute to specify a `Book` object as its model. To do that, it uses a Razor expression to specify the name of the variable within the loop that contains a `Book` object.

## Four attributes of the partial tag helper

Attribute	Description
<code>name</code>	The name of the partial view to render. This attribute is required.
<code>model</code>	The object for the partial view to use as its model. May not be used with the <code>for</code> attribute.
<code>for</code>	The object for the partial view to use as its model. May not be used with the <code>model</code> attribute.
<code>viewData</code>	The ViewDataDictionary object for the partial view to use as its ViewData.

### The `_BookLinkPartial` partial view

```
@model Book

<a asp-action="Details" asp-controller="Book"
    asp-route-id="@Model.BookId"
    asp-route-slug="@Model.Title.Slug()">
    @Model.Title
</a>
```

### The partial view included in a view that has the same model object

#### The Home/Index view

```
@model Book
...
<h5>
    <partial name="_BookLinkPartial" />
</h5>
...
```

### The partial view included in a view that has a different model object

#### The Book/List view

```
@model BookListViewModel
...
@foreach (Book book in Model.Items) {
<tr>
    <td>
        <partial name="_BookLinkPartial" model="@book" />
    </td>
...
}
```

## Description

- By default, the model object of a partial view is the model object of the parent view.
- By default, the ViewData dictionary of a partial view is the ViewData dictionary of the parent view.
- If you want the partial view to use a different model or ViewData dictionary, you can use the attributes of the partial tag helper to specify that model or dictionary.

---

Figure 15-14 How to pass data to a partial view

## How to work with view components

Tag helpers and partial views provide many benefits, but they have downsides too. Tag helpers become unwieldy if they generate too much HTML, and partial views may behave differently from view to view. Both depend on the controller of the parent view for their data, and both are difficult to test. Luckily, view components solve most of these problems.

### How to create and use a view component

A view component has two parts. First it has a class that functions as its controller. Second, it has a partial view that functions as its view. The class is a regular C# class that inherits the ViewComponent class, which is in the Microsoft.AspNetCore.Mvc namespace. Many developers store these classes in a separate folder named Components, but that isn't required.

Unlike the TagHelper class, the ViewComponent class doesn't have virtual methods to override. Instead, MVC typically expects a view component class to contain a method named `Invoke()`. This method is described in figure 15-15.

The first example shows that the view component class in the second example is stored in the Components folder. This class is named `CartBadge`, and it inherits the `ViewComponent` class. The `ViewComponent` class defines a synchronous `Invoke()` method. Like a controller class, the view component class has a constructor that accepts an `ICart` object via dependency injection.

Within the view component class, the `Invoke()` method functions like an action method. It passes the `Count` property of the `Cart` object to the view via the `View()` method. That makes it easy to test this view component using the techniques described in chapter 14. This is an improvement on the `CartBadge` tag helper presented earlier in this chapter because that tag helper would be hard to test.

The third example shows the partial view that's returned by the view component class. This partial view has a model object of a nullable int, which is the type that's returned by the `Count` property of the `Cart` object. Then, the partial view uses that object as the content of a `<span>` tag.

By default, MVC searches for the partial view that's associated with the view component class in the paths presented in the fourth example. However, you can override these default paths as shown in the next figure.

The last example shows how to use tag helper syntax to add a view component to a view. Here, the tag name consists of a prefix of "vc:" plus the name of the view component class. As with tag helpers, MVC translates between the Pascal case of the class name and the kebab case of the tag.

Although you'll typically code synchronous methods in your view components, you can also code asynchronous methods. To do that, you use the `InvokeAsync()` method instead of the `Invoke()` method. In that case, your method signature looks like this:

```
public async Task<IVViewComponentResult> InvokeAsync()
```

## A method MVC looks for in a ViewComponent class

Method	Description
<code>Invoke([params])</code>	Defines the logic of the view component and then calls a partial view. Is similar to an action method of a controller. Can have zero or more parameters. Returns an <code>IViewComponentResult</code> object. Has an asynchronous version named <code>InvokeAsync()</code> that returns a <code>Task&lt;IViewComponentResult&gt;</code> object.

## The folder that stores the view component classes

/Components

## A view component class that passes the Cart count to a partial view

```
using Microsoft.AspNetCore.Mvc;      // for view component classes
...
public class CartBadge : ViewComponent
{
    private ICart cart { get; set; }
    public CartBadge(ICart c) => cart = c;

    public IViewComponentResult Invoke() => View(cart.Count);
}
```

## The Default.cshtml partial view

```
@model int?
<span class="badge badge-light">@Model</span>
```

## The paths that MVC searches for a view component's partial view

/Views/ControllerName/Components/ViewComponentName/ViewName  
 /Views/Shared/Components/ViewComponentName/ViewName

## A layout that uses tag helper syntax to call the view component

```
<span class="fas fa-shopping-cart"></span>&ampnbspCart
<vc:cart-badge></vc:cart-badge>
```

### Description

- A *view component* is a class that sends data to a partial view. You can think of a view component as a controller for a partial view.
- To create a view component, you can create a new class file in the Components folder that inherits the `ViewComponent` class. Then, you typically code an `Invoke()` method as described above.
- The partial view for a view component is usually named `Default.cshtml`.
- You use tag helper syntax with a prefix of “vc:” to use a view component. For this to work, you must register the custom tag helpers for your app as described in figure 15-1.

---

Figure 15-15 How to create and use a view component

## How to pass data to a view component

One way to pass data to a view component is via dependency injection as shown in the previous figure. Another way is to add one or more parameters to the Invoke() method. These parameters correspond to the attributes in the tag helper for the view component. Visual Studio even provides IntelliSense support for them. As you would expect, MVC automatically translates between the camel casing of the parameter name and the kebab casing of the tag helper attribute.

Figure 15-16 starts by presenting a view model class named DropDownListViewModel. It contains the data needed to create a <select> element. This includes a property for a string-string dictionary that stores the drop-down items, a property for the selected value, and a property for the default value.

The second example presents a view component class named AuthorDropDown. This view component creates a drop-down list of authors. Since the view component class receives an IRepository<Author> object in its constructor via dependency injection, it doesn't need to get this data from the controller of its parent view. This can simplify the controllers and views in an app as shown in the next figure.

The Invoke() method of the AuthorDropDown view component accepts a string parameter named selectedValue. This method starts by getting a list of authors from the database ordered by first name. Then, it creates a new DropDownListViewModel object. When it does that, it assigns the selectedValue parameter to the SelectedValue property, a value of "All" to the DefaultValue property, and the list of authors to the Items property. For this final assignment, the code transforms the list of authors to a dictionary with the author's ID as the key and the author's full name as the value.

The last statement in the Invoke() method calls the View() method and passes it two arguments. The first argument specifies a fully qualified path to the view file. This overrides the default search done by MVC. That means that, rather than looking for a view file named Default.cshtml in the default folders described in the previous figure, MVC looks for a view file named DropDownList.cshtml in the specified path. This is a useful technique when you have several view components that share the same partial view.

The second argument specifies the DropDownListViewModel object that contains a dictionary of authors, a selected value, and a default value. As a result, the Invoke() method passes the partial view named DropDownList all the data it needs to create a <select> element.

The third example shows the code for the DropDownList partial view. Its model is a DropDownListViewModel object. It uses this view model to build a <select> element used for filtering. Since a partial view is a normal view, it can use built-in tag helpers like asp-items in its HTML.

The fourth example shows how to use the AuthorDropDown view component in a view. This code gets an author filter value from the view model and uses the selected-value attribute to pass it to the selectedValue parameter of the view component class.

## The DropDownListViewModel class

```
public class DropDownListViewModel
{
    public Dictionary<string, string> Items { get; set; }
    public string SelectedValue { get; set; }
    public string DefaultValue { get; set; }
}
```

## A view component with an Invoke() method that has a parameter

```
public class AuthorDropDown : ViewComponent
{
    private IRepository<Author> data { get; set; }
    public AuthorDropDown(IRepository<Author> rep) => data = rep;

    public IViewComponentResult Invoke(string selectedValue)
    {
        var authors = data.List(new QueryOptions<Author> {
            OrderBy = a => a.FirstName
        });

        var vm = new DropDownListViewModel {
            SelectedValue = selectedValue,
            DefaultValue = "All",
            Items = authors.ToDictionary(
                a => a.AuthorId.ToString(), a => a.FullName)
        };
        return View("~/Views/Shared/Components/Common/DropDown.cshtml", vm);
    }
}
```

## The DropDownList partial view

```
@model DropDownListViewModel

<select name="filter" class="form-control m-2"
    asp-items="@new SelectList(
        Model.Items, "Key", "Value", Model.SelectedValue))">
    <option value="@Model.DefaultValue">All</option>
</select>
```

## A view that uses the view component

```
<label>Author: </label>
<vc:author-drop-down selected-value="@Model.CurrentRoute.AuthorFilter">
</vc:author-drop-down>
```

## Description

- You can pass data to a view component by coding parameters in the Invoke() method. Then, you code the parameter value as an attribute in the tag helper for the view component.
- You can override the default search for the partial view by passing a fully qualified view name and path as the first argument of the View() method.

---

Figure 15-16 How to pass data to a view component

## How view components can simplify an app

So far, this book has presented versions of the Bookstore app that have separate view model classes for the Book Catalog page and the Author catalog page. That's because the Book Catalog page has some drop-down lists for filtering, so the List() action method for that page needs to provide that data. To do that, it adds that data to the view model that's sent to the view for the Book Catalog page.

However, if you use view components for those drop-down lists, the responsibility for getting that data is moved to those view components. Because of that, the List() action method no longer needs to handle this filtering data and can focus on retrieving the list of books the catalog page needs. This simplifies the Book controller code.

In addition, once the List() action method for the Book Catalog page is simplified, it can share a generic view model class with the List() action method for the Author Catalog page. This simplifies the view model code as well.

The first example in figure 15-17 shows the simplified generic GridViewModel class. This class has an Items property of type `IEnumerable<T>`, which means it can hold a collection of Book objects or Author objects, as needed. Then, like before, it has a `RouteDictionary` property to hold the current sorting, paging, and filtering route segments. In addition, it has a `TotalPages` property to hold the total number of paging links needed.

The next two examples in this figure show the List() action methods of the Book controller and of the Author controller. Before, the Book controller's List() action method had to retrieve data for the filter drop-down lists. Now, because that's done by view components, this method works more like the List() action method of the Author controller. Both methods initialize the `GridBuilder` and `QueryOptions` objects they need to get the route data from session state and the list data from the database. (Due to space considerations, this code isn't shown here.) Then, they each create a `GridViewModel` object of the correct type, assign the appropriate values, and pass it to the view.

## The generic GridViewModel class

```
public class GridViewModel<T>
{
    public IEnumerable<T> Items { get; set; }
    public RouteDictionary CurrentRoute { get; set; }
    public int TotalPages { get; set; }
}
```

## The List() action method of the Book controller

```
public ViewResult List(BooksGridDTO values)
{
    /* code that creates GridBuilder and QueryOptions objects to
     * retrieve a list of paged, sorted, and filtered books */

    var vm = new GridViewModel<Book> {
        Items = data.List(options),
        CurrentRoute = builder.CurrentRoute,
        TotalPages = builder.GetTotalPages(data.Count)
    };

    return View(vm);
}
```

## The List() action method of the Author controller

```
public ViewResult List(GridDTO vals)
{
    /* code that creates GridBuilder and QueryOptions objects to
     * retrieve a list of paged and sorted authors */

    var vm = new GridViewModel<Author> {
        Items = data.List(options),
        CurrentRoute = builder.CurrentRoute,
        TotalPages = builder.GetTotalPages(data.Count)
    };

    return View(vm);
}
```

## Description

- The use of view components can simplify your controllers and view models. That's because the controllers and view models no longer need to perform the tasks that are handled by the view components. Instead, the controllers and view models can focus on the primary task that's handled by the view.

---

Figure 15-17 How view components can simplify an app

## The Bookstore app

---

The previous figure showed how view components can simplify the controllers and view models for the Bookstore app. Now, this chapter finishes by showing how view components, partial views, and custom tag helpers can simplify the layout and Book/List view of the Bookstore app.

### The Book Catalog page

---

Figure 15-18 presents the Book Catalog page of the Bookstore app. This page gets its data from the List() action method of the Book controller presented in the previous figure. Although this page looks the same as the Book Catalog page from chapter 13, its code uses custom tag helpers, partial views, and view components as shown by the numbered items.

The first item shows the links in the Bootstrap navbar, with the Books link marked as active. In chapter 13, the app marked active links by using Razor code to call a static method in the model. Now, this is handled by the ActiveNavbar tag helper shown in the next figure.

The second item shows a Bootstrap badge with a value that indicates how many items are in the cart. In chapter 13, the app displayed the item count with Razor code that retrieved cart data from session state. In chapter 14, it handled this by injecting an ICart object into the layout. Now, the CartBadge view component retrieves its own Cart data so the layout doesn't have to.

The third item shows an `<h4>` element that displays a message. In chapter 13, the app handled this by using Razor code to check for a TempData message and display the message if it existed. Now, it's handled by the TempMessage tag helper.

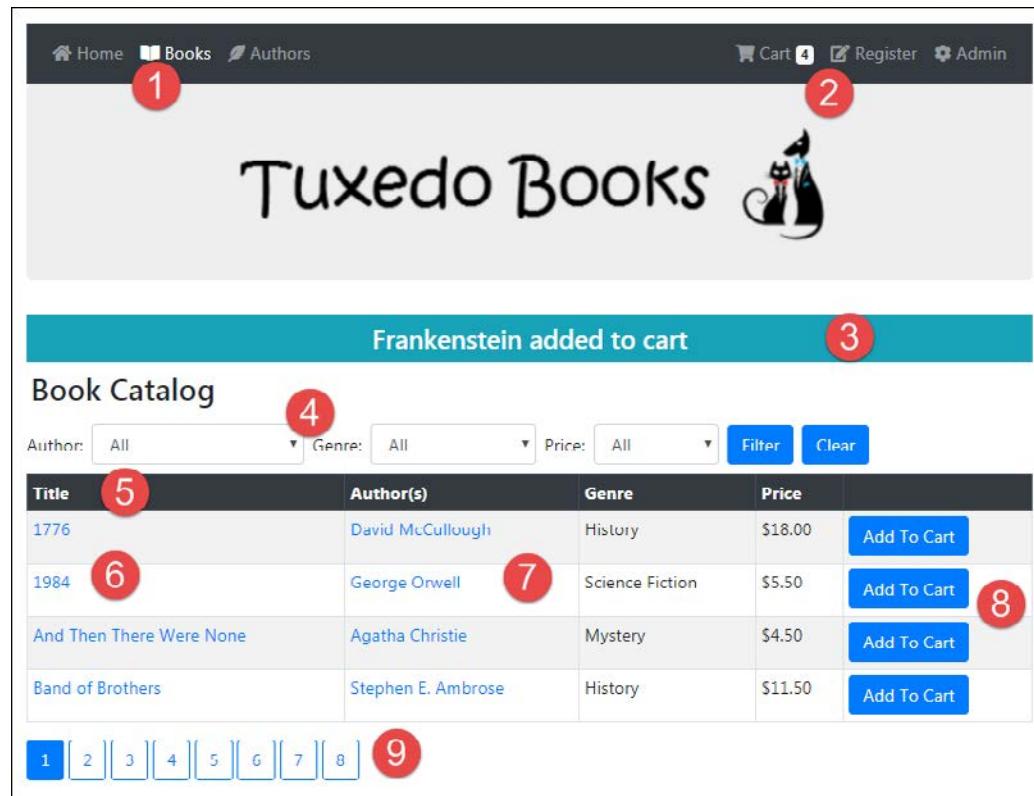
The fourth item shows a drop-down list for filtering books by author, as well as drop-down lists for filtering by genre and price. In chapter 13, the Book controller passed the author, genre, and price data for these elements to the view, and the view created them using repetitive HTML and Razor code. Now, they're handled by view components, each of which gets its own data and uses the same partial view. This reduces code duplication.

The fifth item shows links for sorting books by title, genre, and price. In chapter 13, the view created these links using repetitive HTML and Razor code. Now, they're created by SortingLink tag helpers, which output the link URLs and reduce code duplication.

The sixth and seventh callouts show links to details pages for books and for authors. In chapter 13, these links were coded with standard HTML and built-in tag helpers. However, because details links appear on several pages in this app, this caused code duplication. Now, these links are handled by partial views, which reduces code duplication.

The eighth callout shows `<button>` elements that are styled with Bootstrap CSS classes. Before, the app hardcoded the Bootstrap classes wherever a `<button>` or `<a>` element needed them. Now, the Button tag helper automatically applies the Bootstrap classes to specified elements.

## The Book Catalog page



## The custom tag helpers, partial views, and view components in this page

1. The ActiveNavbar tag helper from figure 15-10, updated as shown in the next figure.
2. The CartBadge view component from figure 15-15.
3. The TempMessage tag helper from figure 15-11.
4. The AuthorDropDown view component from 15-16. The other drop-down lists shown here are view components that work similarly.
5. A SortingLink tag helper that's similar to the PagingLink tag helper from figure 15-12.
6. The \_BookLink partial view from figure 15-14.
7. An \_AuthorLink partial view that works similarly to the \_BookLink partial view.
8. The Button tag helper from figure 15-6.
9. The paging-links partial view, containing the paging-link tag helper from figure 15-12.

## Description

- This version of the Bookstore app uses custom tag helpers, partial views, and view components to simplify code and reduce code duplication.

Figure 15-18 The Book Catalog page of the Bookstore app

This reduces code duplication, and it also makes it easier to change the Bootstrap classes for all specified elements if you ever need to do that.

The ninth callout shows link buttons for paging through the books. In chapter 13, the view created these links using repetitive HTML and Razor code. Now, they're created by PagingLink tag helpers, which output the link URLs and reduce code duplication. In addition, the tag helpers are coded within a partial view, so it's easy to add the code to a page.

## The updated ActiveNavbar tag helper

---

The Bookstore app has two Bootstrap navbars. One is in the main layout and the other is in the nested layout in the Admin area. The ActiveNavbar tag helper presented in figure 15-10 marks the correct links as active in both of these navbars.

Unfortunately, this tag helper doesn't mark the Admin link as active for all the pages in the Admin area. For instance, when you navigate to the Admin area, the Admin link in the main layout and the Manage Books link are both marked as active. This is what you want. However, when you click on either the Manage Authors or Manage Genres links, the Admin link is greyed out, like the rest of the nav links in the main layout. This is *not* what you want.

The ActiveNavbar tag helper behaves like this because it looks for a match for both the area and controller names. For the Admin link and the Manage Books link, this works because it matches the area value of "Admin" and the controller value of "Book". For the Manage Author and Manage Genre links, however, the controller value doesn't match.

To fix this, you can tell the tag helper that the Admin link in the main layout should be marked active when only the area value matches. To do that, the app updates the tag helper class to add a Boolean property named IsAreaOnly as shown in figure 15-19. Then, the Process() method adds an else if block to the if statement. Within this block, the code checks whether the IsAreaOnly value is true and the current area and the asp-area match. If so, the code marks the Admin link as active.

As a result, if the Admin link includes the my-mark-area-active attribute, this tag helper marks both the Admin link and the Manage link as active on every page in the Admin area. For instance, the screen in this figure shows the Manage Authors page with both the Admin link and the Manage Authors link marked as active.

## The ActiveNavbar tag helper updated to account for the Admin area

```
[HtmlTargetElement("a", Attributes = "[class=nav-link]", ParentTag = "li")]
public class ActiveNavbarTagHelper : TagHelper
{
    [ViewContext]
    [HtmlAttributeNotBound]
    public ViewContext ViewCtx { get; set; }

    [HtmlAttributeName("my-mark-area-active")]
    public bool IsAreaOnly { get; set; }

    public override void Process(TagHelperContext context,
        TagHelperOutput output)
    {
        string area = ViewCtx.RouteData.Values["area"]?.ToString() ?? "";
        string ctrlr = ViewCtx.RouteData.Values["controller"].ToString();

        string aspArea = context.AllAttributes["asp-area"]?.Value?
            .ToString() ?? "";
        string aspCtrlr = context.AllAttributes["asp-controller"].Value
            .ToString();

        if (area == aspArea && ctrlr == aspCtrlr)
            output.Attributes.AppendCssClass("active");
        else if (IsAreaOnly && area == aspArea)
            output.Attributes.AppendCssClass("active");
    }
}
```

## The two active navbar links on the Manage Authors page of the Admin area



## Description

- The Bookstore app uses an updated version of the ActiveNavbar tag helper.
- This tag helper adds a Boolean property that allows you to identify navbar links that should be marked active when just the area names match.

Figure 15-19 The updated ActiveNavbar tag helper

## The layout

---

Figure 15-20 shows the layout for the Bookstore app. This updated layout no longer uses any Razor code blocks. That's because it no longer needs to retrieve or manipulate any data, since that's all handled by the tag helpers and view components now.

The body of the layout has a `<nav>` element that contains the HTML necessary for a Bootstrap navbar. It uses a partial view for the HTML for the navbar's menu button. This keeps the layout cleaner.

Within the navbar, the navigation links are `<a>` elements that have a `class` attribute with a value of "nav-link" and a parent tag that's an `<li>` element. That's exactly what the `ActiveNavbar` tag helper presented in this chapter targets. As a result, no additional attributes need to be added to most of the navigation links for the tag helper to work.

The only exception is the navigation link for the Admin area. This link needs to tell the tag helper to mark it active when only area names match. To do that, it includes the `my-mark-area-active` attribute. Since it's a Boolean value, the code only needs to include the attribute name. In other words, it's not necessary to specify a value for this attribute. However, it could also be coded with a `true` value like this:

```
my-mark-area-active="true"
```

Within the start and end tags for the navigation link for the cart, a `<span>` element displays an icon for the cart. Then, the code uses the `Cart Badge` view component to display a badge with the number of items in the cart. This creates another `<span>` element styled as a Bootstrap badge.

Within the main section of the body, the `TempMessage` tag helper displays any message that's stored in `TempData`. Since this is a conditional tag helper, this non-standard element is either transformed to a standard HTML `<h4>` element or suppressed and not output at all.

## The layout

```
<!DOCTYPE html>
<html lang="en">
<head>...</head>
<body>
    <div class="container">
        <nav class="navbar navbar-expand-md navbar-dark bg-dark">
            <partial name="_NavbarMenuButtonPartial" />
            <div class="collapse navbar-collapse" id="menu">
                <ul class="navbar-nav mr-auto">
                    <li class="nav-item">
                        <a class="nav-link" asp-action="Index"
                           asp-controller="Home" asp-area="">
                            <span class="fas fa-home"></span>&ampnbspHome
                        </a>
                    </li>
                    @* nav item links for Books and Authors go here *@
                </ul>
                <ul class="navbar-nav ml-auto">
                    <li class="nav-item">
                        <a class="nav-link" asp-action="Index"
                           asp-controller="Cart" asp-area="">
                            <span class="fas fa-shopping-cart"></span>
                            &ampnbspCart
                            <vc:cart-badge></vc:cart-badge>
                        </a>
                    </li>
                    @* nav item link for Registration goes here *@
                    <li class="nav-item">
                        <a class="nav-link" asp-action="Index"
                           asp-controller="Book" asp-area="Admin"
                           my-mark-area-active>
                            <span class="fas fa-cog"></span>&ampnbspAdmin
                        </a>
                    </li>
                </ul>
            </div>
        </nav>

        <header class="jumbotron text-center">
            <a asp-action="Index" asp-controller="Home">
                
            </a>
        </header>

        <main>
            <my-temp-message />
            @RenderBody()
        </main>
    </div>
    <script src="~/lib/jquery/dist/jquery.min.js"></script>
    <script src="~/lib/bootstrap/dist/js/bootstrap.bundle.min.js"></script>
    <script src="~/js/site.js" asp-append-version="true"></script>
    @RenderSection("Scripts", required: false)
</body>
</html>
```

---

Figure 15-20 The layout of the Bookstore app

## The Book/List view

---

Figure 15-21 shows the Book/List view for the Bookstore app. It has two `<form>` elements. The first one contains filter drop-down lists and posts to the `Filter()` action method, and the second one contains the Add to Cart buttons that post to the `Add()` action method.

The filter form has three view components. The `AuthorDropDown` view component creates the `<select>` element for filtering by author, the `GenreDropDown` view component creates the `<select>` element for filtering by genre, and the `PriceDropDown` view component creates the `<select>` element for filtering by price. Each of these view components gets its own data from the database, and each accepts a value from the view that indicates which item to mark as selected.

The filter form also has two submit buttons. Neither of these buttons uses Bootstrap classes. However, the `Button` tag helper applies to any element that has a `type` attribute with a value of “submit”, and it provides Bootstrap classes to style buttons. As a result, you can have `<button>` elements like this throughout your app, and the tag helper applies the styles. Because the styling for all of these buttons is stored in the same place, this makes it easy to change the styling of your buttons.

The add form has three `SortingLink` tag helpers that create links that allow the user to sort books by title, genre, and price. Each tag helper accepts a value from the view that indicates what field the link should sort by. In addition, each tag helper accepts the value of the `CurrentRoute` property of the `GridViewModel<Book>` model object, which the tag helper uses to build the appropriate URL for the link.

The add form also includes partial views named `_BookLinkPartial` and `_AuthorLinkPartial`. Both of these partial views add links to a details page. The book partial view needs a `Book` object as a model, and the author partial view needs an `Author` object as a model. Since the model object of the Book/List view is a `GridViewModel<Book>` object, the code can use the `model` attribute to pass an appropriate model object to each partial view.

Like the filter form, the add form also has a button with a `type` attribute of “submit”. Again, the `Button` tag helper applies the styling for this button.

Below the add form, the `_PagingLinksPartial` view uses the data in the view’s model to create the links for paging through the books. Here, the model for this partial view is the same as the model for the parent view. As a result, there’s no need to use the `model` attribute of the `<partial>` element to specify a different model.

## The Book/List view

```
@model GridViewModel<Book>
...
<form asp-action="Filter" method="post" class="form-inline">
    <label>Author: </label>
    <vc:author-drop-down selected-value="@Model.CurrentRoute.AuthorFilter">
    </vc:author-drop-down>

    <label>Genre: </label>
    <vc:genre-drop-down selected-value="@Model.CurrentRoute.GenreFilter">
    </vc:genre-drop-down>

    <label>Price: </label>
    <vc:price-drop-down selected-value="@Model.CurrentRoute.PriceFilter">
    </vc:price-drop-down>

    <button type="submit" class="mr-2">Filter</button>
    <button type="submit" name="clear" value="true">Clear</button>
</form>

<form asp-action="Add" asp-controller="Cart" method="post">
    <table class="table table-bordered table-striped table-sm">
        <thead class="thead-dark">
            <tr>
                <th>
                    <my-sorting-link sort-field="Title"
                        current="@Model.CurrentRoute">Title</my-sorting-link></th>
                <th>Author(s)</th>
                <th>
                    <my-sorting-link sort-field="Genre"
                        current="@Model.CurrentRoute">Genre</my-sorting-link></th>
                <th>
                    <my-sorting-link sort-field="Price"
                        current="@Model.CurrentRoute">Price</my-sorting-link></th>
                <th></th>
            </tr>
        </thead>
        <tbody>
            @foreach (Book book in Model.Items) {
                <tr>
                    <td><partial name="_BookLinkPartial" model="@book" /></td>
                    <td>
                        @foreach (var ba in book.BookAuthors) {
                            <p><partial name="_AuthorLinkPartial"
                                model="@ba.Author" /></p>
                        }
                    </td>
                    <td>@book.Genre?.Name</td>
                    <td>@book.Price.ToString("c")</td>
                    <td><button type="submit" name="id" value="@book.BookId">
                        Add To Cart</button></td>
                </tr>
            }
        </tbody>
    </table>
</form>
<partial name="_PagingLinksPartial" />
```

Figure 15-21 The Book/List view of the Bookstore app

## Perspective

This chapter has shown how to create and use custom tag helpers, partial views, and view components. You can use these features to reduce code duplication in your views, which makes your views more flexible and easier to maintain. In addition, you can use these features to simplify the code for your controllers and view models.

## Terms

partial view  
view component

## Summary

- A *partial view* can contain HTML and Razor code that can be used in multiple views within an app or multiple apps.
- A *view component* is a class that sends data to a partial view. You can think of a view component as a controller for a partial view.

### Exercise 15-1 Add some custom tag helpers and a view component

In this exercise, you'll add some custom tag helpers to the Class Schedule web app from the exercises for chapter 12. Then, you'll add a view component.

#### Run the app and review the HTML

1. Open the Ch15Ex1ClassSchedule app in the ex\_starts directory.
2. If you didn't do the exercise for chapter 12 or 14, open the Package Manager Console and enter the Update-Database command to create the database for this app.
3. Run the app. On the first page, note that the links for the days aren't working. Also, note that the Edit and Delete links to the right of each class aren't working.
4. Click the Add Class link. On the Add Class page, note that the Cancel link doesn't work and the Add button doesn't have any styling.
5. In the Views folder, open the Home/Index view. Note that this code uses a <my-link-button> element for the day of the week links. This element is not standard HTML. In addition, note that the Edit and Delete links uses the same non-standard <my-link-button> element.
6. Open the Class/Add view. Note that the Add submit button doesn't specify Bootstrap CSS classes for styling, and the Cancel link uses a non-standard <my-link-button> element.

### Add a tag helper to style submit buttons

7. Open the \_ViewImports.cshtml file and add an @addTagHelper directive that registers all custom tag helpers for this web app.
8. Open the TagHelpers folder and review the code in the ExtensionMethods.cs file.
9. Add a class named SubmitButtonTagHelper that inherits the TagHelper class.
10. Within this class, add an HtmlTargetElement attribute that applies this tag helper to all submit buttons.
11. Override the Process() method and add code that uses the AppendCssClass() extension method to apply the Bootstrap btn and btn-dark classes.
12. Run the app and view the Add buttons on the Add Class and Add Teacher pages. They should be styled correctly now.

### Add a tag helper to transform the <my-link-button> element

13. In the TagHelpers folder, add a class named MyLinkButtonTagHelper that inherits the TagHelper class.
14. Within this class, add public string properties named Action, Controller, and Id as well as a public Boolean property named IsActive.
15. Update the class to receive a LinkGenerator object and a ViewContext object via dependency injection.
16. In the Process() method, assign the value of the Action property to a string variable. Or, if the Action property is null, assign the value of the “action” item of the RouteData.Values collection.
17. Repeat the previous step for the Controller property.
18. Create an anonymous object for the id route segment, and assign it to a variable. Or, if the Id property is null, assign a null value to the variable.
19. Use the action, controller, and id variables with the LinkGenerator object to create a URL, and assign this URL to a string variable.
20. Assign Bootstrap CSS classes to a string variable. If the value of the IsActive property is true, assign the btn and btn-dark classes. Otherwise, assign the btn and btn-outline-dark classes.
21. Use the variables for the URL and the CSS classes with the BuildLink() extension method to transform a non-standard <my-link-button> element to a standard <a> element.
22. Run the app. Then, view the link buttons on the various pages. If you hover your mouse over a link button, the app should display the URL that the tag helper generated for it. And if you click on a link, it should work correctly.

**Add a view component for the link buttons for the days of the week**

23. Add a folder named Components.
24. In the Components folder, add a new class named DayFilter that inherits the ViewComponent class.
25. Update the class to receive an IRepository<Day> object via dependency injection. To do that, you can use the Teacher controller as a guide.
26. Add an Invoke() method that returns an IViewComponentResult object.
27. Add code to the Invoke() method that uses the IRepository<Day> object to get a collection of Day objects sorted by DayId. Use the View() method to pass this collection to the partial view.
28. Create a folder for the component's partial view. Make sure to create this folder in the location where MVC expects it.
29. In this folder, add a new partial view named Default.cshtml.
30. Declare the model of the partial view to be an IEnumerable<Day> object.
31. Add a Razor code block to the partial view.
32. Move the code that gets the id value of the current route from the Razor code block of the Home/Index view to the Razor code block in the partial view.
33. Move the Razor foreach loop from the Home/Index view to this partial view. Change the loop condition to loop through the Days collection that's stored in the model rather than the collection that's stored in the ViewBag.
34. Build the solution. Then, use tag helper syntax to add the view component to the Home/Index view where the foreach loop used to be. Note how this simplifies the view code.
35. Open the HomeController class and review its code. This controller no longer needs to send Day objects to the view. As a result, you can remove all of that code. Also, you can modify this class to use a Repository<Class> object, instead of the unit of work object. Note how this simplifies the controller code.
36. Run the app and make sure the day link buttons still work correctly.

# How to authenticate and authorize users

The Bookstore app that you've seen in previous chapters allows any user to access any of its pages. Now, this chapter shows how to restrict access to some pages so only authorized users can access those pages. To provide this functionality, this chapter shows how to use ASP.NET Core Identity.

<b>An introduction to authentication .....</b>	<b>650</b>
Three types of authentication .....	650
How individual user account authentication works.....	652
An introduction to ASP.NET Identity .....	654
How to restrict access to controllers and actions .....	656
<b>How to get started with Identity.....</b>	<b>658</b>
How to add Identity classes to the DB context.....	658
How to add Identity tables to the database.....	660
How to configure the middleware for Identity .....	662
How to add Log In/Out buttons and links to the layout.....	664
How to start the Account controller .....	666
<b>How to register a user.....</b>	<b>668</b>
The Register view model .....	668
The Account/Register view .....	670
The Register() action method for POST requests .....	672
The LogOut() action method for POST requests .....	672
<b>How to log in a user .....</b>	<b>674</b>
The Login view model.....	674
The Account/Login view .....	676
The LogIn() action method for POST requests.....	678
<b>How to work with roles .....</b>	<b>680</b>
Properties and methods for working with roles .....	680
The User entity and view model.....	682
The User controller and its Index() action method .....	684
The User/Index view .....	686
Other action methods of the User controller .....	690
The code that restricts access .....	692
How to seed roles and users.....	694
<b>More skills for working with Identity .....</b>	<b>696</b>
How to change a user's password .....	696
How to add more user registration fields.....	698
<b>Perspective .....</b>	<b>700</b>

## An introduction to authentication

---

If you want to limit access to all or part of your ASP.NET app to certain users, you can use *authentication* to verify each user's identity. Then, once you have authenticated the user, you can use *authorization* to check if the user has the appropriate privileges for accessing a page. That way, you can prevent unauthorized users from accessing pages that they shouldn't be able to access.

### Three types of authentication

---

Figure 16-1 describes three types of authentication you can use in ASP.NET apps. The first, called *Windows-based authentication*, requires that you set up a Windows user account for each user. Then, you use standard Windows security features to restrict access to all or part of the app. When a user attempts to access the app, Windows displays a login dialog box that asks the user to supply the username and password of the Windows account. This type of authentication is most appropriate for a local setting like a company intranet.

To use *individual user account authentication*, you have a login page that typically requires the user to enter a username and password. Then, ASP.NET displays this page automatically when it needs to authenticate a user who's trying to access the app. This type of authentication works well with web apps, and it's the type of authentication that's presented in this chapter.

In recent years, authentication services offered by third parties such as Google, Facebook, and others have also become popular. This type of authentication provides several advantages, but showing how to use these services is beyond the scope of this book.

## Windows-based authentication

- Causes the browser to display a login dialog box when the user attempts to access a restricted page.
- Is supported by most browsers.
- Is configured through the IIS management console.
- Uses Windows user accounts and directory rights to grant access to restricted pages.
- Is most appropriate for an intranet app.

## Individual user account authentication

- Allows developers to code a login page that gets the username and password.
- Encrypts the username and password entered by the user if the login page uses a secure connection.
- Doesn't rely on Windows user accounts.

## Third-party authentication services

- Is provided by third parties such as Google, Facebook, Twitter, and Microsoft using technologies like OpenID and OAuth.
- Allows users to use their existing logins and frees developers from having to worry about the secure storage of user credentials.
- Can issue identities or accept identities from other web apps and access user data on other services.
- Can use two-factor authentication.

## Description

- *Authentication* refers to the process of validating the identity of a user so the user can be granted access to an app. A user must typically supply a username and password to be authenticated.
- After a user is authenticated, the user must still be authorized to use the requested app. The process of granting user access to an app is called *authorization*.

---

Figure 16-1 Three types of authentication

## How individual user account authentication works

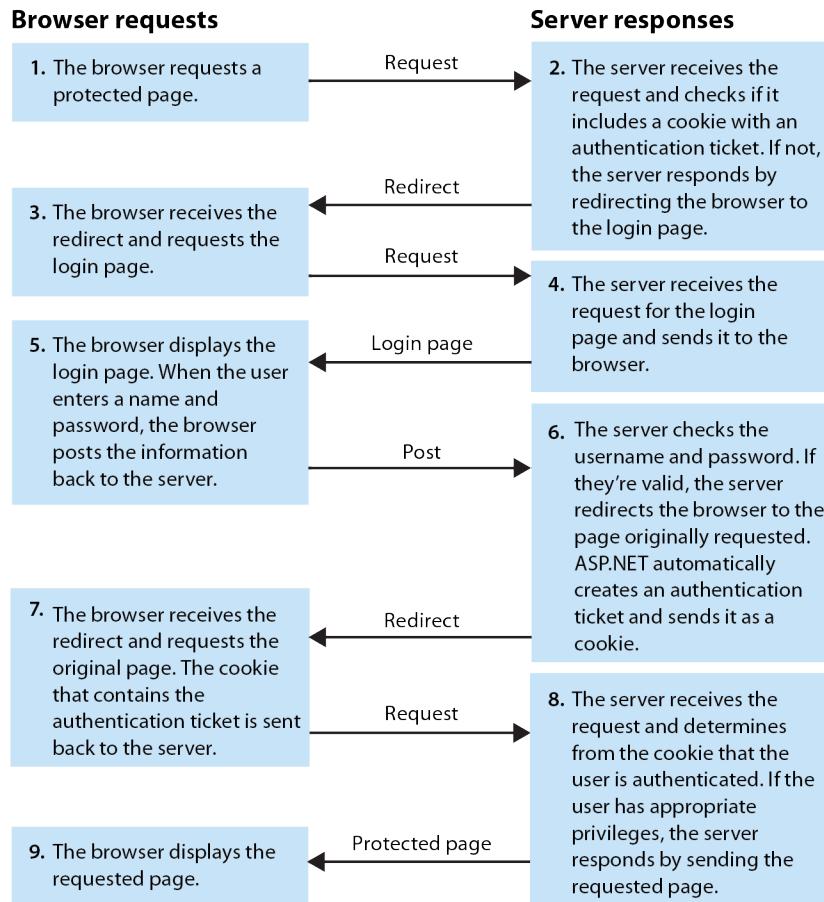
---

To help you understand how individual user account authorization works, figure 16-2 shows a typical series of exchanges that occur between a web browser and a server when a user attempts to access a page that uses individual user accounts. The authentication process begins when a user requests the page. When the server receives the request, it checks whether the user has already been authenticated. To do that, it looks for an *authentication cookie* in the request for the page. If it doesn't find the cookie, it redirects the browser to the login page.

At the login page, the user enters a username and password and posts the login page back to the server. Then, if the server finds the username/password combination in the database, they are valid. In that case, the server creates an authentication cookie and redirects the browser back to the original page. As a result, when the browser requests the original page, it sends the cookie back to the server. This time, the server sees that the user has been authenticated, so it checks whether the user is authorized to view the protected page. If so, the server sends the requested page back to the browser. Otherwise, it sends a page that indicates that access was denied.

By default, the server sends the authentication cookie as a session cookie. In that case, the server only authenticates the user for the current session. However, the user can often specify that the cookie should be sent as a persistent cookie. Then, the browser automatically sends the authentication cookie for future sessions, until the cookie expires.

## HTTP requests and responses with individual user account authentication



### Description

- When ASP.NET Core receives a request for a protected page from a user who has not been authenticated, the server redirects the user to the login page.
- To be authenticated, the user request must contain an *authentication cookie*. By default, this cookie is stored as a session cookie.
- ASP.NET Core automatically creates an authentication cookie when the app indicates that the user should be authenticated. ASP.NET Core checks for the presence of an authentication cookie any time it receives a request for a restricted page.
- The user can often specify that the authentication cookie should be made persistent. Then, the browser automatically sends the authentication cookie for future sessions, until the cookie expires.

Figure 16-2 How individual user account authentication works

## An introduction to ASP.NET Identity

---

In previous versions of ASP.NET, the Membership system provided authentication for ASP.NET apps. However, this system was difficult to customize. In addition, it was hard or impossible to modify it to work with other data stores, devices, or third-party authentication providers. In response to these issues, Microsoft replaced the Membership system with the ASP.NET Identity system. Figure 16-3 describes some of the benefits of the Identity system.

To start, you can use it with all of the ASP.NET frameworks, including Web Forms and MVC. In addition, Identity is easier to customize, unit test, and update than the Membership system. It also supports claims-based authentication, which is a sophisticated form of authentication that can be more flexible than authorizing users based on roles as described later in this chapter.

Finally, it uses middleware called *OWIN*, or *Open Web Interface for .NET*. OWIN is an open-source project that defines a standard interface between .NET web servers and web apps. The goal of OWIN is to create lightweight components with as few dependencies on other frameworks as possible.

This figure also presents some of the main classes that ASP.NET Identity provides. These classes allow you to work with users and roles. A *role* lets you apply the same access rules to a group of users.

## Some classes provided by ASP.NET Identity

Class	Description
<code>IdentityDbContext</code>	An Entity Framework DbContext object for working with the tables of the ASP.NET Identity system.
<code>IdentityUser</code>	Represents a user.
<code>IdentityRole</code>	Represents a role.
<code>UserManager</code>	Provides methods for working with users.
<code>RoleManager</code>	Provides methods for working with roles.
<code>SignInManager</code>	Provides methods for signing in users.
<code>IdentityResult</code>	Represents the result of an identity operation.

## Some benefits of Identity

- It can be used with all ASP.NET frameworks, including MVC, Web Forms, Web API, and SignalR to build web, phone, Windows Store, or hybrid apps.
- You have control over the schema of the data store that holds user information, and you can change the storage system from the default of SQL Server.
- It's modular, so it's easier to unit test.
- It supports claims-based authentication, which can be more flexible than using simple roles.
- It supports third-party authentication providers like Google, Facebook, Twitter, and Microsoft.
- It's based on *OWIN (Open Web Interface for .NET)* middleware, which is an open-source project that defines a standard interface between .NET web servers and web apps.
- It's distributed as a NuGet package, so Microsoft can deliver new features and bug fixes faster than before.

## Description

- The Identity system replaces the Membership system and can be used with all ASP.NET frameworks.
- *Roles* let you apply the same access rules to a group of users.

---

Figure 16-3 An introduction to ASP.NET Identity

## How to restrict access to controllers and actions

---

Figure 16-4 shows how to restrict access to the pages of a web app. To do that, you can apply the attributes in the Authorization namespace to an entire controller or to individual action methods. Since the Authorization namespace isn't part of the Identity package, you can use the Authorization attributes to restrict access to your web pages before you use Identity to provide a way to let authorized users access those pages.

To restrict access to all action methods in a controller, you can decorate the class declaration with an Authorization attribute. For instance, the first example decorates the CartController class with the Authorize attribute. This restricts access to all actions in the Cart controller to users who are logged in. However, the first example doesn't require the user to be a member of any role. By contrast, the second example uses the Roles parameter of the Authorize attribute to only grant access to users in the Admin role.

Although it's common to apply Authorization attributes to the entire controller, you may sometimes need to apply Authorization attributes to a specific action method. To do that, you decorate the declaration for the action method with the appropriate Authorization attribute. For instance, the third example decorates the List() action method with the AllowAnonymous attribute. As a result, all users can access this method. This includes authenticated users who have logged in as well as anonymous users who haven't logged in.

In addition, this example decorates the Add() action method with the Authorize attribute. As a result, only users who are logged in can access this method. Similarly, this example decorates the Delete() action method with the Authorize attribute for users in the Admin role. As a result, only users who are logged in and belong to the Admin role can access this method.

## Attributes for authorization

Attribute	Description
<code>AllowAnonymous</code>	Grants access to all users.
<code>Authorize</code>	Grants access only to logged in users.
<code>Authorize(Roles = "r1, r2")</code>	Grants access only to logged in users that belong to the specified roles.

### The using directive for the Authorization namespace

```
using Microsoft.AspNetCore.Authorization;
```

### Code that only allows logged in users to access an entire controller

```
[Authorize]
public class CartController : Controller {
    ...
}
```

### Code that only allows logged in users in the Admin role to access an entire controller

```
[Authorize(Roles = "Admin")]
public class BookController : Controller {
    ...
}
```

### Code that applies different attributes to different action methods

```
public class ProductController : Controller {
    ...
    [AllowAnonymous]
    [HttpGet]
    public IActionResult List() {
        ...
    }

    [Authorize]
    [HttpGet]
    public IActionResult Add() {
        ...
    }

    [Authorize(Roles = "Admin")]
    [HttpGet]
    public IActionResult Delete(int id) {
        ...
    }
    ...
}
```

### Description

- To restrict access to the pages of a web app, you can apply the attributes in the Authorization namespace to an entire controller or to individual action methods.
- The Authorization attributes are not part of the Identity package, so they work even if you haven't added the Identity package to your app.

Figure 16-4 How to restrict access to controllers and actions

## How to get started with Identity

Now that you've been introduced to some of the basic concepts of authentication and authorization, you're ready to get started with Identity. In particular, this topic shows how to get started with ASP.NET Core Identity.

### How to add Identity classes to the DB context

Before you can work with Identity, you need to add its NuGet package to your project. To do that, you can use the same skills you use to add other NuGet packages. For example, chapter 4 showed how to install the NuGet package for EF Core. Now, figure 16-5 just shows the name of the NuGet package that you need to use ASP.NET Core Identity with EF Core.

Once you've installed the NuGet package for Identity, you can code a User entity class like the one shown in this figure. Since this class inherits the IdentityUser class, it automatically has access to all of the properties defined by the IdentityUser class. As a result, you don't need to code any additional properties if you only want to use properties of the IdentityUser class such as UserName, Email, Password, and ConfirmPassword.

Next, you can add the User entity class to the DB context class for the app. To do that, you must make sure that the DB context class inherits the IdentityDbContext<User> class, not the standard DbContext class. That way, EF knows to create all of the tables that are needed by Identity. Finally, in the OnModelCreating() method, you must make sure to pass the ModelBuilder object to the same method of the base class. That's because the IdentityDbContext<T> class provides an implementation of this method that creates the Identity tables. If you don't call this method, you'll get errors when you attempt to update the database.

## The NuGet package for Identity with EF Core

`Microsoft.AspNetCore.Identity.EntityFrameworkCore`

### Some properties of the `IdentityUser` class

Property	Description
<code>UserName</code>	The username for the user.
<code>Password</code>	The password for the user.
<code>ConfirmPassword</code>	Used to confirm that the password was entered correctly.
<code>Email</code>	The email address for the user.
<code>EmailConfirmed</code>	Used to confirm that the email was entered correctly.
<code>PhoneNumber</code>	The phone number for the user.
<code>PhoneNumberConfirmed</code>	Used to confirm that the phone number was entered correctly.

### The `User` entity class

```
using Microsoft.AspNetCore.Identity;

namespace Bookstore.Models
{
    public class User : IdentityUser {
        // Inherits all IdentityUser properties
    }
}
```

### The Bookstore context class

```
...
using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;

namespace Bookstore.Models
{
    public class BookstoreContext : IdentityDbContext<User>
    {
        public BookstoreContext(DbContextOptions<BookstoreContext> options)
            : base(options) { }

        public DbSet<Author> Authors { get; set; }
        public DbSet<Book> Books { get; set; }
        public DbSet<BookAuthor> BookAuthors { get; set; }
        public DbSet<Genre> Genres { get; set; }

        protected override void OnModelCreating(ModelBuilder modelBuilder)
        {
            base.OnModelCreating();

            // BookAuthor: set primary key
            modelBuilder.Entity<BookAuthor>()
                .HasKey(ba => new { ba.BookId, ba.AuthorId });
            ...
        }
    }
}
```

Figure 16-5 How to add Identity classes to the DB context

## How to add Identity tables to the database

---

For Identity to work, you must create all the tables it needs in your database. These tables include the AspNetUsers table and the AspNetRoles table. To do that, you can follow the procedure at the top of figure 16-6. Here, the second step creates a migration file that contains the code for adding the tables. Then, the third step updates the database by executing the code in the migration file.

If you code the User entity and DB context classes correctly, the Add-Migration command should generate a migration that creates several tables. For instance, this figure shows the start of the Up() method of the generated migration class. This method begins by creating a table named AspNetRoles that stores the roles for the app. Then, it creates a table named AspNetUsers that stores the users for the app.

Both of these tables contain the basic columns (Id, Name, UserName, Password, and so on) that your app needs to work with simple authorization scenarios. In addition, they contain extra columns (TwoFactorEnabled, LockoutEnabled, and so on) that you may need later for more advanced authorization scenarios. To make sure the Update-Database command executed successfully, you can use the SQL Server Object Explorer or another comparable tool to view the tables for your database. If you can view the Identity tables (AspNetRoles, AspNetUsers, and so on) and their columns, the database has been updated successfully.

## A procedure for adding Identity tables to the database

1. Start the Package Manager Console (PMC).
2. Add a migration that adds the tables by entering a command like this one:  
`Add-Migration AddIdentityTables`
3. Update the database by entering a command like this one:  
`Update-Database`

## Some of the Up() method of the generated migration class

```
protected override void Up(MigrationBuilder migrationBuilder)
{
    migrationBuilder.CreateTable(
        name: "AspNetRoles",
        columns: table => new
        {
            Id = table.Column<string>(nullable: false),
            Name = table.Column<string>(maxLength: 256, nullable: true),
            NormalizedName = table.Column<string>(maxLength: 256, nullable: true),
            ConcurrencyStamp = table.Column<string>(nullable: true)
        },
        constraints: table =>
        {
            table.PrimaryKey("PK_AspNetRoles", x => x.Id);
        });
    migrationBuilder.CreateTable(
        name: "AspNetUsers",
        columns: table => new
        {
            Id = table.Column<string>(nullable: false),
            UserName = table.Column<string>(maxLength: 256, nullable: true),
            NormalizedUserName = table.Column<string>(maxLength: 256, nullable: true),
            Email = table.Column<string>(maxLength: 256, nullable: true),
            NormalizedEmail = table.Column<string>(maxLength: 256, nullable: true),
            EmailConfirmed = table.Column<bool>(nullable: false),
            PasswordHash = table.Column<string>(nullable: true),
            SecurityStamp = table.Column<string>(nullable: true),
            ConcurrencyStamp = table.Column<string>(nullable: true),
            PhoneNumber = table.Column<string>(nullable: true),
            PhoneNumberConfirmed = table.Column<bool>(nullable: false),
            TwoFactorEnabled = table.Column<bool>(nullable: false),
            LockoutEnd = table.Column<DateTimeOffset>(nullable: true),
            LockoutEnabled = table.Column<bool>(nullable: false),
            AccessFailedCount = table.Column<int>(nullable: false),
            Discriminator = table.Column<string>(nullable: false)
        },
        constraints: table =>
        {
            table.PrimaryKey("PK_AspNetUsers", x => x.Id);
        });
    ...
}
```

## Description

- For Identity to work, you must create all the tables it needs in your database. These tables include the AspNetUsers table and the AspNetRoles table.

---

Figure 16-6 How to add Identity tables to the database

## How to configure the middleware for Identity

---

Figure 16-7 shows how to configure the middleware for Identity. To do that, you can add the code shown in this figure to the Startup.cs class. To start, you can add a using directive for the Identity namespace as shown by the first example.

In the ConfigureServices() method, you can add a statement that adds the Identity service as shown by the second example. This example adds an Identity service with a user defined by the User class shown earlier in this chapter and a role defined by the built-in IdentityRole class mentioned earlier in this chapter. In addition, it specifies the BookstoreContext class shown earlier in this chapter as the data store. This example uses all of the default options for the Identity service. As a result, Identity requires that a password must be at least 8 characters long with at least one lowercase letter, one uppercase letter, one number, and one special character.

If you want to relax or further restrict the default password options, you can use a lambda expression to set password options as shown in the third example. Here, the password options have been relaxed to only require 6 characters with at least one lowercase and one uppercase letter. Since this allows users to create weak passwords, you probably wouldn't want to relax password requirements on a production system. However, allowing weak passwords can make it easier to test an app.

In the Configure() method, you need to add statements that call the UseAuthentication() and UseAuthorization() methods. As you might expect, you need to code these statements in the correct sequence or they won't work correctly.

## The using directive for the Identity namespace

```
using Microsoft.AspNetCore.Identity;
```

## How to add the Identity service with default password options

```
public void ConfigureServices(IServiceCollection services) {
    ...
    services.AddIdentity<User, IdentityRole>()
        .AddEntityFrameworkStores<BookstoreContext>()
        .AddDefaultTokenProviders();
}
```

## Some properties of the PasswordOptions class

Property	Description
<code>RequiredLength</code>	Specifies the minimum length for the password. The default value is 8.
<code>RequireLowercase</code>	Specifies whether the password requires a lowercase letter.
<code>RequireUppercase</code>	Specifies whether the password requires an uppercase letter.
<code>RequireDigit</code>	Specifies whether the password requires a number.
<code>RequireNonAlphanumeric</code>	Specifies whether the password requires a special character.

## How to configure password options

```
public void ConfigureServices(IServiceCollection services) {
    ...
    services.AddIdentity<User, IdentityRole>(options => {
        options.Password.RequiredLength = 6;
        options.Password.RequireNonAlphanumeric = false;
        options.Password.RequireDigit = false;
    }).AddEntityFrameworkStores<BookstoreContext>()
        .AddDefaultTokenProviders();
}
```

## How to configure your app to use authentication and authorization

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

    app.UseAuthentication();
    app.UseAuthorization();

    app.UseSession();
    ...
}
```

## Description

- By default, a password must be at least 8 characters long with at least one lowercase letter, one uppercase letter, one number, and one special character.

---

Figure 16-7 How to configure the middleware for Identity

## How to add Log In/Out buttons and links to the layout

---

Once you've configured the middleware for Identity, you can begin to add authentication to your app. To start, if your app has a layout that provides a Bootstrap navbar, you can add Log In/Out buttons to it. In addition, you can add a Register link.

In figure 16-8, for example, the first screen shows a navbar that includes a Register link and a Log In button. This is the navbar that's displayed when the user is not logged in. Then, the second screen shows a navbar that includes a Log Out button and a username of joel. This is the navbar that's displayed when the user is logged in.

To provide for a navbar that can change like this, the layout uses a `if` statement to check whether the user is logged in. But first, it specifies a `using` directive for the Identity namespace, and it uses an `inject` directive to inject a `SignInManager` object named `signInManager` into the view. Then, the `if` statement checks whether the user is logged in. To do that, it passes the `User` property of the view to the `IsSignedIn()` method of the `signInManager` object.

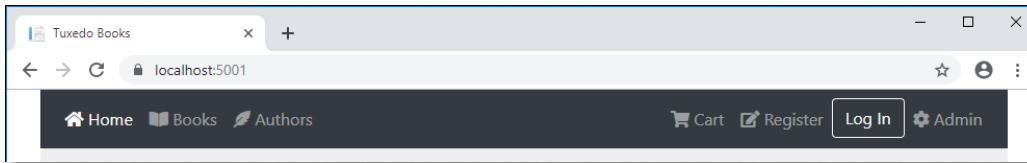
If the user is logged in, the view consists of a form that posts to the `LogOut()` action method of the `Account` controller. This form contains a Log Out button that submits the form. In addition, it contains a `<span>` element that displays the username for the logged in user. To do that, it uses the `User` property of the view to access the authenticated user. Then, it uses the `Identity` and `Name` properties to get the username of that user.

If the user is not logged in, the view displays a Register link that calls the `Register()` action method of the `Account` controller. The user can click this link to register a new account with the website. When the Register page is displayed, the register link should be the active link. To accomplish that, this code uses the `ViewContext` object to get the current action. Then, it uses the static `Nav.Active()` method to determine if the current action is "Register". If so, the `Bootstrap` active class is added to the `class` attribute, and the link is formatted as the active link.

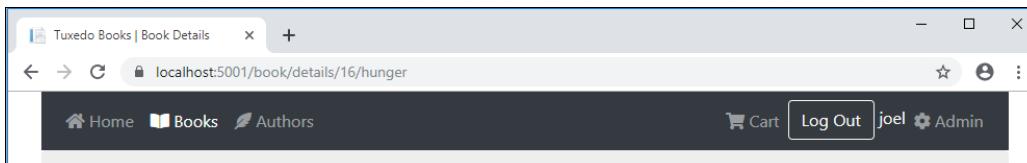
In addition to the Register link, this view displays a Log In link that calls the `LogIn()` action method of the `Account` controller. A registered user can click this link to log in to the website. Notice that the Log In link is formatted as a button. That makes it look like the Log Out button.

If you read chapter 15, you should realize that you could also modify the tag helper that determines the active link for the navbar so it works correctly with these links. In addition, you should realize that you could use a partial view for these Login buttons and links. For instance, you could store this code example in a partial view file named `_LoginLinks`. Then, you could insert these links at any point in your app that needed them.

## The Register link and the Log In button in the navbar



## The Log Out button in the navbar



## Some of the code for the navbar in the layout

```
<!-- Home, Books, Authors, and Cart links go here -->

@using Microsoft.AspNetCore.Identity
@inject SignInManager<User> signInManager
@if (signInManager.IsSignedIn(User))
{
    // signed-in user - Log Out button and username
    <li class="nav-item">
        <form method="post" asp-action="Logout" asp-controller="Account"
              asp-area="">
            <input type="submit" value="Log Out"
                  class="btn btn-outline-light" />
            <span class="text-light">@User.Identity.Name</span>
        </form>
    </li>
}
else
{
    // get current action
    var action = ViewContext.RouteData.Values["action"]?.ToString();

    // anonymous user - Register link and Log In button
    <li class="nav-item @Nav.Active("Register", action)">
        <a asp-action="Register" asp-controller="Account"
           asp-area="" class="nav-link">Register</a>
    </li>
    <li class="nav-item">
        <a asp-action="Login" asp-controller="Account"
           asp-area="" class="btn btn-outline-light">Log In</a>
    </li>
}

<!-- Admin link goes here -->
```

## Description

- The layout for the Bookstore app includes a Bootstrap navbar that contains links that let anonymous users register or log in. If a user is already logged in, the navbar displays the user's name and a button that lets that user log out.

Figure 16-8 How to add Log In/Out buttons and links to the layout

## How to start the Account controller

---

When a browser makes a request that doesn't pass authentication, ASP.NET Core MVC redirects to the /account/login URL by default. That's why it's common to use a controller named Account with a LogIn() action method to display the Login page. For example, figure 16-9 shows some starting code for a controller named Account.

When you code a controller that works with ASP.NET Core Identity, you can inject the `UserManager<T>`, `SignInManager<T>`, and `RoleManager<T>` objects into the controller. To do that, you can code private properties to store the objects. Then, you can code a constructor that accepts the `UserManager<T>`, `SignInManager<T>`, and `RoleManager<T>` objects and assigns them to the corresponding private properties. For instance, this figure defines private variables for `UserManager<User>` and `SignInManager<User>` objects, and it uses a constructor to initialize these objects. Once that's done, the rest of the methods in the controller can use these objects to work with Identity.

As you'll see throughout the rest of this chapter, the `Manager<T>` objects use asynchronous methods to work with the database. An *asynchronous method* can return control to the calling code before it finishes executing. This allows the calling code and the asynchronous method to execute simultaneously. Later, when the asynchronous method finishes executing, it returns its result. To make this possible, an asynchronous method typically runs in a different unit of processing, or *thread*, than the calling code.

By contrast, most of the code presented in this book uses synchronous methods. A *synchronous method* typically runs in the same thread as the calling code. As a result, it must finish executing before the calling code can continue. Synchronous methods are easier to understand and use than asynchronous methods. However, asynchronous methods can improve the responsiveness of an app, especially if the app needs to execute time-consuming tasks such as accessing a database.

Because the `Manager<T>` objects provide asynchronous methods for working with a database, the file that contains calls to these methods typically includes a using directive for the `System.Threading.Tasks` namespace. That's because this namespace contains the `Task<T>` object that's used to specify the return type for an asynchronous method. At this point, your app has the infrastructure it needs for you to begin coding the view models, action methods, and views for registering and logging in the users of your app.

## Some starting code for the Account controller

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Identity;
using Bookstore.Models;

namespace Bookstore.Controllers
{
    public class AccountController : Controller
    {
        private UserManager<User> userManager;
        private SignInManager<User> signInManager;

        public AccountController(UserManager<User> userMngr,
                               SignInManager<User> signInMngr)
        {
            userManager = userMngr;
            signInManager = signInMngr;
        }

        // The Register(), LogIn(), and LogOut()methods go here
    }
}
```

## The URL that MVC redirects to for an unauthenticated request

/account/login

### Description

- When you code a controller that works with ASP.NET Core Identity, you can inject the `UserManager<T>`, `SignInManager<T>`, and `RoleManager<T>` objects into the controller.
- The Manager`<T>` objects use asynchronous methods to work with the database. As a result, the controller that contains them typically includes a using directive for the `System.Threading.Tasks` namespace.
- When you call an *asynchronous method*, it returns control to the calling code before it finishes executing. That way, the calling code and the asynchronous method can execute simultaneously. This is possible because an asynchronous method typically runs in a different *thread* than the calling code.
- In contrast to an asynchronous method, a *synchronous method* typically runs in the same thread as the calling code, and it must finish executing before the calling code can continue.

---

Figure 16-9 How to start the Account controller

## How to register a user

---

To register a user, you can use a Register page like the one shown in figure 16-10. This page allows the user to register a new account with the website, and it logs that user in to the app. If the registration succeeds, the app redirects the user to the Home page. Otherwise, the app displays the Register page again with validation errors.

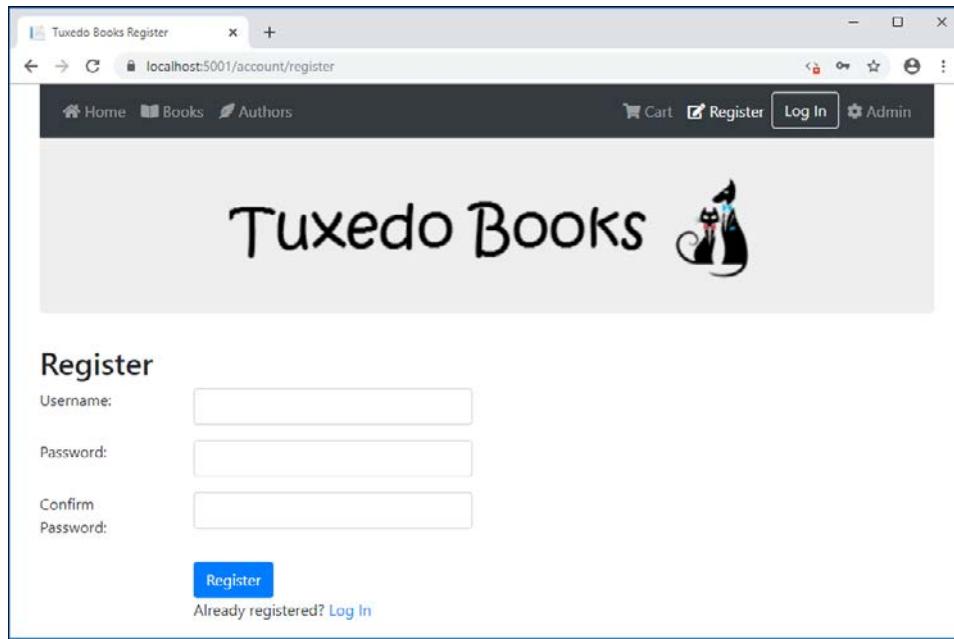
### The Register view model

---

Before you code the Account/Register view, you need to code the model for that view. That view model should define all of the fields that your app requires for the user to register. For instance, the code example in this figure shows the class that defines the Register view model for the Bookstore website. This view model includes the Username and Password properties to require a user to supply a username and a password. To make sure the user enters the password correctly, this view model also includes the ConfirmPassword property.

To provide for data validation, this view model uses some of the data attributes described in chapter 11. For example, it uses the Required attribute to require the user to enter a value for all three fields. In addition, to allow the password fields to use the password options specified in the Startup.cs file, this view model uses the DataType attribute to specify the Password data type. That way, you can easily change the validation that's required for the password by modifying the password options in the Startup.cs file.

## The Register page



## The Register view model

```
using System.ComponentModel.DataAnnotations;

namespace Bookstore.Models
{
    public class RegisterViewModel
    {
        [Required(ErrorMessage = "Please enter a username.")]
        [StringLength(255)]
        public string Username { get; set; }

        [Required(ErrorMessage = "Please enter a password.")]
        [DataType(DataType.Password)]
        [Compare("ConfirmPassword")]
        public string Password { get; set; }

        [Required(ErrorMessage = "Please confirm your password.")]
        [DataType(DataType.Password)]
        [Display(Name = "Confirm Password")]
        public string ConfirmPassword { get; set; }
    }
}
```

## Description

- The Register page creates a new user and signs in to the app as that user.
- If the registration succeeds, the app redirects the user to the Home page.
- If you want the password fields to use the password options specified in the Startup.cs file, the view model must use the DataType attribute to specify the Password type.

Figure 16-10 The Register view model

## The Account/Register view

---

The Register() action method for GET requests just displays the Account/Register view defined by the code in figure 16-11. This view begins by specifying the RegisterViewModel class presented in the previous figure as its model. Then, the view sets the title of the page in the ViewBag and displays a heading for the page.

Most of this view consists of a form that posts to the Register() action of the Account controller. Before the form, a `<div>` element displays a summary for any model-level validation messages. Within the form, the code uses `<div>` elements to display five rows.

The first three rows use the `asp-for` and `asp-validation-for` tag helpers to bind to the Username, Password, and ConfirmPassword properties of the view model. That way, any property-level validation messages are displayed next to their corresponding `<input>` element.

The fourth row displays a Register button that submits the form to the Register() action method for POST requests that's shown in the next figure. If the user enters valid data, this action method registers the user and logs the user in. Otherwise, it displays the Account/Register view again with appropriate validation messages.

The fifth row displays a link to the LogIn() action method for GET requests. This action method displays the Login page shown later in this chapter. That way, a user who is already registered can click this link to jump directly to the Login page.

## The Register() action method for GET requests

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

## The Account/Register view

```
@model RegisterViewModel  
{  
    ViewBag.Title = "Register";  
  
    <h1>Register</h1>  
  
    <div asp-validation-summary="ModelOnly" class="text-danger"></div>  
    <form method="post" asp-action="Register">  
        <div class="form-group row">  
            <div class="col-sm-2"><label>Username:</label></div>  
            <div class="col-sm-4">  
                <input asp-for="Username" class="form-control" />  
            </div>  
            <div class="col">  
                <span asp-validation-for="Username"  
                      class="text-danger"></span>  
            </div>  
        </div>  
        <div class="form-group row">  
            <div class="col-sm-2"><label>Password:</label></div>  
            <div class="col-sm-4">  
                <input type="password" asp-for="Password"  
                      class="form-control" />  
            </div>  
            <div class="col">  
                <span asp-validation-for="Password"  
                      class="text-danger"></span>  
            </div>  
        </div>  
        <div class="form-group row">  
            <div class="col-sm-2"><label>Confirm Password:</label></div>  
            <div class="col-sm-4">  
                <input type="password" asp-for="ConfirmPassword"  
                      class="form-control" />  
            </div>  
        </div>  
        <div class="row">  
            <div class="offset-2 col-sm-4">  
                <button type="submit" class="btn btn-primary">Register</button>  
            </div>  
        </div>  
        <div class="row">  
            <div class="offset-2 col-sm-4">  
                Already registered? <a asp-action="LogIn">Log In</a>  
            </div>  
        </div>  
    </div>  
</form>
```

---

Figure 16-11 The Account/Register view

## The Register() action method for POST requests

Figure 16-12 shows the Register() action method for POST requests. But first, it summarizes some of the methods and properties that are used by this action method. Here, all of the methods end with Async to indicate that they are asynchronous methods. This can improve the responsiveness of your web app, especially if the Identity operations take a long time to execute. However, the technique for calling an asynchronous method is a little different from the technique for calling a synchronous method.

The Register() action method shows how asynchronous methods work. To start, the declaration for this method includes the `async` keyword to indicate that it is also asynchronous. This is necessary since it makes calls to the `CreateAsync()` and `SignInAsync()` methods. In addition, the declaration for this method indicates that it returns an object of the `Task<T>` class that's available from the `System.Threading.Tasks` namespace mentioned earlier in this chapter.

Within the action method, the code starts by checking whether the model state is valid. If not, it passes the view model to the Account/Register view so it can display validation messages. However, if the model is valid, the code creates a `User` object and sets its `Username` property. Then, it attempts to create the user by passing the `User` object and the user's password to the `CreateAsync()` method of the private `userManager` property. For this asynchronous method call to work, it must be preceded by the `await` operator. This operator suspends execution of the Register() action method and returns control to the calling code. Then, when the `CreateAsync()` method finishes, the Register() action method resumes execution.

After calling the `CreateAsync()` method, the code uses the `IdentityResult` object that's returned to check whether the create operation succeeded. If so, it calls the `SignInAsync()` method of the private `signInManager` property to sign the user in using a session cookie. A session cookie makes sense here since you typically want to use a Remember Me check box to allow your users to choose whether they want to use a persistent cookie. After signing the user in, this code ends the action method by redirecting to the Home page for the app.

If the create user operation fails, the code loops through the `Errors` collection of the `IdentityResult` object and adds each error to the `ModelState` property. It adds these errors as model-level errors, so the Account/Register view displays them above the form.

## The LogOut() action method for POST requests

This figure also shows the LogOut() action method for POST requests that's called when the user clicks the Log Out button in the navbar. This action method is declared much like the LogIn() action method. However, LogOut() only contains two statements. The first statement calls the `SignOutAsync()` method of the private `signInManager` property to sign the user out. For this to work, the code includes the `await` keyword before the method call. Then, the second statement redirects the user to the Home page.

### Three methods of the UserManager class

Method	Description
<code>CreateAsync(user)</code>	Creates a user and returns an IdentityResult object.
<code>UpdateAsync(user)</code>	Updates a user and returns an IdentityResult object.
<code>DeleteAsync(user)</code>	Deletes a user and returns an IdentityResult object.

### Two methods of the SignInManager class

Method	Description
<code>SignInAsync(user, isPersistent)</code>	Logs in a user and returns an IdentityResult object. If the isPersistent argument is true, Identity uses a persistent cookie to keep the user logged in across multiple sessions. Otherwise, Identity uses a session cookie and the user is logged out when the current session ends.
<code>SignOutAsync()</code>	Logs out a user and returns an IdentityResult object.

### Two properties of the IdentityResult class

Property	Description
<code>Succeeded</code>	A Boolean value that indicates whether the operation was successful.
<code>Errors</code>	A collection of errors for the operation.

### The Register() action method for POST requests

```
[HttpPost]
public async Task<IActionResult> Register(RegisterViewModel model) {
    if (ModelState.IsValid) {
        var user = new User { UserName = model.Username };
        var result = await userManager.CreateAsync(user, model.Password);
        if (result.Succeeded) {
            await signInManager.SignInAsync(user, isPersistent: false);
            return RedirectToAction("Index", "Home");
        }
        else {
            foreach (var error in result.Errors) {
                ModelState.AddModelError("", error.Description);
            }
        }
    }
    return View(model);
}
```

### The LogOut() action method for POST requests

```
[HttpPost]
public async Task<IActionResult> LogOut() {
    await signInManager.SignOutAsync();
    return RedirectToAction("Index", "Home");
}
```

Figure 16-12 The Register() and LogOut() action methods for POST requests

## How to log in a user

---

To log in a user, you can use a Login page like the one shown in figure 16-13. This page allows a user who has already registered to log in to the website. If the log in succeeds, the app redirects the user to the page specified by the ReturnURL parameter in the query string. In this figure, for example, the ReturnURL parameter has a value of admin.

In addition, if the user checks the Remember Me box, the app uses a persistent cookie to keep the user logged in across multiple sessions. Otherwise, the app uses a session cookie that expires at the end of each session.

### The Login view model

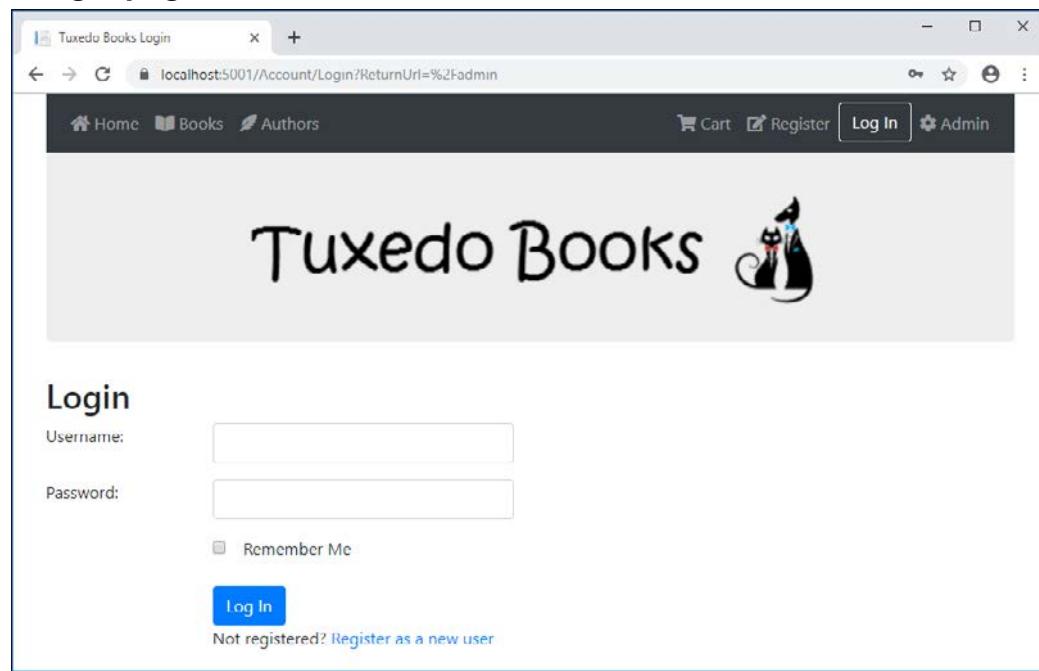
---

Before you code the Account/Login view, you need to code the model for that view. That view model should define all of the fields needed by this page. For instance, this figure shows the class that defines the Login view model for the Bookstore website. Obviously, this view model needs to include the Username and Password properties that the user must enter to register.

In addition, the view model includes two more properties. First, the ReturnURL property stores the URL that the app should return to if the log in operation succeeds. Second, the RememberMe property stores a Boolean value that indicates whether to use a persistent cookie or a session cookie for authentication.

To provide for data validation, this view model uses some of the data attributes described in chapter 11. Note that it doesn't use the DataType attribute for the password field as described earlier in this chapter, though. That's because the app doesn't need to display validation messages for a password that doesn't meet the minimum requirements. Instead, the app only needs to display a message that indicates whether the username/password combination is valid or not.

## The Login page



## The Login view model

```
using System.ComponentModel.DataAnnotations;

namespace Bookstore.Models
{
    public class LoginViewModel
    {
        [Required(ErrorMessage = "Please enter a username.")]
        [StringLength(255)]
        public string Username { get; set; }

        [Required(ErrorMessage = "Please enter a password.")]
        [StringLength(255)]
        public string Password { get; set; }

        public string ReturnUrl { get; set; }

        public bool RememberMe { get; set; }
    }
}
```

### Description

- If the user logs in successfully, the app redirects the user to the page specified by the ReturnURL query string.
- If the user checks the Remember Me box, the app uses a persistent cookie to keep the user logged in across multiple sessions. Otherwise, the app uses a session cookie that expires at the end of each session.

Figure 16-13 The Login view model

## The Account/Login view

---

The LogIn() action method for GET requests displays the Account/Login view defined by the code in figure 16-14. But first, it creates a LoginViewModel object and initializes its ReturnURL property to the returnUrl parameter of the action method. This works because the name of this parameter matches the name of the query string parameter.

This Account/Login view itself begins by specifying the LoginViewModel class as its model. Then, the view sets the title of the page in the ViewBag, displays a heading for the page, and displays a summary for any model-level validation messages.

Most of the Account/Login view consists of a form that posts to the LogIn() action method of the Account controller. Note that the `<form>` tag uses the `asp-route-returnUrl` tag helper to specify a value for the returnUrl query string parameter. Within the form, the code uses `<div>` elements to display five rows.

The first two rows use the `asp-for` and `asp-validation-for` tag helpers to bind to the `Username` and `Password` properties of the view model. That way, any property-level validation messages are displayed next to the corresponding `<input>` element.

The third row displays the Remember Me check box and uses the `asp-for` tag helper to bind it to the `RememberMe` property of the view model. By default, this box is unchecked. As a result, the app uses a session cookie for authentication. However, if the user checks this box, the app uses a persistent cookie for authentication.

The fourth row displays a Log In button that submits the form to the LogIn() action method for POST requests that's shown in the next figure. If the user enters valid data, this action method logs the user in. Otherwise, it displays the Account/Login view again with appropriate validation messages.

The fifth row displays a link to the Register() action method for GET requests. This action method displays the Register page shown earlier in this chapter. That way, a user who isn't registered can click this link to jump directly to the Register page.

## The LogIn() action method for GET requests

```
[HttpGet]
public IActionResult LogIn(string returnUrl = "")
{
    var model = new LoginViewModel { ReturnUrl = returnUrl };
    return View(model);
}
```

## The Login view

```
@model LoginViewModel
@{
    ViewBag.Title = "Login";
}



# Login



</div>
<form method="post" asp-action="LogIn"
      asp-route-returnUrl="@Model.ReturnUrl">
    <div class="form-group row">
        <div class="col-sm-2"><label>Username:</label></div>
        <div class="col-sm-4">
            <input asp-for="Username" class="form-control" />
        </div>
        <div class="col">
            <span asp-validation-for="Username" class="text-danger"></span>
        </div>
    </div>
    <div class="form-group row">
        <div class="col-sm-2"><label>Password:</label></div>
        <div class="col-sm-4">
            <input type="password" asp-for="Password"
                  class="form-control" />
        </div>
        <div class="col">
            <span asp-validation-for="Password" class="text-danger"></span>
        </div>
    </div>
    <div class="form-group row">
        <div class="offset-sm-2 col-sm-4">
            <input type="checkbox" title="Remember Me" asp-for="RememberMe"
                  class="form-check" />
            <label>Remember Me</label>
        </div>
    </div>
    <div class="row">
        <div class="offset-2 col-sm-4">
            <button type="submit" class="btn btn-primary">Log In</button>
        </div>
    </div>
    <div class="row">
        <div class="offset-2 col-sm-4">
            Not registered?
            <a asp-action="Register">Register as a new user</a>
        </div>
    </div>
</form>


```

Figure 16-14 The Account/Login view

## The LogIn() action method for POST requests

Figure 16-15 shows the LogIn() action method for POST requests. But first, it summarizes the PasswordSignInAsync() method that's used by this action method. This method is an asynchronous method that works much like the other asynchronous methods described earlier in this chapter.

The LogIn() action method shows how to use this method. To start, the declaration for this action method is coded with the `async` keyword and returns a `Task<IActionResult>` object as described earlier in this chapter. Then, within the action method, the code starts by checking whether the model state is valid. If not, it adds a model-level validation error that says “Invalid username/password”, and it passes the view model to the Account/Login view so it can display validation messages. In most cases, that's adequate validation for a Login page.

On the other hand, if the model is valid, the code calls the `PasswordSignInAsync()` method of the private `signInManager` property and passes it the username, the password, and the value of the `RememberMe` property. In addition, it passes a Boolean value of `false` as the fourth argument. That way, the user is not locked out if the sign in operation fails. In most cases, that's adequate. However, if you want to make your page more secure, you can set this argument to `true`.

If the sign in operation succeeds, the code checks that the string for the `ReturnURL` property of the view model is not null or empty and that it contains a local URL. If both of these conditions are true, this code redirects to the URL specified by the `ReturnURL` property. Otherwise, the action method redirects to the Home page. Here, checking that the `ReturnURL` contains a local URL helps to protect against a hacker redirecting the browser to a malicious website.

## Another method of the SignInManager class

Method	Description
<code>PasswordSignInAsync(username, password, isPersistent, lockoutOnFailure)</code>	Logs in a user and returns an IdentityResult object. When lockoutOnFailure is set to true, Identity locks the user out if the sign in fails.

## The Login() action method for POST requests

```
[HttpPost]
public async Task<IActionResult> LogIn(LoginViewModel model)
{
    if (ModelState.IsValid)
    {
        var result = await signInManager.PasswordSignInAsync(
            model.Username, model.Password, isPersistent: model.RememberMe,
            lockoutOnFailure: false);

        if (result.Succeeded)
        {
            if (!string.IsNullOrEmpty(model.ReturnUrl) &&
                Url.IsLocalUrl(model.ReturnUrl))
            {
                return Redirect(model.ReturnUrl);
            }
            else
            {
                return RedirectToAction("Index", "Home");
            }
        }
    }
    ModelState.AddModelError("", "Invalid username/password.");
    return View(model);
}
```

---

Figure 16-15 The LogIn() action method for POST requests

## How to work with roles

So far, you've learned how to allow a user to log in. As a result, you can restrict access to users who are logged in. For example, you might want to restrict access to the Cart page so any user who is logged in can view it.

However, you may also need to restrict access to users who are members of a specified role. For example, if your website has an Admin area, you might want to restrict access to users who are members of the Admin or Manager role. The figures that follow show how to do that.

### Properties and methods for working with roles

Figure 16-16 shows some of the properties and methods you can use to work with users and roles. To work with roles, you can use the properties and methods of the RoleManager class. For example, you can create, update, or delete a role. You can find a role by id or name. And you can get a collection of all roles.

Once you've created a role or two, you can use the properties and methods of the UserManager class to work with users and roles. For example, you can add a user to a role or remove a user from a role. You can find a user by id or name. You can check whether a user is in a specified role. You can get a collection of role names that the specified user belongs to. And you can get a collection of all users.

The first code example shows how to loop through all users and their roles. To do that, this code uses the Users and Roles collections that are available from the UserManager and RoleManager classes. Within the inner loop, this code uses the IsInRoleAsync() method to check whether the current user is a member of the current role. If so, you can perform some processing.

The second and third examples show how to use a RoleManager object to create and delete a role. Here, the second example uses the CreateAsync() method to create a role named Admin. Then, the third example uses the FindByIdAsync() method to find a role by its ID. Then, it uses the DeleteAsync() method to delete the role that matches that ID.

The fourth and fifth examples show how to use a UserManager object to add a user to a role and remove a user from a role. Here, the fourth example uses the FindByIdAsync() method to get the user with the specified ID. Then, it uses the AddToRoleAsync() method to add that user to the role named Admin. For this to succeed, the role named Admin must already exist.

The fifth example uses the FindByIdAsync() method to get a user with the specified ID. Then, it uses the RemoveFromRoleAsync() method to remove that user from the Admin role.

When you use the methods shown in this figure, it's common to check whether the operation succeeded. Then, you can perform some processing if the operation succeeded or you can handle the error if the operation didn't succeed. In this figure, the last four examples show the if statement that checks whether the operation succeeded, but it doesn't show any of the processing for these if statements. Instead, these examples just use an ellipsis (...) to show that you could perform some processing here.

## Some of the properties and methods of the RoleManager class

Property/Method	Description
<code>Roles</code>	Returns an IQueryable object of roles.
<code>FindByIdAsync(id)</code>	Returns an IdentityRole object for the specified role ID.
<code>FindByNameAsync(name)</code>	Returns an IdentityRole object for the specified role name.
<code>CreateAsync(role)</code>	Creates a role and returns an IdentityResult object.
<code>UpdateAsync(role)</code>	Updates a role and returns an IdentityResult object.
<code>DeleteAsync(role)</code>	Deletes a role and returns an IdentityResult object.

## More properties and methods of the UserManager class

Property/Method	Description
<code>Users</code>	Returns an IQueryable object of users.
<code>FindByIdAsync(id)</code>	Returns an IdentityUser object for the specified user ID.
<code>FindByNameAsync(name)</code>	Returns an IdentityUser object for the specified username.
<code>IsInRoleAsync(user, roleName)</code>	Returns a Boolean value that indicates whether the user is in the specified role.
<code>AddToRoleAsync(user, roleName)</code>	Adds the specified user to the specified role.
<code>RemoveFromRoleAsync(user, roleName)</code>	Removes the specified user from the specified role.
<code>GetRolesAsync(user)</code>	Returns a collection of role names for the specified user.

### Code that loops through all users and their roles

```
foreach (User user in userManager.Users) { // all users
    foreach (IdentityRole role in roleManager.Roles) { // all roles
        if (await userManager.IsInRoleAsync(user, role.Name)) {
            // perform some processing if user is in role
        }
    }
}
```

### Code that creates a role named Admin

```
var result = await roleManager.CreateAsync(new IdentityRole("Admin"));
if (result.Succeeded) { ... }
```

### Code that deletes a role

```
IdentityRole role = await roleManager.FindByIdAsync(id);
var result = await roleManager.DeleteAsync(role);
if (result.Succeeded) { ... }
```

### Code that adds a user to the role named Admin

```
User user = await userManager.FindByIdAsync(id);
var result = await userManager.AddToRoleAsync(user, "Admin");
if (result.Succeeded) { ... }
```

### Code that removes a user from the role named Admin

```
User user = await userManager.FindByIdAsync(id);
var result = await userManager.RemoveFromRoleAsync(user, "Admin");
if (result.Succeeded) { ... }
```

Figure 16-16 Properties and methods for working with roles

## The User entity and view model

---

Now that you know how to use the properties and methods for working with roles, you're ready to learn how to create a Manage Users page like the one shown in figure 16-17. This page lets you create and delete the Admin role, create and delete users, add users to the Admin role, and remove users from the Admin role. In this figure, the User Manager page displays three users. Two of the users don't belong to any role, and the third user belongs to the Admin role.

Before creating this Manage Users page, you need to update the User entity class so it includes a RoleNames property that can store the names of all roles that the user belongs to. This provides an easy way for the User/Index view to display all roles for a user. When you code the RoleNames property, you need to add the NotMapped attribute so EF doesn't create a RoleNames column in the AspNetUsers table.

In addition, you need to create a User view model that stores a collection of User and Role objects. This provides an easy way for the view to access the users and roles that it needs to display.

For now, the Manage Users page only works with a single role named Admin. However, both the User entity and User view model classes are coded in a way that allows them to work with multiple roles if necessary. As a result, if you ever need to update this view to work with multiple roles, these classes are ready for it.

## The Manage Users page

The screenshot shows the 'Manage Users' page of the Tuxedo Books application. At the top, there's a navigation bar with links for Home, Books, Authors, Cart, Register, Log In, and Admin. The main title 'Tuxedo Books' is displayed with a cat icon. Below the title, there are four tabs: Manage Books, Manage Authors, Manage Genres, and Manage Users, with 'Manage Users' being the active tab. The 'Manage Users' section contains a table with columns for Username and Roles. The rows show three users: joelmurach (Admin), gracehopper (Admin), and admin (Admin). Each row has three buttons: 'Delete User', 'Add To Admin', and 'Remove From Admin'. Below this is a 'Manage Roles' section with a table showing the Admin role, which has a 'Delete Role' button.

## The updated User entity

```
using System.ComponentModel.DataAnnotations.Schema; // for NotMapped attr
...
public class User : IdentityUser
{
    [NotMapped]
    public IList<string> RoleNames { get; set; }
}
```

## The User view model

```
public class UserViewModel
{
    public IEnumerable<User> Users { get; set; }
    public IEnumerable<IdentityRole> Roles { get; set; }
}
```

## Description

- The Manage Users page lets you add and delete users, add and remove users from the Admin role, and delete the Admin role.
- If the Admin role doesn't exist, the Manage Users page displays a button that you can click to create the Admin role.

Figure 16-17 The User entity and view model

## The User controller and its Index() action method

Figure 16-18 shows the User controller and its Index() action method. Here, the User controller is decorated with an Authorize attribute that only allows users of the Admin role to access this page. That's typically what you want for a production app. However, when you're developing this page, you can comment out this attribute until you successfully create a user that's a member of the Admin role.

The User controller begins by declaring private properties named userManager and roleManager. Then, the constructor for the User controller initializes these properties to instances of the UserManager and RoleManager objects. That way, the entire controller can use the private properties to easily access these objects.

Since the Index() method calls asynchronous methods, it includes the `async` keyword and returns a `Task` object. Within this method, the first statement creates a list of `User` objects named `users`. Then, the code loops through all users.

Within this loop, the first statement gets a list of all role names that the user is a member of and assigns that list to the `RoleNames` property of the `User` object. This works because the call to the `GetRolesAsync()` method returns an `IList<string>` object, and that's also the type of the `RoleNames` property of the `User` object. Then, the second statement adds the updated `User` object to the list of `User` objects named `users`.

After the loop, the code creates the `User` view model and initializes its `Users` and `Roles` properties. To do that, it assigns the list of `User` objects named `users` to the `Users` property. Then, it uses the `RoleManager` object to assign its `Roles` property to the `Roles` property of the view model.

Finally, the Index() method passes the view model to the view. This renders the `User/Index` view that's presented in the next figure.

## The User controller and its Index() action method

```
using System.Threading.Tasks;
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Authorization;
using Microsoft.AspNetCore.Identity;
using Bookstore.Models;
...
[Authorize(Role = "Admin")]
[Area("Admin")]
public class UserController : Controller
{
    private UserManager<User> userManager;
    private RoleManager<IdentityRole> roleManager;

    public UserController(UserManager<User> userMngr,
        RoleManager<IdentityRole> roleMngr)
    {
        userManager = userMngr;
        roleManager = roleMngr;
    }

    public async Task<IActionResult> Index()
    {
        List<User> users = new List<User>();
        foreach (User user in userManager.Users)
        {
            user.RoleNames = await userManager.GetRolesAsync(user);
            users.Add(user);
        }
        UserViewModel model = new UserViewModel
        {
            Users = users,
            Roles = roleManager.Roles
        };
        return View(model);
    }

    // the other action methods
}
```

---

Figure 16-18 The User controller and its Index() action method

## The User/Index view

---

The User/Index view shown in figure 16-19 begins by identifying the `UserViewModel` class as its model. Then, the view sets the title of the page in the `ViewBag` and displays a heading for the page.

After that, it includes an `Add` link that calls the `Add()` action method of the `User` controller. This action method displays a `User/Add` view that you can use to add a user. However, this chapter doesn't present the code for this action method and its view because it works much like the `Account/Register` view presented earlier in this chapter. In fact, the `Add()` action method and the `User/Add` view use the `RegisterViewModel` presented earlier in this chapter.

After the `Add` link, the `User/Index` view displays the header row of a table. Then, a Razor `if` statement checks whether any `User` objects exist in the view model. If not, it displays a row that indicates that there are no user accounts. Otherwise, it loops through all `User` objects.

Within the loop, the code creates one row for each user where each row has five columns. The first column displays the username. The second column displays any roles that the user is in. To do that, the code for this column uses a Razor `foreach` statement to loop through all roles for the user.

The next three columns display buttons that you can use to work with each user. The third column displays a button that you can use to delete the user. This button calls the `Delete()` action method and passes the user's ID as a route parameter. The fourth and fifth columns work similarly, but they display buttons that you can use to add the user to the Admin role or remove the user from the Admin role. These buttons call the `AddToAdmin()` and `RemoveFromAdmin()` action methods of the `User` controller.

Note that to make this app more efficient, you could display the `Add To Admin` button only if the user isn't already a member of the Admin role. To do that, you could code the `<form>` element for the column within a Razor `if` statement like this:

```
@if (!user.RoleNames.Contains("Admin"))
```

Similarly, you could display the `Delete From Admin` button only if the user is a member of the Admin role.

## The User/Index view

```
@model UserViewModel
{
    ViewData["Title"] = " | Manage Users";
}

<h1 class="mb-2">Manage Users</h1>

<h5 class="mt-2"><a href="#" asp-action="Add">Add a User</a></h5>

<table class="table table-bordered table-striped table-sm">
    <thead>
        <tr><th>Username</th><th>Roles</th><th></th><th></th><th></th><th></th></tr>
    </thead>
    <tbody>
        @if (Model.Users.Count() == 0)
        {
            <tr><td colspan="5">There are no user accounts.</td></tr>
        }
        else
        {
            @foreach (User user in Model.Users)
            {
                <tr>
                    <td>@user.UserName</td>
                    <td>
                        @foreach (string roleName in user.RoleNames)
                        {
                            <div>@roleName</div>
                        }
                    </td>
                    <td>
                        <form method="post" asp-action="Delete"
                              asp-route-id="@user.Id">
                            <button type="submit" class="btn btn-primary">
                                Delete User</button>
                        </form>
                    </td>
                    <td>
                        <form method="post" asp-action="AddToAdmin"
                              asp-route-id="@user.Id">
                            <button type="submit" class="btn btn-primary">
                                Add To Admin</button>
                        </form>
                    </td>
                    <td>
                        <form method="post" asp-action="RemoveFromAdmin"
                              asp-route-id="@user.Id">
                            <button type="submit" class="btn btn-primary">
                                Remove From Admin</button>
                        </form>
                    </td>
                </tr>
            }
        }
    </tbody>
</table>
```

---

Figure 16-19 The User/Index view (part 1)

After the table of users, the User/Index view uses a Razor if statement to check whether any Role objects exist in the view model. If not, it displays a form that contains a button that you can use to create the Admin role. Clicking this button calls the CreateAdminRole() action method of the User controller.

On the other hand, if one or more roles exist, this code displays a table that has one row for each role. Here, each row has two columns. The first column displays the name of the role. Then, the second column displays a button that you can use to delete the role. This button calls the DeleteRole() action method of the User controller and passes the role's ID as a route parameter.

## The User/Index view (continued)

```
<h1 class="mb-2">Manage Roles</h1>

@if (Model.Roles.Count() == 0)
{
    <form method="post" asp-action="CreateAdminRole">
        <button type="submit" class="btn btn-primary">
            Create Admin Role</button>
    </form>
}
else
{
    <table class="table table-bordered table-striped table-sm">
        <thead>
            <tr><th>Role</th><th></th></tr>
        </thead>
        <tbody>
            @foreach (var role in Model.Roles)
            {
                <tr>
                    <td>@role.Name</td>
                    <td>
                        <form method="post" asp-action="DeleteRole"
                            asp-route-id="@role.Id">
                            <button type="submit" class="btn btn-primary">
                                Delete Role</button>
                        </form>
                    </td>
                </tr>
            }
        </tbody>
    </table>
}
```

---

Figure 16-19 The User/Index view (part 2)

## Other action methods of the User controller

---

Figure 16-20 shows the action methods of the User controller that are called by the buttons of the User/Index view. For example, if you click the Delete button for a user, the app calls the Delete() action method to delete the specified user. Similarly, if you click the Add To Admin button, the app calls the AddToAdmin() action method.

The Delete() action method accepts the user's ID as a parameter. Within the method, the first statement gets a User object for the user with that ID. Then, the code checks whether this User object is null. If not, the code attempts to delete the user. If the delete operation fails, the code loops through all errors and creates an error message that it adds to TempData with a key of "message". For the Bookstore website, this causes the layout to display the error message across the top of the User Manager page. If the delete operation succeeds, the User Manager page displays the remaining users and roles for the app.

The AddToAdmin() action method also accepts the user's ID as a parameter. Within the method, the first statement gets an IdentityRole object for the role named Admin. Then, the code checks whether that IdentityRole object is null. If so, it adds an error message to TempData with a key of "message" and redirects to the User/Index view. This displays an error message across the top of the User Manage page that indicates that the Admin role doesn't exist and needs to be created. However, if the Admin role exists, this method finds the User object with the ID parameter. Then, it adds this user to the Admin role.

The RemoveFromAdmin() action method works much like the AddToAdmin() action method. However, removing a user from a role that doesn't exist doesn't cause an error. As a result, there's no need to check whether the Admin role exists in this method. So, this method just finds the User object that matches the ID parameter, removes that user from the Admin role, and redirects to the User/Index page.

The DeleteRole() action method accepts the role's ID as a parameter. Within this method, the first statement gets the IdentityRole object with the ID parameter. Then, this method deletes this role and redirects to the User/Index page.

The CreateAdminRole() action method doesn't accept any parameters. It just creates the Admin role and redirects to the User/Index page.

As you review these methods, you should realize that the ones with "Admin" in their names are hard-coded to work with a role named Admin. For the Bookstore website, that's adequate since it only supports the Admin role. However, if you need to support multiple roles, you could rename and refactor these methods to work with multiple methods. For example, you could rename CreateAdminRole() to CreateRole() and refactor it to accept an argument that specifies the name of the role to create.

## Other action methods of the User controller

```
[HttpPost]
public async Task<IActionResult> Delete(string id)
{
    User user = await userManager.FindByIdAsync(id);
    if (user != null) {
        IdentityResult result = await userManager.DeleteAsync(user);
        if (!result.Succeeded) { // if failed
            string errorMessage = "";
            foreach (IdentityError error in result.Errors) {
                errorMessage += error.Description + " | ";
            }
            TempData["message"] = errorMessage;
        }
    }
    return RedirectToAction("Index");
}

// the Add() methods work like the Register() methods from 16-11 and 16-12

[HttpPost]
public async Task<IActionResult> AddToAdmin(string id)
{
    IdentityRole adminRole = await roleManager.FindByNameAsync("Admin");
    if (adminRole == null) {
        TempData["message"] = "Admin role does not exist. "
            + "Click 'Create Admin Role' button to create it.";
    }
    else {
        User user = await userManager.FindByIdAsync(id);
        await userManager.AddToRoleAsync(user, adminRole.Name);
    }
    return RedirectToAction("Index");
}

[HttpPost]
public async Task<IActionResult> RemoveFromAdmin(string id)
{
    User user = await userManager.FindByIdAsync(id);
    await userManager.RemoveFromRoleAsync(user, "Admin");
    return RedirectToAction("Index");
}

[HttpPost]
public async Task<IActionResult> DeleteRole(string id)
{
    IdentityRole role = await roleManager.FindByIdAsync(id);
    await roleManager.DeleteAsync(role);
    return RedirectToAction("Index");
}

[HttpPost]
public async Task<IActionResult> CreateAdminRole()
{
    await roleManager.CreateAsync(new IdentityRole("Admin"));
    return RedirectToAction("Index");
}
```

---

Figure 16-20 Other action methods of the User controller

## The code that restricts access

---

Figure 16-21 shows the code that restricts access to some pages of the Bookstore website. To start, the first example shows the using directive for the namespace that contains the Authorization attributes that you can use to restrict access. Then, it shows how to apply these attributes to some of the controllers of the Bookstore website.

The second example shows the Authorize attribute that's applied to the Cart controller and all of its action methods. This attribute restricts access to all of the Cart pages to users that are logged in. As a result, if you are not logged in and you attempt to access any Cart page, the app redirects you to the Login page. At this point, you must log in to access the Cart page. Of course, if you haven't registered yet, you must register before you can log in.

The third example shows the Authorize attribute that's applied to the Book controller in the Admin area. However, this attribute is also applied to all other controllers in the Admin area such as the Author and Genre controllers. This Authorize attribute restricts access to users who are members of the Admin role. As a result, if you are logged in as a user who is not a member of the Admin role and you attempt to access one of these Admin pages, the app redirects to this URL by default:

**/Account/AccessDenied**

For this to work, you can add an action method named AccessDenied() to the Account controller as shown in the fourth example. This method can just return an Account/AccessDenied view like the one shown in the fifth example. Of course, if you log in as a user that's a member of the Admin role, you can access the Admin pages.

## The using directive for the Authorization attributes

```
using Microsoft.AspNetCore.Authorization;
```

## The Cart controller requires users to be logged in

```
[Authorize]  
public class CartController : Controller {  
    ...  
}
```

## All controllers in the Admin area require users to be in the Admin role

```
[Authorize(Roles = "Admin")]  
[Area("Admin")]  
public class BookController : Controller {  
    ...  
}
```

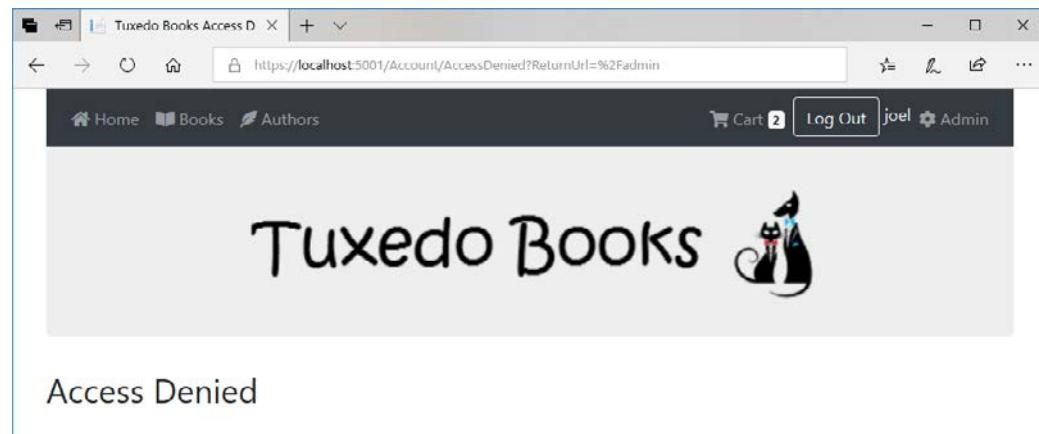
## The AccessDenied() action method of the Account controller

```
public ViewResult AccessDenied()  
{  
    return View();  
}
```

## The code for the Account/AccessDenied view

```
@{  
    ViewBag.Title = "Access Denied";  
}  
<h2>Access Denied</h2>
```

## The Account/AccessDenied view



## Description

- If you are not logged in and you attempt to access the Cart or Admin pages, the app redirects you to the Login page.
- If you are logged in as any user, you can access the Cart pages.
- If you are logged in as a user that's in the Admin role, you can access the Admin pages. Otherwise, the app redirects you to the AccessDenied view.

Figure 16-21 The code that restricts access

## How to seed roles and users

---

Figure 16-22 presents some skills that you can use to seed your database with some initial roles and users. In particular, it shows how to create a user named admin that you can use to access the pages in the Admin area of the Bookstore website.

To start, you can create a static asynchronous method in the DB context class like the CreateAdminUser() method shown in the first example. This method accepts a parameter named serviceProvider. Within the method, the first two statements use this parameter to get the UserManager<User> and RoleManager<IdentityRole> objects that the method needs to work with users and roles. Then, the code creates three string variables that store a username of admin, a password of Sesame, and a role name of Admin.

Next, this code checks whether the specified role already exists. If it doesn't, this code creates the role. Then, the code checks whether the specified username already exists. If it doesn't, this code creates a User object and stores the username in it. Then, it creates the user. If this operation succeeds, the code adds the user to the Admin role.

To execute the CreateAdminUser() method, you can call it from the Configure() method that's in the Startup.cs file. When you do that, you need to pass the ApplicationServices property that's available from its app argument as the first argument of the CreateAdminUser() method. In addition, since the CreateAdminUser() method is an asynchronous method, you must chain the Wait() method to the end of the method call.

For this approach to work, you must modify the default configuration options in the Program.cs file. To do that, you can add the code that's highlighted in the third example. This code changes the options for the application service provider so dependency injection can be used to pass it from one scope (the Startup class) to another scope (the BookstoreContext class). If you forget to do this, you won't be able to pass the application services from the Configure() method to the CreateAdminUser() method, and the web server won't start correctly.

As you review this code, note that it only creates one role named Admin and one user with a username of admin and a password of Sesame. However, if necessary, it would be easy to modify this code to create multiple roles and users. For example, you could store all role names in a list and loop through the list to create each role. Similarly, you could store all username/password combinations for a specified role in a dictionary and loop through each key/value pair to create all of the users for that role.

### A method that's added to the DB context class

```

using Microsoft.Extensions.DependencyInjection;
using Microsoft.AspNetCore.Identity;
using Microsoft.AspNetCore.Identity.EntityFrameworkCore;
...
public static async Task CreateAdminUser(IServiceProvider serviceProvider)
{
    UserManager<User> userManager =
        serviceProvider.GetRequiredService<UserManager<User>>();
    RoleManager<IdentityRole> roleManager =
        serviceProvider.GetRequiredService<RoleManager<IdentityRole>>();

    string username = "admin";
    string password = "Sesame";
    string roleName = "Admin";

    // if role doesn't exist, create it
    if (await roleManager.FindByNameAsync(roleName) == null) {
        await roleManager.CreateAsync(new IdentityRole(roleName));
    }

    // if username doesn't exist, create it and add it to role
    if (await userManager.FindByNameAsync(username) == null) {
        User user = new User { UserName = username };
        var result = await userManager.CreateAsync(user, password);
        if (result.Succeeded) {
            await userManager.AddToRoleAsync(user, roleName);
        }
    }
}

```

### A statement in the Startup.cs file that calls the method

```

public void Configure(IApplicationBuilder app, IWebHostEnvironment env)
{
    // all other configuration statements

    BookstoreContext.CreateAdminUser(app.ApplicationServices).Wait();
}

```

### An option in the Program.cs file that allows the method to execute

```

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

    });

```

### Description

- If you need to seed your database with some initial roles and users, you can create a static asynchronous method in the DB context class. Then, you can call this method from the Startup.cs file. For this approach to work, you must modify the default configuration options in the Program.cs file as shown in this figure.

Figure 16-22 How to seed roles and users

## More skills for working with Identity

At this point, you have learned how to restrict access to the pages of the Bookstore website. In addition, you have learned how to create a User Manager page that you can use to view and modify the users of an app and to control whether they are members of the Admin role. However, there are many more skills for working with Identity that you should be aware of. The next two figures describe a couple of them.

### How to change a user's password

If you need to change a user's password, you can use a view like the Change Password view shown in figure 16-23. Here, the Change Password view displays the username but doesn't allow you to change it. In addition, it displays text boxes that let you enter the old and new passwords for the user.

The view model for the User/Change Password view includes properties for storing the username, the old password, and the new password. In addition, it uses data attributes to provide data validation for the old and new passwords. Here, no data validation is required for the username since the app always sets it correctly. Similarly, the data validation for the old password is minimal because the most important part of its validation is whether it matches the password that's stored in the database. However, the data validation for the new password is the same as it is for the Account/Registration view.

When the user enters the old and new passwords and clicks the Change Password button, the view posts the view model to the ChangePassword() action method. This action method begins by checking whether the model state is valid. If not, it displays the view again with the appropriate validation messages. However, if the model state is valid, it gets the User object with the username that's available from the view model. Then, it uses the ChangePasswordAsync() method to change the password for the User object from the old password to the new password. If this operation succeeds, the code redirects to the User/Index page. Otherwise, it adds model-level errors to the ModelState property and displays the User/Change Password view with the model-level validation errors.

Note that this figure doesn't show the ChangePassword() method for GET requests. That's because this method just displays the User/ChangePassword view. Similarly, this figure doesn't show the code for the User/ChangePassword view. That's because this code is similar to the code for the Account/Register view that was presented earlier in this chapter.

## The User/Change Password view

The screenshot shows a web page titled "Change Password". At the top, there are navigation links: "Manage Books", "Manage Authors", "Manage Genres", and "Manage Users". Below the title, there are three input fields: "Username" (containing "joelmurach"), "Old Password", and "New Password". At the bottom of the form is a blue "Change Password" button.

## The User/ChangePassword view model

```
using System.ComponentModel.DataAnnotations;

namespace Bookstore.Models
{
    public class ChangePasswordViewModel
    {
        public string Username { get; set; }

        [Required(ErrorMessage = "Please enter password.")]
        public string OldPassword { get; set; }

        [Required(ErrorMessage = "Please enter new password.")]
        [DataType(DataType.Password)]
        public string NewPassword { get; set; }
    }
}
```

## The ChangePassword() action method for POST requests

```
[HttpPost]
public async Task<IActionResult> ChangePassword(
    ChangePasswordViewModel model)
{
    if (ModelState.IsValid)
    {
        User user = await userManager.FindByNameAsync(model.Username);
        var result = await userManager.ChangePasswordAsync(user,
            model.OldPassword, model.NewPassword);
        if (result.Succeeded)
        {
            return RedirectToAction("Index");
        }
        else
        {
            foreach (IdentityError error in result.Errors)
            {
                ModelState.AddModelError("", error.Description);
            }
        }
    }
    return View(model);
}
```

Figure 16-23 How to change a user's password

## How to add more user registration fields

The Account/Register view presented earlier in this chapter only requires two fields: username and password. But what if you wanted to require more fields for a user to register? For example, what if you also wanted to require the user to enter a name, an email address, or a birth date? In that case, you can use the technique described in figure 16-24 to require more fields.

If you want to add a registration field that's available from the IdentityUser class, you can just add it to the Register view model. That's because the AspNetUsers table already contains the column needed to store the field.

For instance, the IdentityUser class presented earlier in this chapter includes an Email property that's inherited by the User class shown in this figure. As a result, you don't need to add an Email property to the User class. Instead, you only need to add the Email property to the Register view model as shown in this figure. When you do, you can use the DataType attribute to validate the email address field. This works much like using the DataType attribute to validate the password field.

However, if you want to add a registration field that isn't available from the IdentityUser class, you can add a property to the User class. Then, you need to add a migration that contains the code for adding the new column in the AspNetUsers table, and you need to update the database.

For instance, the FirstName and LastName properties are not available from the IdentityUser class, so the User class shown in this figure adds those properties. To add the corresponding columns to the AspNetUsers database, you can use the Add-Migration command to add a migration for those columns to the project. Then, you can use the Update-Database command to execute that migration and add the columns to the database.

After the database is ready to handle the new registration fields, you need to modify the Registration view model and view to work with the new fields. For instance, the Registration view model shown in this figure adds the FirstName and LastName properties along with data attributes for validation. Although this figure doesn't show the code for the view, you should be able to modify the existing Account/Register view to add the rows that get the Email, FirstName, and LastName fields from the user.

## The updated User class

```
public class User : IdentityUser
{
    public string FirstName { get; set; } // needs to be added to DB
    public string LastName { get; set; } // needs to be added to DB

    [NotMapped]
    public IList<string> RoleNames { get; set; }
}
```

## The updated Register view model

```
public class RegisterViewModel
{
    [Required(ErrorMessage = "Please enter a username.")]
    [StringLength(255)]
    public string Username { get; set; }

    [Required(ErrorMessage = "Please enter a first name.")]
    [StringLength(255)]
    public string FirstName { get; set; }

    [Required(ErrorMessage = "Please enter a last name.")]
    [StringLength(255)]
    public string LastName { get; set; }

    [Required(ErrorMessage = "Please enter an email address.")]
    [DataType(DataType.EmailAddress)]
    public string Email { get; set; } // from IdentityUser class

    [Required(ErrorMessage = "Please enter a password.")]
    [DataType(DataType.Password)]
    [Compare("ConfirmPassword")]
    public string Password { get; set; }

    [Required(ErrorMessage = "Please confirm your password.")]
    [DataType(DataType.Password)]
    [Display(Name = "Confirm Password")]
    public string ConfirmPassword { get; set; }
}
```

## Description

- If you want to add a registration field that's available from the IdentityUser class, you can just add it to the Register view model. That's because the AspNetUsers table already contains the column needed to store the field.
- If you want to add a registration field that isn't available from the IdentityUser class, you can add a property to the User class. Then, you need to add a migration for the new column in the AspNetUsers table, and you need to update the database.
- After the database is ready to handle the new registration fields, you need to modify the Registration view model and view to work with the new fields.

---

Figure 16-24 How to add more user registration fields

## Perspective

Now that you've read this chapter, you should understand how to use Authorization attributes to restrict access to the pages of your web app. In addition, you should understand how to use ASP.NET Core Identity to authenticate users and allow users with appropriate authorization to access those restricted pages. There's still plenty more to learn about authorization and authentication, though. For example, if you have serious security concerns for your app, you may want to learn more about locking a user out after failed login attempts. Or, you may want to implement two-factor authentication. Whatever your authentication and authorization requirements are, this chapter should provide you with a solid foundation in these topics.

## Terms

authentication	OWIN (Open Web Interface for .NET)
authorization	role
Windows-based authentication	asynchronous method
individual user account authentication	thread
authentication cookie	synchronous method

## Summary

- If you want to limit access to all or part of your app to authorized users, you can use *authentication* to verify each user's identity.
- Once you have authenticated a user, you can use *authorization* to check if the user has the appropriate privileges to access a page.
- *Windows-based authentication* requires that you set up a Windows user account for each user. Then, you use standard Windows security features to restrict access to all or part of the app.
- *Individual user account authentication* uses a login page that typically requires the user to enter a username and password. ASP.NET displays this page automatically when it needs to authenticate a user who's trying to access the app.
- To be authenticated, the user request must contain an *authentication cookie*. By default, this cookie is stored as a session cookie, but it can also be stored as a persistent cookie.
- ASP.NET Identity is based on *OWIN (Open Web Interface for .NET)* middleware, which is an open-source project that defines a standard interface between .NET web servers and web apps.
- *Roles* let you apply the same access rules to a group of users.

- An *asynchronous method* can return control to the calling code before it finishes executing. This allows the calling code and the asynchronous method to execute simultaneously. To make this possible, an asynchronous method typically runs in a different *thread* than the calling code.
- A *synchronous method* typically runs in the same thread as the calling code. As a result, it must finish executing before the calling code can continue.

## Exercise 16-1 Review and improve the Bookstore web app

In this exercise, you'll review the authentication code in the Bookstore web app, update the database to include the Identity tables, test the authentication and authorization functionality, and add an admin user and some custom fields.

### Review the code

1. Open the Ch16Ex1Bookstore app in the ex\_starts folder.
2. In the Views/Shared folder, open the \_Layout file. In the Bootstrap navbar, note the Razor code that uses dependency injection and the SignInManager class to determine whether a user is logged in.
3. In the Models/DomainModels folder, open the User entity class. Note that it inherits the IdentityUser class and adds a RoleNames property that isn't mapped to the database.
4. In the Models/ViewModels folder, open the RegisterViewModel and LoginViewModel classes. Note the properties that these view models provide.
5. In the Controllers folder, open the AccountController class. Note that it gets UserManager and SignInManager classes by dependency injection and uses asynchronous action methods to register, log in, and log out.
6. In the Controllers folder, open the CartController class. Note that it's decorated with the Authorize attribute.
7. In the Areas/Admin/Controllers folder, open the Book controller. Note that it's decorated with the Authorize attribute and specifies the Admin role.

### Add the Identity tables to the database

8. In the Models/DataLayer folder, open the BookstoreContext file. Update it to inherit the IdentityDbContext<User> class.
9. Display the Package Manager Console (PMC) window and use the Add-Migration command to add a migration named AddIdentityTables. Then, review the migration file that's generated by this command.
10. Still in the PMC, run the Update-Database command to create the Identity tables. If you encounter errors at this step, you can start over by running the Drop-Database command. Then, you can run the Update-Database command again.

### Test the Cart and Admin pages

11. Run the app and click the Cart link. It should redirect to the Login page.
12. Click the “Register as a new user” link and enter the fields necessary to register with the app. This should log you in.
13. Click the Cart link again. This should display the Cart page.
14. Click the Admin link. This should display a message that indicates that access is denied for that page.
15. Click the Log Out button. This should log you out.

### Seed an admin user and role

16. In the BookstoreContext file, add a static asynchronous method named CreateAdminUser() as described in figure 16-22.
17. In the Program.cs file, modify the default configuration options as shown in figure 16-22.
18. In the Startup.cs file, at the end of the Configure() method, call the static CreateAdminUser() method and pass it the AppServices property of the IAppBuilder object. Since it’s an asynchronous method, be sure to chain the Wait() method at the end of this method call.
19. Run the app again and log in with a username of “admin” and a password of “Sesame”. Then, click the Admin link. This should display the Admin page.
20. Navigate to the Manage Users page. This should display the user you added earlier in this exercise. Experiment with this page. For example, you might want to add the user you created earlier in this exercise to the Admin role.

### Add fields to the Registration page

21. In the Models/DomainModels folder, open the User entity class and add string properties named Firstname and Lastname.
22. Open the Package Manager Console (PMC) and add a new migration for these new registration fields.
23. Use the PMC to update the database. This should add Firstname and Lastname columns to the AspNetUsers table in the database.
24. In the Sql Server Object Explorer, navigate to the AspNetUsers table and expand the Columns folder to confirm that this table now has Firstname and Lastname columns.
25. In the Models/ViewModels folder, open the RegisterViewModel class and modify it to add the Firstname, Lastname, and Email properties as shown in figure 16-24.
26. In the Controllers folder, open the Account controller and find the Register() action method for POST requests. In this method, update the code to add the Firstname, Lastname, and Email values to the User object.
27. In the Views folder, open the Account/Register view and add fields for the Firstname, Lastname, and Email properties.
28. Run the app and register a new user.

# How to use Visual Studio Code

This chapter shows how to use Visual Studio Code (also known as VS Code) with this book. This source code editor is an increasingly popular alternative to the Visual Studio IDE that has been presented so far in this book. Although VS Code doesn't provide as many features as the Visual Studio IDE, some programmers prefer the simplicity and speed that VS Code provides.

<b>How to work with existing projects .....</b>	<b>704</b>
How to install VS Code .....	704
How to open and close a project folder .....	704
How to view and edit files .....	706
How to run and stop a project.....	708
How to create the database for a project .....	710
A summary of .NET EF Core commands.....	712
<b>How to start a new project.....</b>	<b>714</b>
How to create a new project .....	714
How to add NuGet packages to a project .....	714
How to work with the folders and files.....	716
How to install and manage client-side libraries .....	718
<b>How to debug a project.....</b>	<b>720</b>
How to set a breakpoint .....	720
How to work in break mode .....	722
<b>Perspective .....</b>	<b>724</b>

## How to work with existing projects

This chapter begins by showing how to work with existing projects, such as the projects for this book that you can download from murach.com. These projects were originally created with the Visual Studio IDE, but it's easy to use VS Code to work with them. But first, if VS Code isn't already installed on your system, you need to install it.

### How to install VS Code

Installing VS Code is similar to installing Visual Studio as described in appendixes A (Windows) and B (macOS). The main difference is that you begin by searching the Internet for “Visual Studio Code download”. Then, you download the installer file, run it, and respond to the resulting dialog boxes.

### How to open and close a project folder

Once you have installed VS Code, you can start it and use it to open any existing projects. This includes projects that were created by the Visual Studio IDE. Figure 17-1 shows how to open a Visual Studio project with VS Code.

To do that, you can use the File→Open Folder item that's available from the menu system. Then, you can use the resulting dialog to select the folder for the project. For example, to open the Future Value app from chapter 2, you select the FutureValue folder for the project that's within the Ch02FutureValue folder for the solution. This is possible because VS Code doesn't need a solution or project file to store configuration information like Visual Studio does. Instead, when you open a project folder, VS Code provides access to all the source code files and other resources that it contains.

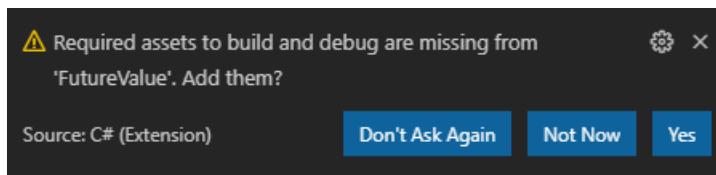
Once you've opened a project folder, VS Code displays its folders and files in the Explorer window that's on its left side of the main window. In addition, VS Code may prompt you with a warning dialog in the lower-right corner of the main window. If you encounter a dialog like this, you typically want to respond with an affirmative answer such as Yes or Restore.

If you use the dialog shown in this figure to add the required assets, VS Code creates a new subfolder named .vscode. This folder typically contains two JSON files named launch.json and tasks.json, and these files contain the configuration that's needed by VS Code to run a project as described later in this chapter.

When you use VS Code to open a folder, it's important to close the folder when you're done with it. To do that, you can use the File→Close Folder item that's available from the menu system.

Most of the time, that's all you need to use VS Code to work with an existing Visual Studio project. However, if you need to work with a Visual Studio solution that contains multiple projects, or if you want to save special configurations and settings related to your projects, you can use the File→Save Workspace As item to save a configuration file.

## A typical error message after opening a Visual Studio project



### How to open a project folder

1. Start VS Code.
2. Select File→Open Folder from the menu system.
3. Use the resulting dialog to select the folder that contains the Visual Studio project.  
For the Future Value app from chapter 2, you can select this folder:  
`\murach\aspnet_core_mvc\book_apps\Ch02FutureValue\FutureValue`
4. Click Select Folder.

### How to fix errors after opening a Visual Studio project

- If you get a dialog that indicates that “Required assets to build and debug are missing” and asks if you would like to add them, click Yes. This adds a .vscode subfolder to your project folder that contains .json files with VS Code configurations.
- If you get a dialog that says “There are unresolved dependencies”, click Restore.
- If VS Code still displays many errors, including underlined code, close VS Code and start it again. The problems should go away when VS Code rereads its configuration files.

### How to close a project folder

- Select File→Close Folder from the menu system.

### Description

- *Visual Studio Code* (also known as *VS Code*) is a *source code editor*, which is simpler than an integrated development environment (IDE) like Visual Studio.
- Installing VS Code is similar to installing Visual Studio as described in appendixes A (Windows) and B (macOS), except that you begin by searching the Internet for “Visual Studio Code download”.
- To open a Visual Studio project with VS Code, you can open the folder for the Visual Studio project.

---

Figure 17-1 How to open and close a project folder

Then, if you want, you can use the File→Add Folder To Workspace item to add a second project folder to the workspace. When you’re done with the workspace, you can use the File→Close Workspace item to close the workspace and all of its folders. Later, you can use the File→Open Workspace item to open the workspace again.

## How to view and edit files

---

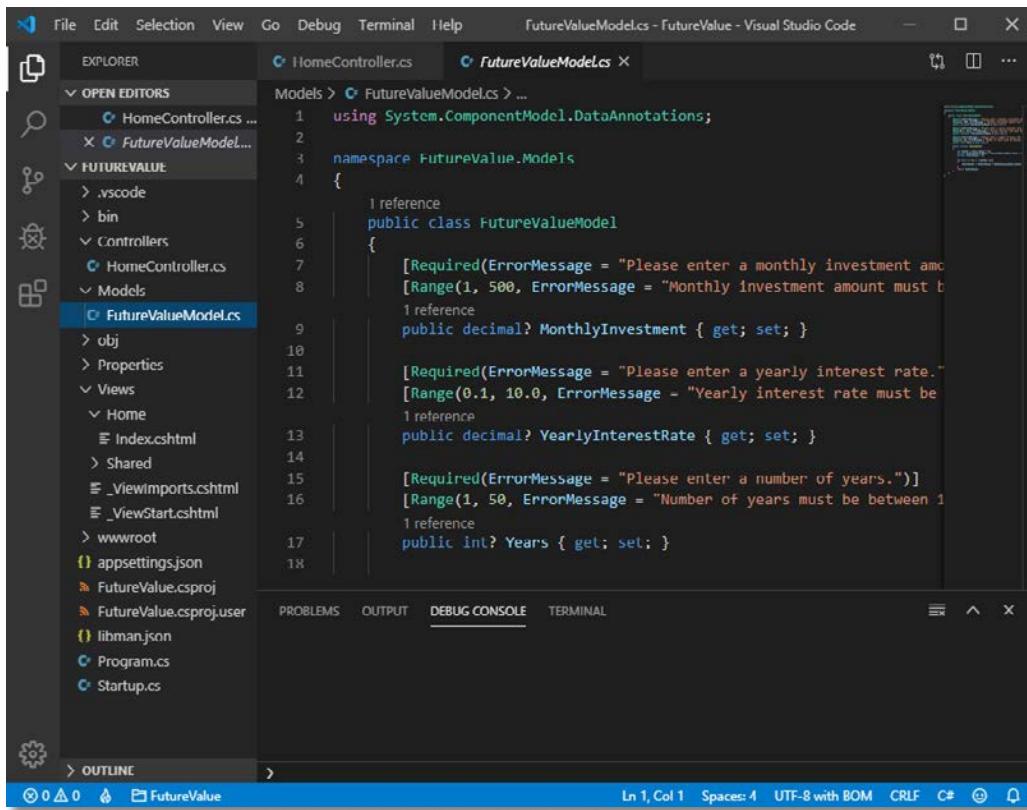
After you open a project folder, you can use VS Code to view and edit the files in the project. This works much like using the Visual Studio IDE. However, VS Code provides three modes for viewing and editing files.

When you click on a file in the Explorer window on the left side of the main window, VS Code displays the file in Preview mode. This displays the file in a tab with the name of the file in italics. In figure 17-2, for example, the second tab displays the file for the FutureValueModel class in Preview Mode. If you click on the name of a different file, VS Code loads that file in the same tab of the editor. This is an excellent way to quickly view and edit various files, especially those files that you don’t want to keep open for a long time.

When you double-click a file in the Explorer, VS Code displays the file in Standard Mode. This displays the file in a tab with the name of the file in normal, non-italic font. In this figure, for example, the first tab displays the file for the HomeController class in Standard Mode. While Preview Mode is ideal for switching between multiple files and spending a short time in each of them, Standard Mode is designed to be used for files that you want to keep open indefinitely.

Zen Mode removes all user interface components and only displays the editor. Some developers feel that the lack of user interface distractions helps them focus on the code and leads to greater productivity.

## VS Code with files in Standard and Preview Modes



### Three modes

Mode	Useful for...
Preview	Switching between files to quickly view or edit them.
Standard	Opening a file indefinitely for viewing and editing.
Zen	Focusing on editing a file's code without distraction of other interface elements.

### Description

- To preview a file, click on it in the Explorer window. This displays the file in a tab in the editor with the name of the file in italics, indicating that you are in Preview Mode. If you click another other file, VS Code reuses the tab.
- To open a file, double-click on it in the Explorer window. This displays the file in a tab in the editor with the name of the file in normal font style, indicating that you are in Standard Mode.
- To work on a file in Zen mode, open the file, click inside it, press **Ctrl+K**, release both keys, and immediately press **Z** for Zen. This displays the file in the editor without any other part of the VS Code interface. To exit Zen Mode, press the **Esc** key *twice*.

Figure 17-2 How to view and edit files

## How to run and stop a project

---

You can run an app in several ways from within VS Code. For example, you select the Debug→Start Without Debugging item from the menu system. However, you can also press Ctrl+F5 to accomplish the same task. This works much like it does for the Visual Studio IDE.

When you run a web app like the ones used throughout this book, VS Code starts the web server and begins listening for requests issued for your web app. This is shown by the output that VS Code displays in its Debug Console. In figure 17-3, for example, the Debug Console below the editor shows that it is starting the Kestrel web server and listening for requests here:

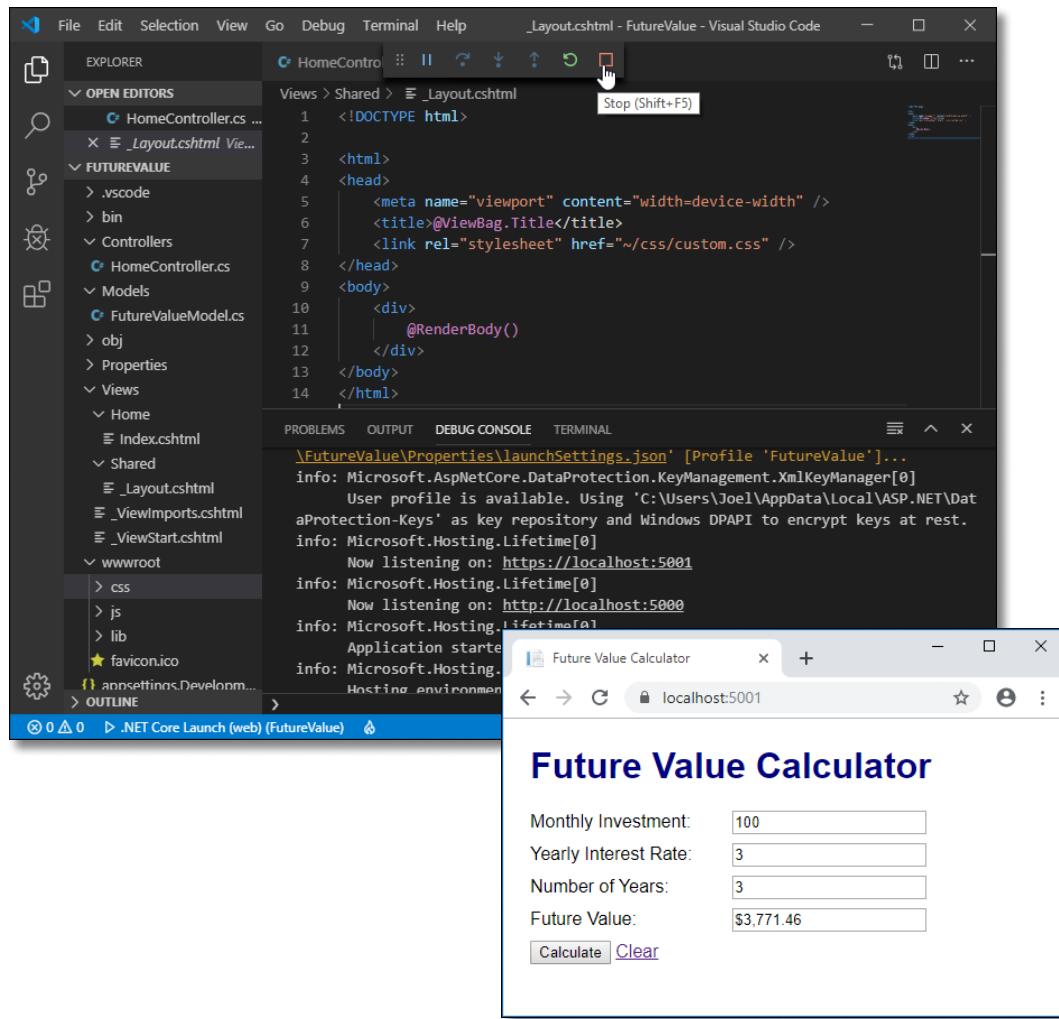
`https://localhost:5001`

In addition, VS Code automatically starts the default web browser for your operating system and navigates to the URL shown above. This displays the web app in your default browser. For example, if you use VS Code to run the Future Value app from chapter 2, it automatically displays that app in a browser as shown in this figure.

Note that if this is the first time you've run one of the apps for this book, a message may be displayed asking if you want to trust the ASP.NET Core SSL certificate, which is used to secure the localhost server. Trusting this certificate allows you to test your app locally on your own machine without your browser displaying this message. Or, if you prefer, you can tell the browser to ignore the warnings and proceed anyway.

You can also stop an app in several ways from within VS Code. For example, you can click the Stop button in the toolbar that's displayed across the top of the main window. Or, you can press Shift+F5. This stops the Kestrel server, but doesn't close the web browser. Conversely, closing the web browser doesn't stop the server.

## VS Code with the app running in a browser



### Description

- To run an app in the default browser for your operating system, press Ctrl+F5. This starts the app without debugging and automatically launches the browser.
- To stop an app, press Shift+F5 or click the Stop button in the toolbar.
- When VS Code runs the app on the Kestrel server, it uses the Debug Console window to display information about the server including the ports that it is using.
- If you get messages about trusting and installing an SSL certificate, you can click Yes. And if a web page is displayed indicating that the connection is not private, you can click the link to proceed.

Figure 17-3 How to run and stop a project

## How to create the database for a project

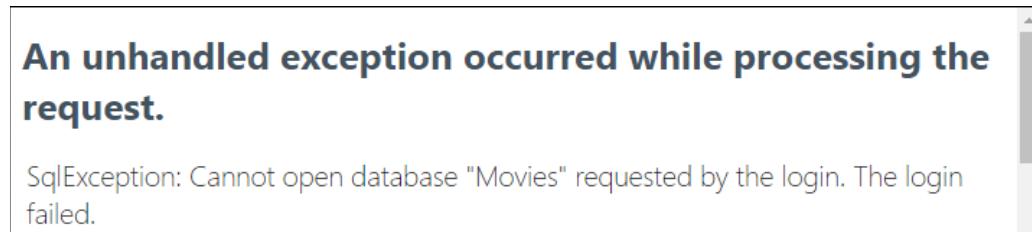
---

If you run a project, and you get an error message like the one that's shown at the top of figure 17-4, it's probably because you need to create the database that the project uses. To use VS Code to create the database for a project, you can use its Terminal window. This window provides access to a *command-line interface (CLI)* that allows you to execute commands by entering a line of text at the command prompt.

If the CLI tools for .NET EF Core aren't already installed on your system, you can follow the first procedure shown in this figure to install them. Then, you can use the second procedure to execute a .NET EF Core command that creates the database. This works much like the Update-Database command that's described in chapters 4 and 12.

After you've used VS Code's Terminal window to create the database for a project, you can run the app again. This time, the app should be able to access the database and run successfully.

### The error message that's displayed if the database hasn't been created



### A command prompt for the Ch04MovieList app

A screenshot of a terminal window titled "1: powershell". The window shows a Windows PowerShell session with the following text:

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

PS C:\murach\aspnet_core_mvc\book_apps\Ch04MovieList\MovieList> dotnet ef database update
```

The terminal window has a standard OS X-style interface with tabs for PROBLEMS, OUTPUT, DEBUG CONSOLE, and TERMINAL, and a toolbar with file and terminal control icons.

### How to install the CLI tools for .NET EF Core

1. Display the Terminal by selecting Terminal→New Terminal.
2. At the command prompt, enter this command:  
`dotnet tool install --global dotnet-ef`

### How to create the database for a project

1. Display the Terminal by selecting Terminal→New Terminal.
2. At the command prompt, enter this command:  
`dotnet ef database update`

### Description

- If a project uses a database, you may need to create the database before the project can run successfully.
- A *command-line interface (CLI)* allows you to execute commands by entering a line of text at the command prompt.
- When you use VS Code, you often need to use the command line provided by the Terminal window to execute CLI commands such as the .NET EF Core commands.
- Before you can use the .NET EF Core commands, you need to install the CLI tools for EF Core. You only need to do this once.

---

Figure 17-4 How to create the database for a project

## A summary of .NET EF Core commands

Figure 17-5 starts by reviewing the command that you need to install the CLI tools for .NET EF Core. Then, it presents five of the .NET EF Core commands. If you've read chapters 4 and 12, you should realize that these commands correspond to the PowerShell commands that are available from Visual Studio's Package Manager Console (PMC). That's because the CLI commands just provide a different way to access the same EF Core commands as the PMC.

When you use a CLI command for .NET EF Core, you preface the command with `dotnet ef`. For example, you can update the database by entering this command:

```
dotnet ef database update
```

In addition, you may sometimes need to include a parameter after the name of the command. For example, to add a migration named `Initial`, you can enter this command:

```
dotnet ef migrations add Initial
```

If you understand how to use the PowerShell commands described in chapters 4 and 12, you shouldn't have much trouble understanding how to convert most of those commands to CLI commands. However, some commands require several arguments that you must specify as well as one or more options that you can specify. As a result, it can take some effort to figure out how they work.

In this figure, for instance, the last example presents the `dbcontext scaffold` command. This command begins by specifying two required arguments. The first one specifies that the connection string is stored in the `appsettings.json` file with a name of `BookstoreContext`. The second one specifies that the `DbContext` provider is EF Core for SQL Server.

After the two required parameters, this example uses an option to specify the output directory like this:

```
--output-dir Models/DataLayer
```

Here, the output directory is the `DataLayer` subfolder of the `Models` folder. In addition, to make this option easy to read, it uses two dashes to specify the long version of the option's name. However, you can use a single dash and a short version of the option's name like this:

```
-o Models/DataLayer
```

After the first option, the second option specifies that the command should use attributes (not the Fluent API) to configure the model whenever possible. And the third option specifies that the command should overwrite any existing files. Again, both of these options use two dashes and the long version of the name, although they could use a single dash and the short version of the name.

For more information about the commands shown in this figure, including detailed descriptions of their arguments and options, you can search the Internet for “`dotnet ef commands`”. This should lead to the online documentation for these commands.

## A command that installs the CLI tools for .NET EF Core

```
dotnet tool install --global dotnet-ef
```

## Some of the .NET EF Core commands (prefix with dotnet ef)

Command	Description
<code>migrations add</code>	Adds the migration file with the specified name.
<code>migrations remove</code>	Removes the last migration file from the Migrations folder. Only works with migrations that have not yet been applied with the database update command.
<code>database update</code>	Updates the database to the last migration or to the migration specified by the optional name argument.
<code>database drop</code>	Deletes the database.
<code>dbcontext scaffold</code>	Generates DB context and entity classes from an existing database. This command uses arguments to specify the connection string and database context provider. In addition, it provides other options that control how this command works.

### Two commands that add migrations

```
dotnet ef migrations add Initial
dotnet ef migrations add Genre
```

### A command that updates the database to the last migration

```
dotnet ef database update
```

### A command that updates the database to the specified migration

```
dotnet ef migrations update Initial
```

### A command that removes the last migration

```
dotnet ef migrations remove
```

### A command that drops the database

```
dotnet ef database drop
```

### A command that generates DB classes from an existing database

```
dotnet ef dbcontext scaffold name=BookstoreContext
Microsoft.EntityFrameworkCore.SqlServer --output-dir Models/DataLayer
--data-annotations --force
```

## Description

- The command-line interface (CLI) for .NET EF Core can access all of the same EF Core functionality as Visual Studio's Package Manager Console (PMC).
- For more information about the arguments and options for these commands, you can view the online documentation here:

<https://docs.microsoft.com/en-us/ef/core/miscellaneous/cli/dotnet>

---

Figure 17-5 A summary of .NET EF Core commands

## How to start a new project

So far, you have learned how to use VS Code to open an existing project such as a project created by Visual Studio. That allows you to work with Visual Studio projects like the ones included with the download for this book. Now, you'll learn how to use VS Code to start a new project.

### How to create a new project

Figure 17-6 shows how to use VS Code to create a new project for an ASP.NET Core MVC web app. To start, you can use your operating system to create a root folder to store the project. Then, you can use VS Code to open that root folder as described earlier in this chapter. When you do, VS Code should display the empty root folder in its Explorer window.

After you use VS Code to open the empty root folder, you can open a Terminal and use the CLI to execute a command that creates the starting structure for the app. In this figure, the command in step 5 creates a new project that's based on the ASP.NET Core Web Application (Model-View-Controller) template that's described in chapter 2.

However, if you wanted to create a new project that's based on the ASP.NET Core Empty template, you could enter a command like this:

```
dotnet new web
```

This template also works as described in chapter 2.

The command-line interface (CLI) can access all of the same templates as Visual Studio. The table in this figure only shows the two templates presented in this book. For a complete list of templates, you can search the Internet for “dotnet new command” to find the online documentation for this command.

After you use a template to create the starting folders and files for a new project, you can work with it like any other project. For example, you can run the project, edit existing code, add new code files, and so on.

### How to add NuGet packages to a project

When you want to add a NuGet package to a VS Code project, you can use the CLI to do that. For example, to create a new project that uses EF Core to work with a SQL Server database, you typically want to use the commands shown in figure 17-6 to add the packages that a project needs to work with EF Core. Similarly, if you want to use the Newtonsoft JSON library in your project, you can use the command shown in this figure to add the package for that library to your new project.

## How to create a new project for an ASP.NET Core MVC web app

1. Use your operating system to create a new root folder for your project.
2. Start VS Code.
3. Select File→Open Folder and use the resulting dialog to select the root folder for the project.
4. Select Terminal→New Terminal to open a Terminal window.
5. Enter the following command:

```
dotnet new mvc
```

## Two templates you can use to create ASP.NET Core projects

Template name	CLI argument
Web Application (Model-View-Controller)	<code>mvc</code>
Empty	<code>web</code>

## How to use the CLI to add NuGet packages to the project

### Two commands that add EF Core packages

```
dotnet add package Microsoft.EntityFrameworkCore.SqlServer  
dotnet add package Microsoft.EntityFrameworkCore.Design
```

### A command that adds the Newtonsoft JSON package

```
dotnet add package Microsoft.AspNetCore.Mvc.NewtonsoftJson
```

## Description

- The command-line interface (CLI) can access all of the same templates as Visual Studio. The table in this figure shows both of the templates presented in this book.
- The dotnet add package command adds a NuGet package reference to the project file and then runs the dotnet restore command to install the package.
- After creating a new project, you may need to close the project folder and open it again to get VS Code to display the dialog that asks if you want to add required assets for building and debugging a project. Then, you can click Yes to add these assets. These assets allow you to run a project as described earlier.

---

Figure 17-6 How to create a new project and add NuGet packages to it

## How to work with the folders and files

---

When you open an existing project, all the folders and files you need to work with may already exist. In that case, you can use VS Code to view and edit these folders and files as described earlier in this chapter.

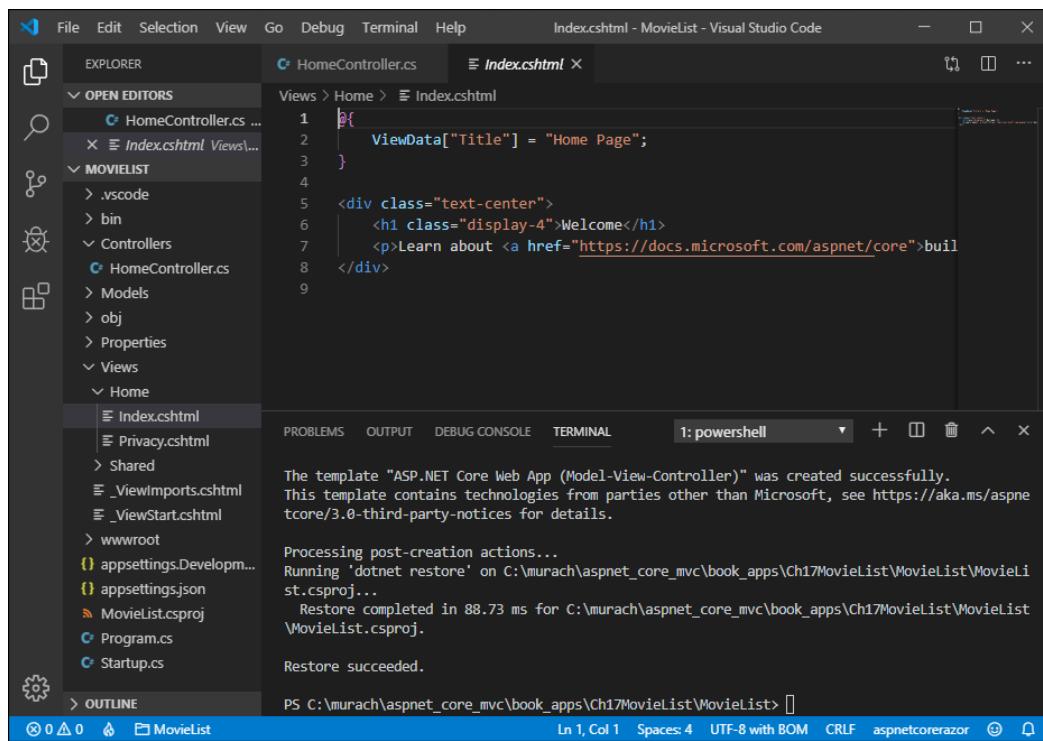
However, when you create a new project, you typically need to work with the project's folders and files as described in figure 17-7. For example, to add a folder to the main folder for the project, you can point to the name of the project and click the New Folder button that's displayed to the right of the name. To add a folder to any other folder, you can right-click on the folder in the Explorer and select New Folder. Either way, you can enter a name for the folder after clicking New Folder.

Once a folder exists, you can add a file to the folder by right-clicking on it and selecting New File. Then, you can enter the name for the file. When you enter a name for a file, you should include an extension that identifies the type of file that you're creating. For example, you can use an extension of .cs for C# files, .cshtml for Razor view files, .css for CSS files, .js for JavaScript files, and so on.

When you use VS Code to add a new file, it doesn't generate any starting code for the file. As a result, you must enter all the code for the file yourself. Or, if you prefer, you can copy and paste some starting code from another file. For example, if you create a new file for a controller, you might want to copy some starting code from another controller and paste it into the new controller. Then, you can edit that code so it works for the new controller.

If you're used to relying on the code that's generated by Visual Studio, this might seem intimidating at first. But once you get used to it, you might begin to prefer the simplicity of the blank file that's provided by VS Code.

## VS Code after some starting folders and files have been added



## Description

- To add a folder to the project folder, you can point to the name of the project and click the New Folder icon that's displayed to its right. Then, you can enter a name for the folder.
- To add other folders, you can right-click on a folder in the Explorer window and select New Folder. Then, you can enter a name for the folder.
- To add a file, you can right-click on a folder in the Explorer window and select New File. Then, you can enter the name for the file, including its extension.
- Unlike Visual Studio, VS Code doesn't generate any starting code for new files. As a result, you must enter all code for the file yourself.
- To rename a folder or file, right-click on the folder or file and select Rename. Then, edit the name.
- To delete a folder or file, right-click on the folder or file and select Delete.

Figure 17-7 How to work with the folders and files

## How to install and manage client-side libraries

In chapter 3, you learned how to use the graphical user interface (GUI) for LibMan that's available from Visual Studio to install and manage client-side libraries such as Bootstrap and jQuery. Now, figure 17-8 shows how to use the command-line interface (CLI) for LibMan to install and manage client-side libraries.

To start, the first example shows how to install the CLI tools for LibMan. You only need to do this if these tools aren't already installed on your system. In other words, you only need to run this command once.

After you install the CLI tools for LibMan, you can create the libman.json file that stores information about each client-side library that's being managed. Again, you only need to do this once.

After you create the libman.json file, you can use VS Code to open this file. Then, you can edit this file so it includes the name, version number, and destination for each client-side library. For example, when you create the libman.json file, it usually generates this JSON code:

```
{  
  "version": "1.0",  
  "defaultProvider": "cdnjs",  
  "libraries": []  
}
```

Within the square brackets, you can add one or more libraries, separated by commas. In this figure, for example,

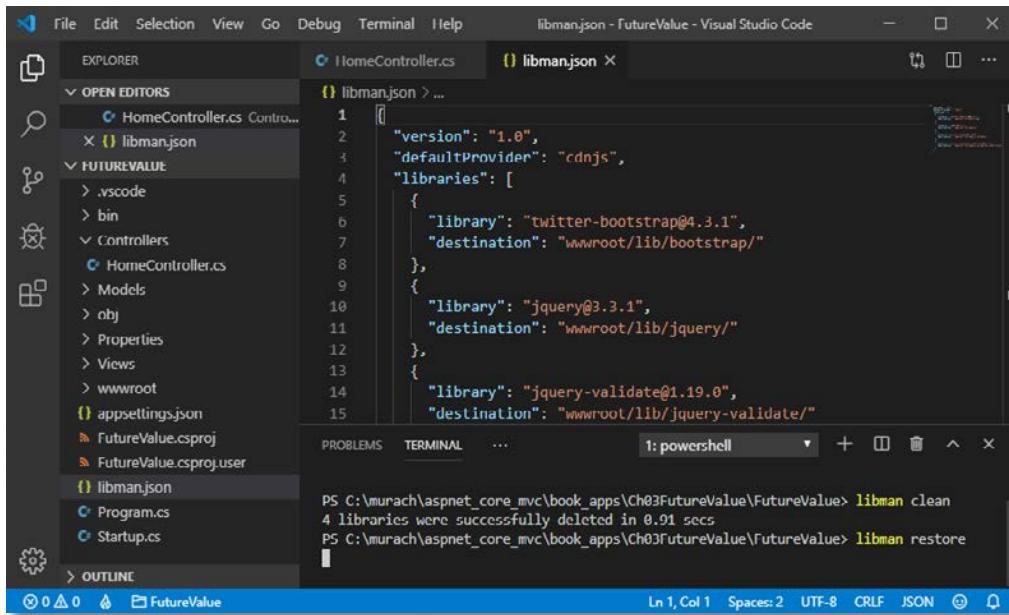
```
"libraries": [  
  {  
    "library": "twitter-bootstrap@4.3.1",  
    "destination": "wwwroot/lib/bootstrap/"  
  }  
]
```

specifies that LibMan should install version 4.3.1 of the Twitter Bootstrap library in the wwwroot/lib/bootstrap/ folder for this project.

After you've edited the libman.json file so it specifies the client-side libraries that you want to use, you can install or update that library by running the libman restore command. Or, if you want to delete all client-side libraries managed by LibMan, you can run the libman clean command.

If necessary, you can manually delete folders that contain client-side libraries that you don't want to use. To do that, you typically begin by expanding the wwwroot/lib folder to view the folder for the library. Then, you can right-click on the library's folder, select Delete, and respond to the resulting dialog.

## VS Code using the CLI tools for LibMan to install client-side libraries



### A command that installs the CLI tools for LibMan

```
dotnet tool install --global Microsoft.Web.LibraryManager.Cli
```

### How to create a libman.json file for a project

- At the Terminal command prompt for the project, enter this command:  
**libman init**
- Press Enter to accept the default provider of CDNJS.

### How to use LibMan to install client-side libraries

- Open the libman.json file and edit it to include the client-side libraries you want to install as described in chapter 3.
- At the Terminal command prompt, enter this command:  
**libman restore**

### How to delete all client-side libraries installed by LibMan

- At the Terminal command prompt, enter this command:  
**libman clean**

### Description

- The command-line interface (CLI) can use LibMan to manage client-side libraries in a way that's similar to the graphical user interface (GUI) for LibMan that was presented in chapter 3.

---

Figure 17-8 How to install and manage client-side libraries

## How to debug a project

---

In chapter 5, you learned how to use Visual Studio's debugger to debug a project. Now, you'll learn how to use VS Code's debugger to perform the same task. These two debuggers work similarly, though the VS Code debugger doesn't provide as many features. Still, some programmers prefer the simplicity of the VS Code debugger.

### How to set a breakpoint

---

Figure 17-9 shows how to use VS Code to set a *breakpoint* in an ASP.NET Core app. To start, you can set a breakpoint before you run an app or as an app is executing. However, a web app ends after it generates a page. So, if you switch from the browser to VS Code to set a breakpoint, the breakpoint won't be taken until the next time the page is executed. As a result, if you want a breakpoint to be taken the first time a page is executed, you need to set the breakpoint before you run the app.

After you set a breakpoint and run the app, the app enters *break mode* before it executes the statement that contains the breakpoint. In this figure, for example, the app will enter break mode before it executes the statement that saves changes to the database. Then, you can use the debugging features to debug the app.

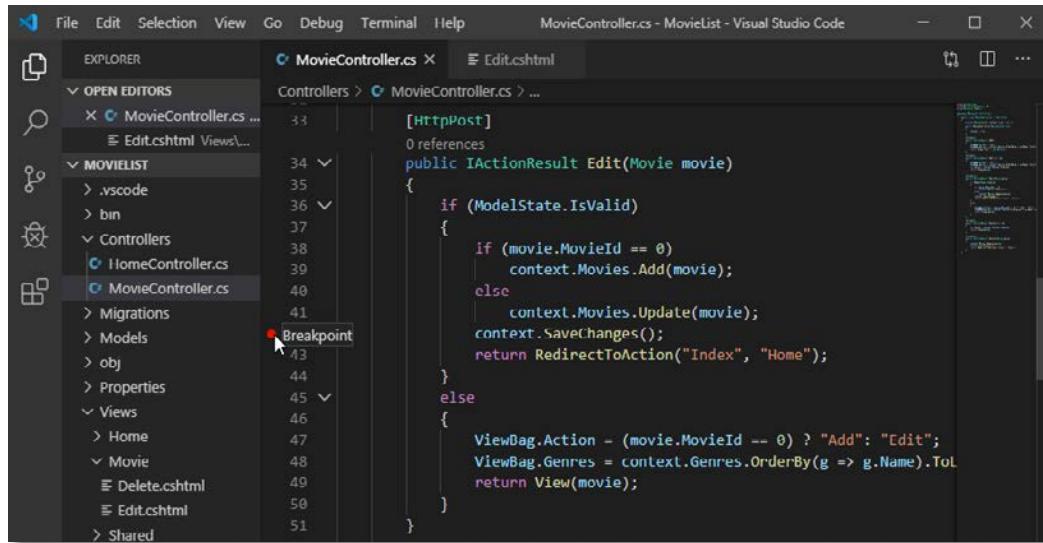
In some cases, you may want to set more than one breakpoint. You can do that either before you begin the execution of the app or while the app is in break mode. Then, when you run the app, it stops at the first breakpoint. And when you continue execution, the app executes up to the next breakpoint.

Once you set a breakpoint, it remains active until you remove it. In fact, it remains active even if you close the project and open it again later. If you want to remove a breakpoint, you can use one of the techniques presented in this figure.

If you don't want to remove a breakpoint completely but you don't want to stop on it, you can disable it by right-clicking on it and selecting the Disable Breakpoint item. Then, if you later want to stop on that breakpoint, you can enable it by right-clicking on it and selecting the Enable Breakpoint item.

One easy way to enable and disable breakpoints is to use the Debug menu. This menu includes items that let you enable or disable all breakpoints.

## The Movie controller with a breakpoint



### How to set and remove breakpoints

- To set a breakpoint, click in the margin indicator bar to the left of the line number for a statement. This highlights the statement and adds a breakpoint indicator (a red dot) in the margin.
- To remove a breakpoint, click the breakpoint indicator.
- To remove all breakpoints, select Debug→Remove All Breakpoints.

### How to enable and disable breakpoints

- To disable a breakpoint, right-click it and select Disable Breakpoint.
- To enable a breakpoint, right-click it and select Enable Breakpoint.
- To disable all breakpoints, select Debug→Disable All Breakpoints.
- To enable all breakpoints, select Debug→Enable All Breakpoints.

### Description

- When ASP.NET Core encounters a *breakpoint*, it enters *break mode* before it executes the statement on which the breakpoint is set.
- You can set and remove breakpoints before you run an app or while you're in break mode.
- You can only set a breakpoint on a line that contains an executable statement.

---

Figure 17-9 How to set a breakpoint

## How to work in break mode

To work in break mode, you must run your app with debugging. To do that, you can press F5. Or, you can select Debug→Start Debugging from the menu system. Then, you can test your app. When the app hits a breakpoint, it enters break mode.

Figure 17-10 shows the Movie controller of the Movie List app in break mode. In this mode, the next statement to be executed is highlighted. Then, you can use the debugging information that's available to try to determine the cause of an exception or a logical error.

One way to get information about a variable is to view it in the Variables window that's displayed on the left side of the main window. This window displays information about the variables within the scope of the current method. If the code in a controller is currently executing, this window also includes information about the controller and all of its properties such as its ViewBag property. In this figure, for example, the Variables window shows the data for the Movie object that's stored in the variable named movie. This includes the values for its properties such as its Genre property, whose value is null.

Below the Variables window, the Watch window lets you specify an expression that you want to watch. For example, the Watch window in this figure includes an expression that displays the Name property of the Movie object. To add an item to a Watch window, hover the mouse pointer over the title bar for this window to display an Add (+) button. Then, click on the Add (+) button and enter the expression you want to watch.

Once you're in break mode, you can use the Step Into, Step Over, and Step Out buttons that are available from the Debug toolbar to control the execution of the app. Or, you can use the shortcut keys for these buttons. For example, the tooltip in this figure shows that the Step Into button has a shortcut key of F11.

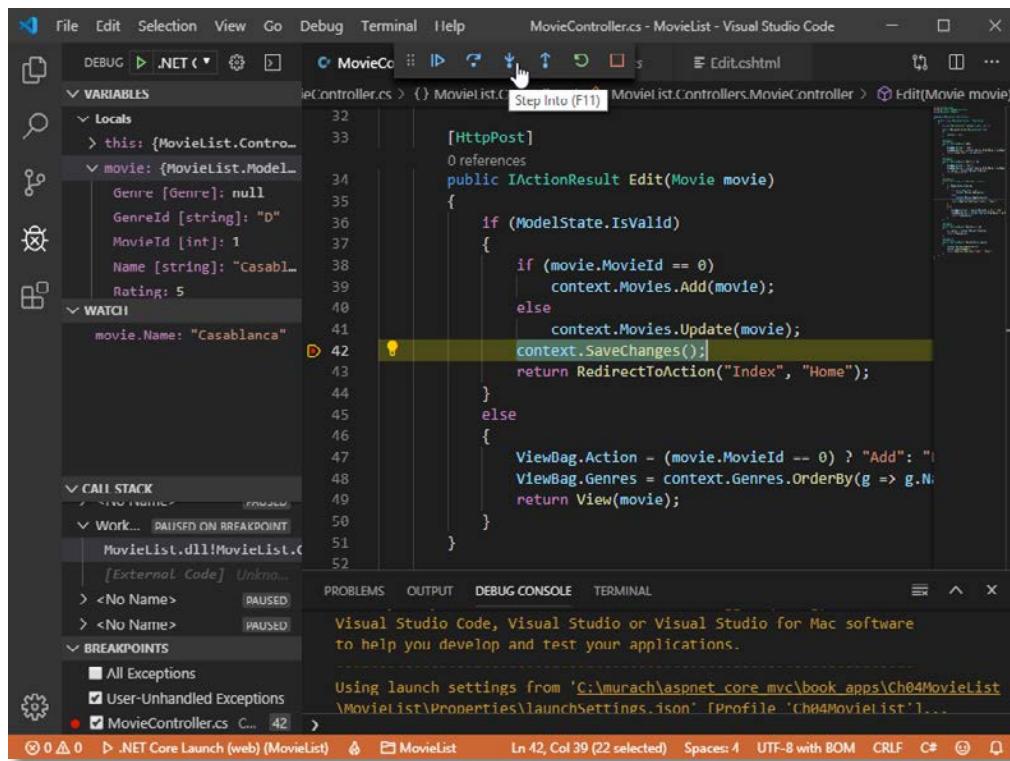
To execute the statements of an app one statement at a time, you can use the Step Into button. Each time you use this button, the app executes the next statement and returns to break mode. That way, you can check the values of variables and properties as you step through code one statement at a time. The Step Over button is similar to the Step Into button, but it executes the statements in called methods without interruption (they are "stepped over").

The Step Out button executes the remaining statements in a method without interruption. It's common to use this button if you've stepped into a method that you don't want to step through. In that case, you can use the Step Out button to execute the rest of the statements in the method. Then, the app enters break mode before the next statement in the calling method is executed.

The Continue button continues execution of the app with debugging. As a result, the app continues to run unless it hits another breakpoint and enters break mode again.

When you're done debugging, you can click the Stop button on the far right side of the Debug toolbar. Or, you can press Shift+F5. This works the same as stopping an app when you run it without debugging as described earlier in this chapter. Then, you can click the Explorer button on the far left side of the window to display the Explorer window instead of the debugging windows.

## The Movie List app in break mode



## How to run an app with debugging

- Press F5.
- Select Debug→Start Debugging.

## Description

- When you enter break mode, the debugger highlights the next statement to be executed. In addition, it displays the debugging windows on the left side of the main window, and it displays the Debug toolbar across the top of the main window.
- You can use the Step Into (F11), Step Over (F10), and Step Out (Shift+F11) buttons of the Debug toolbar to execute one or more statements and return to break mode.
- You can use the Continue (F5) button of the Debug toolbar to continue running the app with debugging.
- You can use the Variables window to view information about the variables and properties within the scope of the current method.
- You can use the Watch window to view the values of expressions that you specify.
- When you're done with the debugging windows, you can display the Explorer window again by clicking Explorer in the vertical toolbar on the far left side of the main window.

Figure 17-10 How to work in break mode

## Perspective

This chapter has presented the basic skills for using VS Code to work with ASP.NET Core MVC apps. This includes many skills for using the command-line interface (CLI) commands to work with ASP.NET Core. If you've already been using Visual Studio to work with ASP.NET Core MVC apps as described in this book, you shouldn't have any trouble using VS Code to perform the same tasks. And if you experiment with VS Code, you may find that you prefer it to Visual Studio for some tasks.

On the other hand, if you skipped straight to this chapter from chapter 1 and you don't have experience with Visual Studio, it might be a little more difficult for you to get started with VS Code. Still, with a little help from this chapter, you should be able to use VS Code to create and run the apps described in this book. For example, you should be able to use VS Code to create and run the Future Value app described in chapter 2. Similarly, you should be able to use VS Code to create and run the Movie List app described in chapter 4. And so on.

## Terms

Visual Studio Code  
VS Code  
source code editor

command-line interface (CLI)  
breakpoint  
break mode

## Summary

- *Visual Studio Code* (also known as *VS Code*) is a *source code editor*, which is simpler than an integrated development environment (IDE) like Visual Studio.
- A *command-line interface (CLI)* allows you to execute commands by entering a line of text at the command prompt.
- When ASP.NET Core encounters a *breakpoint*, it enters *break mode* before it executes the statement on which the breakpoint is set.

### Exercise 17-1 Experiment with VS Code

In this exercise, you'll use VS Code to open and run some projects that are included with the download for this book. Then, you'll learn how to create the Future Value app from chapter 2 from scratch.

#### Run the Future Value app from chapter 2

1. Open the FutureValue folder that's in the book\_apps/Ch02FutureValue folder.
2. If you get any error dialogs about missing assets or unresolved dependencies, click on the appropriate buttons to add the assets and resolve the dependencies.

3. Double-click on the file for the HomeController class to open it in Standard Mode. Note that its filename is displayed in a normal font on its tab.
4. Click on the file for the FutureValueModel class to view it in Preview Mode. Note that its filename is displayed in italics on the tab.
5. Click on the file for the Home/Index view to view it in Preview Mode. Note that it reuses the tab that was used by the previous file.
6. Press Ctrl+K, release both keys, and press Z to enter Zen Mode. Note that this removes all user interface components except the editor. Then, press Esc twice to exit Zen mode.
7. Press Ctrl+F5 to run the project without debugging. This should display some messages in the Debug Console and automatically start the app in the default web browser for your operating system.
8. In your browser, use the app to calculate a future value. Then, close the browser.
9. In VS Code, click the Stop button to stop the app. Then, close the project folder.

### Run the Movie List app from chapter 4

10. Open the MovieList folder that's in the book\_apps/Ch04MovieList folder.
11. If you get any error dialogs about missing assets or unresolved dependencies, click on the appropriate buttons to add the assets and resolve the dependencies.
12. Use VS Code to view some of the source code files for the Movie List app. Note that it contains files for working with a database named Movies.
13. Display the Terminal window.
14. Use the dotnet tool install command to install the CLI tools for .NET EF Core.
15. Use the database update command to make sure the Movies database has been created.
16. Press Ctrl+F5 to run the project without debugging. This should display some messages in the Debug Console and automatically start the app in the default web browser for your operating system.
17. In your browser, use the app to view some movies and add a movie. Then, close the browser.
18. In VS Code, click the Stop button to stop the app. Then, close the project folder.

### Create the Future Value app from scratch

19. Use your operating system to create a folder named Ch17FutureValue in the ex\_solutions folder. Within that folder, create a subfolder named FutureValue.
20. Use VS Code to open the FutureValue subfolder.
21. Start a Terminal window.

22. Use the dotnet new mvc command to add the files for the ASP.NET Web Application (Model-View-Controller) template.
23. Close the project folder and open it again. This should display a dialog that asks if you want to add required assets for building and debugging, and you should click Yes. This should add the .vscode folder to your project.
24. Press Ctrl+F5 to run the project without debugging. This should display some messages in the Debug Console and automatically start the app in the default web browser for your operating system.
25. In your browser, view the page that's displayed by the template for the MVC web app. Then, close the browser.
26. In VS Code, click the Stop button to stop the app.
27. In the Models folder, add a file named FutureValueModel.cs and enter the code for the FutureValueModel class that's shown in chapter 2. To do that, you can copy the code from the Ch02FutureValue app that's in the book\_apps folder.
28. Open the HomeController.cs file and enter the code for the Home controller that's shown in chapter 2. Again, you can copy this code from the app for chapter 2 if you want.
29. In the wwwroot/css folder, add a file named custom.css and enter the code for the custom CSS file that's shown in chapter 2.
30. Open the \_Layout.cshtml file and enter the code for the layout that's shown in chapter 2.
31. Open the Home/Index.cshtml file and enter the code for the Home/Index view that's shown in chapter 2.
32. Press Ctrl+F5 to run the project without debugging. This should start the app and display it in the default browser for your operating system.
33. In your browser, use the app to calculate a future value. Then, close the browser and stop the app.

### Set a breakpoint and step through code

34. In the FutureValueModel.cs class, place a breakpoint at the first line of the CalculateFutureValue() method.
35. Press F5 to run the project with debugging.
36. In your browser, use the app to calculate a future value. When VS Code goes into break mode, step through the code line by line and inspect the variable values in the Variables window.
37. After you step into the for loop, click Continue to continue execution.
38. Return to the browser and view the future value calculation. Then, close the browser.
39. In VS Code, click the Stop button to stop the app. Then, close the project folder.

# Appendix A

## How to set up Windows for this book

This appendix shows how to install the software that we recommend for developing ASP.NET Core MVC web apps on a Windows system. Then, it shows how to install the source code and create the databases for this book.

As you read this appendix, please remember that most websites are continually updated. As a result, some of the procedures may have changed since this book was published. Nevertheless, these procedures should still be good guides to installing the software. And if there are significant changes to these setup instructions, we will post updates on our website ([www.murach.com](http://www.murach.com)).

How to install Visual Studio.....	728
How to install the source code for this book.....	730
How to create the databases for this book.....	732

## How to install Visual Studio

---

Figure A-1 shows how to install the Community edition of the Visual Studio IDE (Integrated Development Environment). To do that, you download the setup program from the website address shown in this figure. You can also search the Internet for “Visual Studio Community download” and then download the setup program from any website that provides for that. Then, you run the executable file that’s downloaded.

When the setup program runs, it will display a page like the one shown in this figure. Then, you can follow the steps summarized here to install Visual Studio. To start, you select the workload you want to install. The workload you select determines the components that are installed. To develop ASP.NET Core MVC apps, for example, you select the “.NET Core cross-platform development” workload. Then, the required components are listed under the Included heading at the right side of the page.

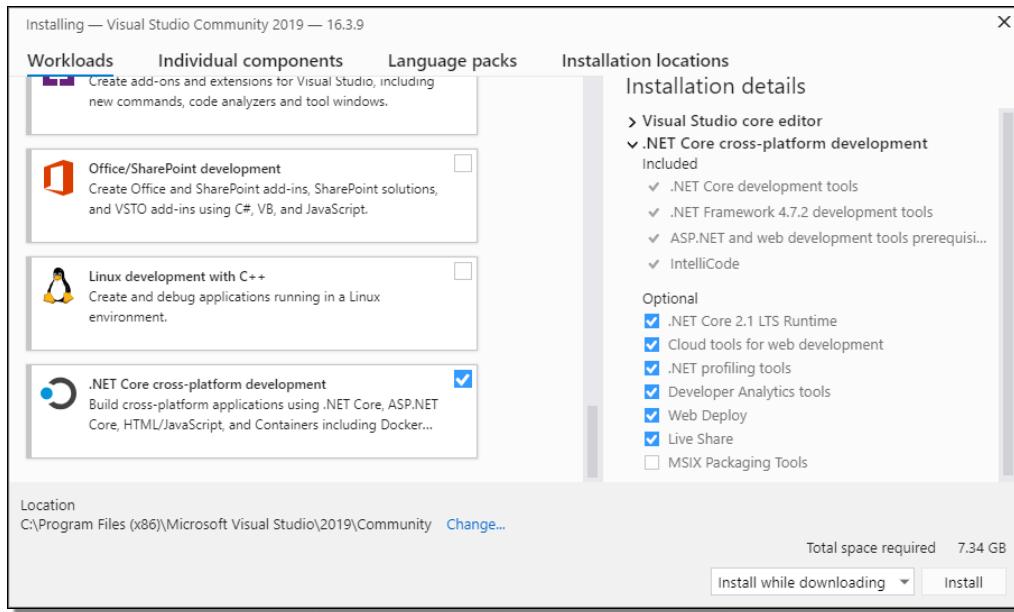
In addition to the required components, a number of optional components are installed by default. In most cases, you’ll just accept the defaults. If you know that you won’t use a component, though, you can deselect it. Then, if you need it in the future, you can run the setup program again to install it.

You can also select and deselect individual components from the Individual components tab. When you first display this tab, the components that will be installed based on the selections on the Workloads tab will be selected. That includes the .NET Core 3.1 SDK, the NuGet Package Manager, and SQL Server Express 2016 LocalDB. Although you’ll want to install the .NET Core 3.1 SDK and the NuGet Package Manager, you can deselect SQL Server Express 2016 LocalDB if it’s already installed on your computer. In addition, you may want to add components. For example, if you want to develop web apps that target .NET Core 2.1, you can select the “Web Development tools plus .NET Core 2.1” component.

Once you’ve selected the components you want to install, you click the Install button to start the installation. Then, Visual Studio displays a window that shows the installation progress. When the installation completes, Visual Studio is started by default and you’re asked to sign in to or create your Microsoft account. You’ll also be asked to set the default environment settings for Visual Studio.

Note that the procedure in this figure installs the current version of Visual Studio, which was Visual Studio 2019 at the time of this printing. However, this book should work equally well with later versions of Visual Studio.

## The main page for installing Visual Studio Community



## The download page for Visual Studio Community

<https://www.visualstudio.com/vs/community/>

### How to install the Visual Studio IDE

1. Download Visual Studio Community from the website address shown above, and then run the downloaded executable file.
2. To install the default features for ASP.NET Core development, select the “.NET Core cross-platform development” workload. Then, deselect any optional features you don’t want to install.
3. Click the Install button to start the installation.
4. When the installation is complete, Visual Studio is started by default and you’ll be asked to sign in and set the default environment settings.

### Description

- Visual Studio Community is a free IDE (Integrated Development Environment) that you can use to create ASP.NET Core MVC apps.
- By default, the Visual Studio 2019 setup program installs the most current version of .NET Core, as well as SQL Server 2016 Express LocalDB. If that’s not what you want, you can display the Individual components tab and select just the components you need.
- For more information about installing and using Visual Studio, you can refer to the Visual Studio website.

Figure A-1 How to install Visual Studio

## How to install the source code for this book

---

Figure A-2 shows how to download and install the source code for this book. This includes the source code for the web apps that are presented in this book. In addition, it includes the source code for the starting points and solutions for the exercises that are presented at the end of each chapter.

When you finish this procedure, the book apps, exercise starts, and exercise solutions should be in the folders shown in this figure. Then, you can review the apps that are presented in this book, and you'll be ready to do the exercises in this book.

## The Murach website

[www.murach.com](http://www.murach.com)

## The folder that contains the Visual Studio projects

C:\murach\aspnet\_core\_mvc

## The subfolders

Folder	Description
book_apps	The web applications that are presented throughout this book.
ex_starts	The starting points for the exercises at the end of each chapter.
ex_solutions	The solutions to the exercises.

## How to download and install the files for this book

1. Go to [www.murach.com](http://www.murach.com), and go to the page for *Murach's ASP.NET Core MVC*.
2. If necessary, scroll down to the FREE Downloads tab. Then, click on it.
3. Click the DOWNLOAD NOW button for the exe file, and respond to the resulting pages and dialogs. This should download an installer file named mvc1\_allfiles.exe.
4. Use File Explorer to find the exe file.
5. Double-click this file and respond to the dialogs that follow. This should install the files for the Visual Studio projects for this book in the folder above.

## How to use a zip file instead of a self-extracting zip file

- Although we recommend using the self-extracting zip file (mvc1\_allfiles.exe) to install the downloadable files as described above, some systems don't allow self-extracting zip files to be downloaded. In that case, you can download a regular zip file (mvc1\_allfiles.zip) from our website. Then, you can extract the files stored in this zip file into the C:\murach folder. If the C:\murach folder doesn't already exist, you can create it.

---

Figure A-2 How to install the source code for this book

## How to create the databases for this book

---

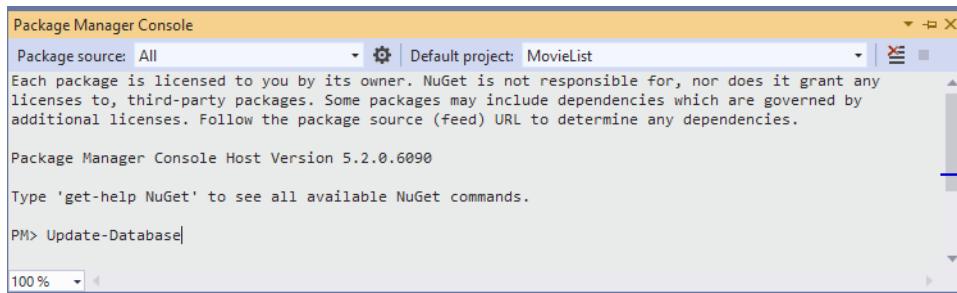
To verify that the software and source code has been installed correctly on your system, you can use Visual Studio to open one of the apps and run it. However, if you run an app and its database hasn't been created yet, you'll get an error message like the one shown at the bottom of figure A-3. This error message indicates that the app can't open the database.

To fix this issue, you can use the procedure shown in this figure to create the database for the app. In this figure, the procedure is for the Movie List app that's described in chapter 4 and the Movies database that it uses. The Movie List app is the first app in this book that uses a database.

Note that if this is the first time you've run one of the apps for this book, a message may be displayed asking if you want to trust the ASP.NET Core SSL certificate, which is used to secure the localhost server.

In addition, this figure lists other apps that use other databases. If you want, you can use the procedure shown in this figure to open these apps now and create their databases too. That way, you won't get an error when you run these apps later. However, if you prefer, you can wait until later to create these databases.

## The Package Manager Console (PMC) window in Visual Studio



### How to create the Movies database and run the Movie List app

1. Start Visual Studio.
2. Select the File→Open→Project/Solution command and use the resulting dialog to open the solution (.sln) file that's in this folder:  
`/murach/aspnet_core_mvc/book_apps/Ch04MovieList`
3. Select the Tools→NuGet Package Manager→Package Manager Console command to display the Package Manager Console.
4. At the command prompt, type “Update-Database” and press Enter. Since the project for this app already contains the database migration files, this should create the database.
5. Press Ctrl+F5 to run the app. If this displays a list of Movies in a browser, the Movies database has been created. (If you get messages about trusting and installing an SSL certificate, you can click Yes. And if a web page is displayed indicating that the connection is not private, you can click the link to proceed.)

### Other apps that you can use to create other databases

App name	Database name
Ch07GuitarShop	GuitarShop
Ch08aNFLTeams	NFLTeams
Ch10bToDoList	TaskList
Ch11Registration	Registration
Ch13Bookstore	Bookstore

### The error message that's displayed if the database hasn't been created

`Cannot open database "Movies" requested by the login. The login failed.`

#### Description

- Many of the apps in this book work with data that's stored in SQL Server Express LocalDB databases. If you run an app and its database hasn't been created yet, you'll get an error message like the one shown above.
- To create a database that's used by one of these book apps, you can open the app and run the “Update-Database” command from the Package Manager Console.

Figure A-3 How to create the databases for this book



# Appendix B

## How to set up macOS for this book

This appendix shows how to install the software that we recommend for developing ASP.NET Core MVC web apps on macOS. Then, it shows how to download the source code for this book, and it describes some of the differences between the Windows and macOS downloads.

As you read this appendix, please remember that most websites are continually updated. As a result, some of the procedures may have changed since this book was published. Nevertheless, these procedures should still be good guides to installing the software. And if there are significant changes to these setup instructions, we will post updates on our website ([www.murach.com](http://www.murach.com)).

How to install Visual Studio.....	736
How to install the source code for this book.....	738
Problems and solutions when using macOS with this book.....	740
How to install and use DB Browser for SQLite .....	742

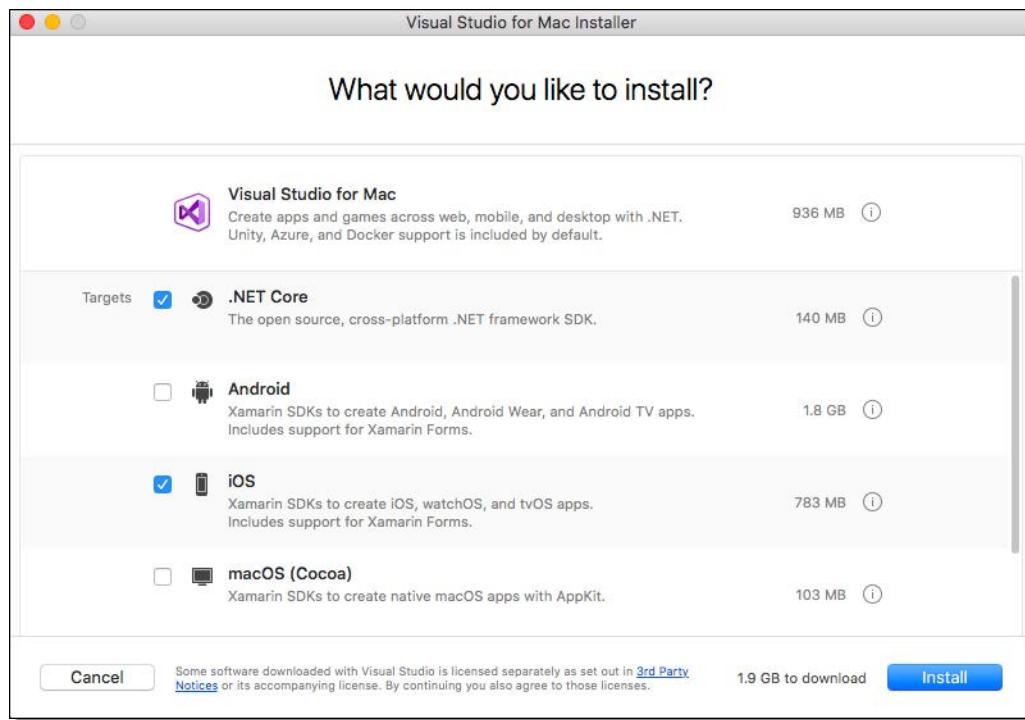
## How to install Visual Studio

---

Figure B-1 shows how to install the Visual Studio IDE (Integrated Development Environment) on macOS. To do that, you download the installer program from the website address shown in this figure. Then, you can run the dmg file that's downloaded. When the dialog shown in this figure is displayed, make sure to select the “.NET Core” option. This will install Visual Studio as well as all of the necessary software for developing ASP.NET Core MVC apps.

Note that the iOS option is selected by default when this installer dialog is displayed. If you leave this option selected, the software you need to develop apps such as those for the iPhone will be installed. Since this software isn't required for this book, you can deselect this option. Then, if you ever want to develop iOS apps, you can run the installer again and select this option.

## The Visual Studio for Mac Installer



## The download page for Visual Studio for Mac

<https://www.visualstudio.microsoft.com/vs/mac/>

## How to install the Visual Studio IDE

1. Find the download page for Visual Studio for Mac by going to the URL above or by searching the Internet for “Visual Studio for Mac download”.
2. Click the appropriate Download button to download the installer program for Visual Studio for Mac to your hard disk. This installer program should be a dmg file.
3. Run the installer program and respond to the resulting dialogs. When the dialog above is displayed, be sure to select the “.NET Core” option. You can also deselect the iOS option if you won’t be developing any of the types of apps listed for this option.

## Description

- Visual Studio is a free IDE (Integrated Development Environment) that you can use to create ASP.NET Core MVC web apps.
- For more information about installing and using Visual Studio, you can refer to the Visual Studio website.

Figure B-1 How to install Visual Studio

## How to install the source code for this book

---

Figure B-2 shows how to download and install the source code for this book. This includes the source code for the web apps that are presented in this book. In addition, it includes the source code for the starting points and solutions for the exercises that are presented at the end of each chapter.

When you finish this procedure, the book apps, exercise starts, and exercise solutions should be in the folders shown in this figure. Then, you can review the apps that are presented in this book, and you'll be ready to do the exercises in this book.

However, you should know that the source code in the zip file for macOS has been modified to work with macOS as described in the next figure. As a result, this source code doesn't match the code that's presented in the book exactly. Still, most of the code matches the code in the book, the most significant differences are described in the next figure, and we've used comments in the code to identify the rest of the differences. So, you shouldn't have much trouble using macOS to work with the downloadable source code for this book.

## The Murach website

[www.murach.com](http://www.murach.com)

### The folder that contains the Visual Studio projects

/murach/aspnet\_core\_mvc

### The subfolders

Folder	Description
book_apps	The web applications that are presented throughout this book.
ex_starts	The starting points for the exercises at the end of each chapter.
ex_solutions	The solutions to the exercises.

## How to download and install the files for this book

1. Go to [www.murach.com](http://www.murach.com), and go to the page for *Murach's ASP.NET Core MVC*.
2. If necessary, scroll down to the FREE Downloads tab. Then, click on it.
3. Click the DOWNLOAD NOW button for the zip file for macOS, and respond to the resulting pages and dialogs. This should download a zip file.
4. Use Finder to locate the zip file on your hard disk, and double-click it to unzip it. This creates the `aspnet_core_mvc` folder and its subfolders.
5. If necessary, use Finder to create the `murach` folder directly on your hard disk. To make that easy to do, you can modify the preferences for Finder so it includes your hard disk in its sidebar.
6. Use Finder to move the `aspnet_core_mvc` folder into the `murach` folder.

### A note about right-clicking

- This book sometimes instructs you to right-click, because that's common in Windows. On macOS, right-clicking is not enabled by default. However, you can enable right-clicking by editing the system preferences for your mouse. Alternately, you can hold down the Ctrl key and click instead of right-clicking.

### A note about the download file for macOS

- Two zip files are available for this book from murach.com. The zip file for Windows contains the source code that matches the code presented throughout this book. The zip file for macOS has been modified to work with macOS. As a result, its source code doesn't match the source code in the book exactly. The most significant differences are described in figure B-3.

---

Figure B-2 How to install the source code for this book

## Problems and solutions when using macOS with this book

---

Figure B-3 presents some problems that you may encounter when using macOS to work with this book. It also presents solutions to those problems. In most cases, these problems have already been fixed in the downloadable source code for this book. Still, you need to understand these problems and their solutions if you're creating web apps from scratch.

For example, if you create the Future Value app from scratch as described in chapter 2, you need to add code that specifies a culture for currency formatting as described in the first problem/solution example. In this example, the code begins by creating a variable named culture for the United States. Then, it supplies this variable as the second argument of the `ToString()` method to use currency formatting that's appropriate for the United States such as \$1,234.56.

Similarly, if you create the Movie List app from scratch as described in chapter 4, you need to use SQLite, not SQL Server Express LocalDB. That's because macOS doesn't support LocalDB. To use SQLite, you can add the appropriate NuGet packages shown in the second problem/solution example. To do that, you can select `Project→Manage NuGet Packages` and use the NuGet Package Manager as described in chapter 4. Then, you can modify the `Startup.cs` file so it uses SQLite, not SQL Server. Finally, you can modify the `appsettings.json` file so it specifies the filename of the SQLite database.

This works because SQLite doesn't use a database server. Instead, it uses a local library to work with a database file. When creating a SQLite database file, it's common to give that file an extension of `.sqlite`. In step 3 of the second example, for instance, the database file is named `Movies.sqlite`.

Unfortunately, Visual Studio for Mac doesn't support the Package Manager Console (PMC) PowerShell commands presented throughout most of this book. As a result, if you need a Powershell command, you'll have to use the equivalent command-line interface (CLI) command. Most of these CLI commands are described in chapter 17. For example, if you want to create the database for the Movie List app from chapter 4, you can use the CLI commands shown in the third problem/solution example. To do that, you can start the Terminal program. Then, you can use it to enter the CLI commands. However, before you execute CLI commands that are specific to a web app, you need to use the `cd` command to change the current directory to the directory that stores the web app as shown in step 3.

Most of the time, SQLite can perform the same tasks as SQL Server Express LocalDB. However, LocalDB provides some features that aren't supported by SQLite. As a result, in some cases, you may need to modify your code so it only uses features that are supported by SQLite. For instance, the fourth problem/solution example shows that the Bookstore app from chapter 13 uses code that can't be translated by SQLite. To fix this, you can modify the code so it uses a different technique get a book at random. If the source code for our apps contains minor fixes like this, it also includes a comment like the one in this example that identifies the change.

**Problem:** macOS doesn't supply a default culture for currency formatting

**Solution:** Add code that specifies a culture for currency formatting

**Example:** The Guitar Shop app from chapter 1

```

@{
    // create culture variable
    var culture = System.Globalization.CultureInfo
        .CreateSpecificCulture("en-US");
}
...
<td>@product.Price.ToString("C", culture)</td><!-- supply culture variable -->

```

**Problem:** macOS doesn't support SQL Server Express LocalDB

**Solution:** Convert the app to use SQLite instead of LocalDB

**Example:** The Movie List app from chapter 4

1. Install the following EF Core NuGet packages for working with SQLite, not SQL Server:

```

EntityFrameworkCore.Sqlite
EntityFrameworkCore.Design

```

2. In the Startup.cs file, use SQLite, not SQL Server. To do that, call the UseSqlite() method, not the UseSqlServer() method, from the options for the DbContext like this:

```
options.UseSqlite(Configuration.GetConnectionString("MovieContext"))
```

3. In the appsettings.json file, code a database connection string that looks like this:

```
"MovieContext": "Filename=Movies.sqlite"
```

**Problem:** Visual Studio for Mac doesn't support PowerShell commands

**Solution:** Use CLI commands instead as described in chapter 17

**Example:** Create the database for the Movie List app from chapter 4

1. Start the Terminal program.

2. If necessary, install the .NET EF tools. To do that, enter this command:

```
dotnet tool install --global dotnet-ef
```

3. Change the current directory to the directory for the project that contains the database:

```
cd /murach/aspnet_core_mvc/book_apps/Ch04MovieList/MovieList
```

4. Create the migration file named Initial. To do that, enter this command:

```
dotnet ef migrations add Initial
```

5. Create the database by running its migration files. To do that, enter this command:

```
dotnet ef database update
```

**Problem:** Some SQL Server features aren't available from SQLite

**Solution:** Modify your code so it doesn't use those features

**Example:** The Bookstore app from chapter 13

```

// get a book at random - updated for SQLite
int bookID = new Random().Next(1, data.Count + 1);
Book random = data.Get(bookID);

```

---

Figure B-3 Problems and solutions when using macOS with this book

## How to install and use DB Browser for SQLite

---

When you use Windows, Visual Studio includes a SQL Server Object Explorer window that allows you to view the SQL Server Express LocalDB databases on your system. When you use macOS with this book, the SQL Server Object Explorer window isn't available from Visual Studio. In addition, you will be using SQLite databases, not SQL Server databases, as described in the previous figure. As a result, to view the SQLite databases on your system, you may want to install DB Browser for SQLite as described by the first procedure in figure B-4.

Once you've installed this program, you can use it to view SQLite databases. To do that, you just open the database file that's in the app's root folder. In this figure, for example, the second procedure shows how to open the Movies.sqlite file for the database of the Movie List app presented in chapter 4. Then, it shows how to view the data in the Movies table. This is often helpful when you need to make sure that a table exists and contains the correct data.

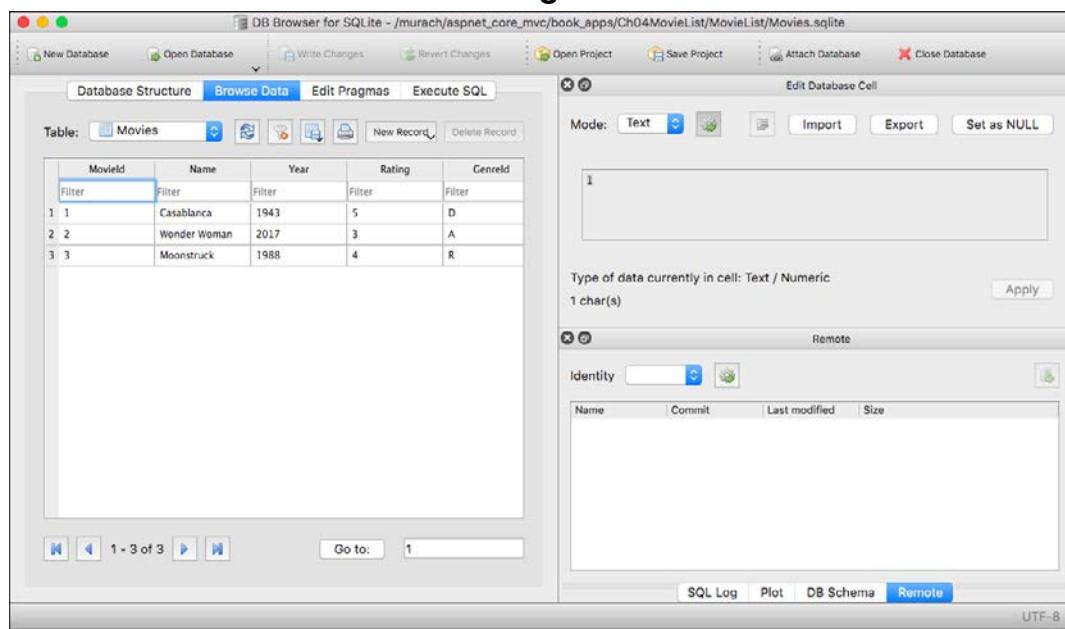
## The download page for DB Browser for SQLite

<http://sqlitebrowser.org/>

## How to install DB Browser for SQLite

1. Find the download page for DB Browser for SQLite. The easiest way to do that is to search the Internet for “DB Browser for SQLite download”.
2. Follow the instructions on that web page to download the installer file. This installer file should be a dmg file.
3. Find the installer file on your system and run it.
4. Accept the default options.

## DB Browser for SQLite after viewing the Movie table



## How to view the Movies database from chapter 4

1. Start DB Browser for SQLite.
2. Click the Open Database button. Then, use the resulting dialog to select the SQLite database file that's located here:  
`\murach\aspnet_core_mvc\Ch04MovieList\MovieList\Movies.sqlite`
3. In the main panel, click the Browse Data tab.
4. Use the drop-down list at the top of the main panel to select the Movies table. This should display all data that's stored in the Movies table.

---

Figure B-4 How to install and use DB Browser for SQLite



# Index

\* symbol (@addTagHelper), 604-605  
 ?. operator, 284-285  
 ?: operator, 232-233, 284-285  
 @addTagHelper directive, 604-605  
 @inject directive, 568-569  
 @model directive, 28-29, 58-59, 252-253  
 @Model property, 28-29, 58-59, 252-253  
 \_Layout.cshtml file, 238-239  
 \_ViewImports.cshtml file, 56-57, 234-235, 238-239  
     @addTagHelper, 604-605  
     importing Models namespace, 252-253  
 \_ViewStart.cshtml file, 234-235, 238-239  
     that sets the default layout, 262-263  
<a> element, 240-241, 246-247  
<option> element, 256-257  
<select> element, 256-257  
<table> element, 260-261

## A

---

AAA pattern, 574-575  
 Absolute URL, 246-247  
 Access, restricting, 656-657, 692-693  
 Account controller, 666-667  
 Action, 26-27, 44-45  
     coding, 202-203  
 Action method, 44-45  
     accessing arguments in URL segments, 208-209  
     for changing a password, 696-697  
     for displaying users and roles, 684-685  
     for logging in a user, 678-679  
     for registering a user, 672-673  
     restricting access, 656-657  
     with dependency injection, 566-567  
     with strongly-typed view, 60-61  
 Action result, 26-27  
 action token (Route attribute), 210-211  
 ActionResult class, 280-281  
     subtypes for redirection, 302-303  
 ActionResult object, 282-283  
 active class (Bootstrap), 114-117, 119  
 Add() method (DbSet), 146-147, 476-477  
 Add-Migration command (PowerShell), 142-143,  
     160-161, 446-449  
     for adding Identity tables, 660-661

AddControllersWithViews() method (Startup.cs), 198-199  
     for enabling TempData, 307  
 AddDbContext() method (Startup.cs), 140-141  
 AddHttpContextAccessor() method (Startup.cs), 561-562  
 AdditionalFields property (RemoteAttribute), 422-423  
 AddMemoryCache() method (Startup.cs), 324-325  
 AddModelError() method (ModelStateDictionary), 406-407  
 AddMvc() method (Startup.cs), 198-199  
 AddScoped() method (IServiceCollection), 561  
 AddSession() method (Startup.cs), 324-325  
 AddSingleton() method (IServiceCollection), 561  
 AddToRoleAsync() method (UserManager), 680-681  
 AddTransient() method (IServiceCollection), 560-561  
 AddValidation() method (IClientModelValidator), 418-419  
 Admin folder (Bookstore website), 502-503  
 Admin page (Bookstore website), 498-499, 546-547  
 Alert (Bootstrap), 116-117  
 alert class (Bootstrap), 116-117  
 alert-context class (Bootstrap), 116-117  
 alert-dismissible class (Bootstrap), 116-117  
 alert-link class (Bootstrap), 116-117  
 AllAttributes property (TagHelperContext), 620-621  
 AllowAnonymous attribute, 656-657  
 Anonymous type, 474-475  
 Any() method (LINQ), 526-527  
 Append() method (Cookies), 348-349  
 AppendHtml() method (TagHelperContent), 616-617  
 Application server, 8-9  
 ApplyConfiguration() method (ModelBuilder), 444-445  
 appsettings.json file (connection string), 140-141  
 Area, 218-221  
     associating with a controller, 220-221  
     setting up, 218-219  
 Area attribute, 220-221  
 area token, 220-221  
 Areas folder (Bookstore website), 502-503  
 aria- attributes  
     for button dropdowns, 112-113  
     for navbars, 120-121  
 aria-label attribute, 116-117  
     for button groups, 108-109

Arrange/Act/Assert (AAA) pattern, 574-575  
 Array, posting, 372-373  
 AsNoTracking() method (DbSet<T>), 472-473  
 ASP.NET Core Identity, 654-655, 658-667  
 ASP.NET Core MVC, 12-15  
 ASP.NET MVC, 12-15  
 ASP.NET Web Forms, 12-15  
 asp-action tag helper, 58-59, 241, 604-605  
 asp-area tag helper, 246-247, 605  
 asp-controller tag helper, 58-59, 240-241, 605  
 asp-for tag helper, 58-59, 254-255, 604-605  
 asp-fragment tag helper, 246-247  
 asp-host tag helper, 246-247  
 asp-items tag helper, 256-257  
 asp-protocol tag helper, 246-247  
 asp-route tag helper, 240-241  
 asp-route-id tag helper, 148-149  
 asp-validation-for tag helper, 408-409, 605  
 asp-validation-summary tag helper, 74-75, 408-409, 605  
 AspNetRoles table, 660-661  
 AspNetUsers table, 660-661  
 Assert class, 574-575  
 Asynchronous method, 666-667  
 Attribute routing, 210-215  
 Attributes property  
     HtmlTargetElement, 614-615  
     TagBuilder, 616-617  
     TagHelperOutput, 608-609  
 Authentication, 650-657  
     claims-based, 654-655  
 Authentication cookie, 650-651  
 Authentication middleware, 16-17  
 Author Catalog page (Bookstore website), 498-499  
 Author/List view (Bookstore website), 518-519  
 AuthorController class (Bookstore website), 516-517  
 Authorize attribute, 656-657  
 AuthorListViewModel class (Bookstore website), 516-517  
 Autos window, 188-189  
 await operator, 672-673

**B**


---

Badge (Bootstrap), 110-111  
     Bookstore navbar, 496-497  
 badge class (Bootstrap), 110-111  
 badge-primary class (Bootstrap), 110-111  
 Benefits of Identity, 654-655  
 Bind a model to a view, 58-59

Bind an HTML element to a property, 58-59  
 Bind attribute, 376-377  
 BindNever attribute, 376-377  
 Book Catalog page (Bookstore website), 496-497  
 Book search (Bookstore website), 546-553  
 Book/List view (Bookstore website), 530-531  
 BookController class (Bookstore website), 528-529  
     with Search() action methods, 550-551  
 BookListViewModel class (Bookstore website), 528-529  
 BookQueryOptions class (Bookstore website), 526-527  
 BooksGridBuilder class (Bookstore website), 524-525  
 Bookstore website, 496-553  
     database classes, 460-465  
     classes for testing, 590-599  
     with custom tag helpers, 638-645  
     with partial views, 638-645  
     with view components, 638-645  
 Bookstore.Tests project, 588-589  
 BookstoreUnitOfWork class (Bookstore website), 526-527  
 Bootstrap, 28-29, 82-123  
     adding, 84-85  
     grid system, 90-93  
 Bootstrap components, 108-117  
 Bootstrap CSS classes  
     for alerts, 116-117  
     for breadcrumbs, 116-117  
     for button dropdowns, 112-113  
     for button groups, 108-109  
     for buttons, 96-97  
     for context, 106-107  
     for forms, 94-95  
     for images, 96-97  
     for jumbotrons, 96-97  
     for list groups, 114-115  
     for margins, 98-99  
     for navbars, 120-123  
     for navs, 118-119  
     for padding, 98-99  
     for tables, 102-103  
     for text, 104-105  
 bootstrap.bundle library, 112  
 Bound property (tag helper), 620-621  
 Boxed layout, 90  
 breadcrumb class (Bootstrap), 117  
 breadcrumb-item class (Bootstrap), 117  
 Breadcrumbs (Bootstrap), 116-117

Break All command (debugging), 186-187  
Break mode  
    Visual Studio, 184-187  
    Visual Studio Code, 720-723  
Breakpoint  
    Visual Studio, 184-185  
    Visual Studio Code, 720-721  
Breakpoints window, 184-185  
Browser, changing from Visual Studio, 50-51, 177  
Browser developer tools, 178-179  
Browser incompatibilities, 176-177  
btn class (Bootstrap), 96-97  
btn-group class (Bootstrap), 108-109  
btn-outline-primary class (Bootstrap), 97  
btn-outline-secondary class (Bootstrap), 96-97  
btn-primary class (Bootstrap), 96-97  
btn-secondary class (Bootstrap), 97  
btn-toolbar class (Bootstrap), 108-109  
Button classes (Bootstrap), 96-97  
Button dropdown (Bootstrap), 112-113  
Button group (Bootstrap), 108-109

## C

---

Cart class (Bookstore website), 538-541  
Cart page (Bookstore website), 532-545  
Cart/Index view (Bookstore website), 544-545  
CartController class (Bookstore website), 542-543  
CartItem class (Bookstore website), 534-535  
CartViewModel class (Bookstore website), 536-537  
Cascading delete, 458-459  
Change tracking, disabling, 472-473  
ChangePasswordAsync() method (UserManager), 696-697  
Claims-based authentication, 654-655  
Class  
    adding, 54-55  
    adding to a unit test, 572-573  
    generic query options, 484-485  
    generic repository, 486-487  
    mapping a dependency, 560-561  
Class-level validation, 416-417  
Classes for testing Bookstore app, 590-599  
CLI commands  
    with macOS, 740-741  
    with VS Code, 710-715  
CLI tools  
    for LibMan, 718-719  
    for .NET EF Core, 710-711  
Client, 4-5  
Client-side libraries  
    adding, 84-85  
    attaching, 88-89  
    enabling, 88-89  
    managing, 86-87  
    working with in VS Code, 718-719  
Client-side validation  
    adding to custom data attributes, 418-419  
    custom, 418-423  
    enabling, 410-411  
close class (Bootstrap), 116-117  
Cloud, 5  
Code block (Razor), 228-229  
Code editor, 22-23  
Code First development (EF Core), 440-449  
Coding by convention, 24-25  
col class (Bootstrap), 90-91  
collapse class (Bootstrap), 120-121  
Command line, 22-23  
Command-line interface (CLI), 710-711  
Compare attribute, 400-401  
Complex type  
    controlling bound values, 376-377  
    with model binding, 364-365  
Components (Bootstrap), 108-117  
Composite primary key, 456-457  
Concurrency, 478-479  
Concurrency conflict, 478-481  
Concurrency exception, 480-481  
Concurrency token, 478-479  
Conditional operator, 232-233, 284-285  
Configuration classes (Bookstore), 460, 464-465  
Configuration files, 444-445  
Configure by convention  
    entity class, 442-443  
    one-to-many relationship, 452-452  
    one-to-one relationship, 454-455  
Configure() method  
    (IEntityTypeConfiguration<T>), 444-445  
Configure() method (Startup.cs), 30-31, 48-49  
    for adding authentication and authorization, 662-663  
    for configuring routing, 198-199  
    for seeding roles and users, 694-695  
ConfigureServices() method (Startup.cs), 30-31, 48-49, 140-141  
    for adding Identity, 662-663  
    for configuring DI, 560-561  
    for configuring URLs, 166-167  
    for endpoint routing, 198-199

- ConfirmPassword property (*IdentityUser*), 658-659
- Connected scenario, 476-477
- Connection string
  - adding, 140-141
  - storing in `appsettings.json` file, 466-467
- container class (*Bootstrap*), 90-91
- container-fluid class (*Bootstrap*), 90-91
- Contains() method (*TempData* dictionary), 306-307
- Content property (*TagHelperOutput*), 616-617
- Content() method, 202-203
- ContentResult class, 281
- ContentResult object, 202-203, 236-237
- Context classes (*Bootstrap*), 106-107
- Context property (view), 326-327
- control-label class (*Bootstrap*), 94-95
- Controller, 10-11, 26-27
  - adding, 44-45
  - associating with an area, 220-221
  - coding, 202-203
  - for a view component, 632-633
  - restricting access, 656-657
  - setting default, 48-49
  - that returns a view, 236-237
  - that uses an id segment, 204-205
  - with dependency injection, 562-563
- Controller class
  - data validation, 398-399
  - methods that return an *ActionResult*, 282-283
  - retrieving GET and POST data, 360-361
  - working with cookies, 348-349
  - working with session state, 326-327
  - working with *TempData*, 306-309
  - working with *ViewBag*, 286-287
    - working with *ViewData*, 284-285
- controller token (*Route* attribute), 210-213
- Controllers folder (*Bookstore website*), 500-501
- Convention over configuration, 24-25
- Cookie, 322-323, 348-349
  - authentication, 650-651
  - for maintaining state, 322-323
- CookieOptions class, 348-349
- Cookies collection, 348-349
- Count property
  - ModelStateDictionary*, 406-407
  - ViewDataDictionary*, 284-285
- Create, Read, Update, Delete (CRUD) operations, 486-487
- CreateAsync() method
  - RoleManager*, 680-681
  - UserManager*, 672-673
- CRUD operations, 486-487
- CSS
  - adding style sheet, 64-65
  - formatting validation messages, 404-405
- Currency formatting with macOS, 740-741
- Custom route, 206-209
  - for Author Catalog (*Bookstore website*), 510-511
  - for Book Catalog (*Bookstore website*), 520-521
- Custom tag helpers, creating, 608-609

## D

---

- danger class (*Bootstrap*), 107
- dark class (*Bootstrap*), 107
- Data
  - accessing using controller, 360-361
  - accessing using model binding, 362-363
  - deleting with EF Core, 146-147, 476-477
  - inserting with EF Core, 146-147, 476-477
  - passing to a custom data attribute, 414-415
  - passing to a partial view, 630-631
  - passing to view components, 632-635
  - querying, 472-473
  - seeding initial, 138-139
  - selecting related, 162-163
  - selecting with LINQ, 144-146
  - updating seed, 158-159
  - updating with EF Core, 146-147, 476-477
  - validating, 398-423
- Data access class, 482-483
- Data attributes
  - adding to generated HTML, 418-419
  - custom, 412-417
  - for configuration, 442-443
  - for validation, 400-401
- Data layer, 482-489
- Data tip, 186-187
- Data transfer object (DTO) class
  - for Author Catalog of *Bookstore website*, 510-511
  - for Book Catalog of *Bookstore website*, 520-521
  - for Book of *Bookstore website*, 536-537
  - for CartItem of *Bookstore website*, 536-537
    - with projections, 474-475
- Data validation, 70-73, 398-423
  - checking, 74-75
  - default, 398-399
- Data validation attributes, 70-71, 400-401
- data-dismiss data attribute (alert), 116-117

data-target attribute, 120-121  
data-toggle attribute, 112-113, 120-121  
Database  
    adding Identity tables, 660-661  
    configuring, 442-443  
    creating from code in EF Core, 440-449  
    creating from Visual Studio on Windows, 732-733  
    creating from VS Code, 710-711  
    creating with migration, 448-449  
    updating with migration, 448-449  
    viewing from DB Browser, 742-743  
    viewing from Visual Studio, 142-143  
Database (DB) context class, 440-441  
database drop command (.NET EF Core), 713  
Database First development (EF Core), 466-471  
Database management system (DBMS), 8-9  
Database server, 8-9  
database update command (.NET EF Core), 713  
DatabaseGenerated attribute, 443  
DataType attribute (password), 668-669  
DB Browser (SQLite), 742-743  
DbContext class (EF Core), 136-137, 440-441  
    Bookstore, 460, 463  
    configuring generated, 468-469  
    generating from a database, 466-467  
    saving changes to database, 146-147  
    seeding roles and users, 694-695  
dbcontext scaffold command (.NET EF Core), 713  
DbContextOptions class (EF Core), 136-137, 140-141  
DbEntityEntry class, 480-481  
DBMS, 8-9  
DbSet class (EF Core), 136-137  
    inserting, updating, and deleting data, 146-147  
DbSet<Entity> class, 144-145  
DbUpdateConcurrencyException, 480-481  
Debug a web app  
    Visual Studio, 176-177, 184-191  
    Visual Studio Code, 720-723  
Debugging commands (Visual Studio), 186-187  
Default Razor layout, 238-239  
Default route (Startup.cs), 198-205  
    adding a slug, 168-169  
    Bookstore website, 496-497  
    configuring, 198-199  
    pattern, 198-201  
Delete behavior, 458-459  
Delete data with EF Core, 146-147  
Delete() method (Cookies), 348-349  
DeleteAsync() method (UserManager), 673  
DeleteBehavior enum, 458-459  
Dependency, 560-561  
    chaining, 562-563  
    mapping, 560-562  
Dependency injection (DI), 560-569  
    enabling, 140-141  
    using with action methods, 566-567  
    using with controllers, 562-563  
    using with HttpContextAccessor, 564-565  
    using with tag helpers, 622-623  
    using with view components, 632-633  
    using with views, 568-569  
Dependency life cycle, 560-561  
Deserialize an object, 328-329  
DeserializeObject() method (JsonConvert), 328-331  
Developer tools (browser), 178-179  
DI, *see dependency injection*  
Dictionary  
    TempData, 306-307  
     ViewData, 284-285  
disable class (Bootstrap), 114-115  
Disconnected scenario, 476-477  
Display attribute, 400-401  
Domain model class, 136-137  
Domain name, 4-5  
Domain property (CookieOptions), 349  
Down() method (migration), 142-143  
Drop-Database command (PowerShell), 446-447  
dropdown class (Bootstrap), 112-113  
dropdown-item class (Bootstrap), 112-113  
dropdown-menu class (Bootstrap), 112-113  
dropdown-toggle class (Bootstrap), 112-113  
DTO class  
    for Author Catalog of Bookstore website, 510-511  
    for Book Catalog of Bookstore website, 520-521  
    for Book of Bookstore website, 536-537  
    for CartItem of Bookstore website, 536-537  
        with projections, 474-475  
Dummy parameter (action methods) 60-61  
Dynamic type (C#), 286-287  
Dynamic web page, 8-9

## E

---

EF Core, 134-143, 440-489  
EF migration commands, 446-449  
Email property (*IdentityUser*), 658-659

EmailConfirmed property (*IdentityUser*), 659  
 Empty template, 40-41  
 EmptyResult class, 281  
 Endpoint routing, 198-199  
 Entity, relating to another entity, 156-157  
 Entity class, 136-137, 440-441  
   adding to, 470-471  
   Bookstore, 460-462  
   generating from a database, 466-467  
   modifying generated, 470-471  
 Entity Framework (EF) Core 134-143, 440-489  
 Entity() method (Fluent API), 443  
 Entity<T> method (ModelBuilder), 138-139  
 EntityState enum, 476-477  
 Entries property (DbUpdateConcurrencyException), 480-481  
 environment tag helper, 604-605  
 Equal() method (Assert), 574-575  
 Error (syntax), 52-53  
 Error List window (Visual Studio), 52-53  
 Error message  
   displaying, 74-75  
   setting for validation attribute, 70-71  
 ErrorCount property (ModelStateDictionary), 406-407  
 ErrorMessage property (data validation attributes), 400-401  
 Errors property (IdentityResult), 672-673  
 Exception, 52-53, 180-181  
 Exception Helper (Visual Studio), 182-183  
 ExecuteSqlRaw() method (DbContext.Database), 478-479  
 Expires property (CookieOptions), 348-349  
 Explorer window (VS Code), 706-707  
 Expression class, 482-483  
 Extension method  
   calling from wrapper class, 332-333  
   CartItem class (Bookstore website), 536-537  
   cookie (Bookstore website), 532-533  
   IQueryable interface, 482-483  
   session (Bookstore website), 504-505  
   string (Bookstore website), 504-505  
   using with tag helpers, 612-613

Fake objects for unit testing  
   creating with Moq, 582-587  
   repository object, 578-579  
   TempData object, 580-581  
 False() method (Assert), 575  
 File  
   adding to VS Code project, 716-717  
   deleting in VS Code, 717  
   renaming in VS Code, 717  
   working with in VS Code, 706-707  
 File() method (Controller), 283  
 Filename, 5  
 FileResult class, 280-281  
 Filtering (Bookstore website), 520-531  
 FilterPrefix class (Bookstore website), 522-523  
 Find() method  
   DbSet<Entity>, 144-146, 472-473  
 FindByIdAsync() method  
   RoleManager, 680-681  
   UserManager, 680-681  
 FindByNameAsync() method  
   RoleManager, 681  
   UserManager, 681  
 FirstOrDefault() method (LINQ), 144-145, 472-473  
 fixed-bottom class (Bootstrap), 122-123  
 fixed-top class (Bootstrap), 122-123  
 Fluent API  
   configuring a database, 442-443  
   configuring relationships, 452-453  
   managing, 444-445  
   many-to-many relationship, 456-457  
   one-to-many relationship, 452-453  
   one-to-one relationship, 454-455  
 Folders  
   Bookstore website, 500-501  
   starting for an app, 234-235  
 Font Awesome, 110-111  
 Font Awesome icons (Bookstore navbar), 496-497  
 for attribute (partial tag helper), 630-631  
 for statement (Razor view), 230-231  
 Foreign key, 450-451  
 Foreign key property (related entity), 156-157  
 ForeignKey attribute, 451  
   one-to-many relationship, 453  
   one-to-one relationship, 455  
 Form  
   accessing data using controller, 360-361  
   accessing data using model binding, 362-363

**F**


---

F12 tools, 178-179  
 Fact attribute, 574-575

Form classes (Bootstrap), 94-95  
 Form property (Request), 360-361  
 form-control class (Bootstrap), 94-95  
 form-group class (Bootstrap), 94-95  
 form-horizontal class (Bootstrap), 94-95  
 form-vertical class (Bootstrap), 94-95  
 Format specifiers for numbers, 248-249  
 Fragment (URL), 246-247  
 Framework (responsive web design), 82-83  
 FromBody attribute, 374-375  
 FromForm attribute, 375  
 FromHeader attribute, 374-375  
 FromQuery attribute, 375  
 FromRoute attribute, 374-375  
 FromServices attribute, 375, 566-567  
 Full width layout, 90  
 Future Value app, 62-63  
     with Bootstrap, 100-101  
     with data validation, 74-75

**G**

GDPR, 324  
 General Data Protection Regulation (GDPR), 324  
 Generic class  
     for query options, 484-485  
     for query options (Bookstore website), 506-507  
     for repository, 486-487  
     for repository (Bookstore website), 508-509  
     mapping a dependency, 560-561  
 GenreController class (Bookstore website), 552-553  
 GET request  
     accessing data using controller, 360-361  
     accessing data using model binding, 362-363  
 GetChildContentAsync() method  
     (TagHelperOutput), 608  
 GetDatabaseValues() method (DbEntityEntry), 480-481  
 GetInt32() (ISession), 326-327  
 GetPathByAction() method (LinkGenerator), 626-627  
 GetRolesAsync() method (UserManager), 681  
 GetString() method (ISession), 327  
 GetValidationState() method  
     (ModelStateDictionary), 406-407  
 Globally unique identifier (GUID), 472-473  
 Glyphicon, 110-111  
 Grid system (Bootstrap), 90-93  
 GridBuilder class (Bookstore website), 514-515

GUID, 472-473  
 Guitar Shop website, 272-275

**H**


---

HasData() method  
     EntityTypeBuilder<T>, 138-139, 158-159  
     Fluent API, 443-445  
 HasForeignKey() method (Fluent API), 451  
     many-to-many relationship, 456-457  
     one-to-one relationship, 455  
 HasKey() method (Fluent API), 443-445  
     many-to-many relationship, 456-457  
 HasMany() method (Fluent API), 451  
     one-to-many relationship, 452  
 HasMaxLength() method (Fluent API), 443  
 HasOne() method (Fluent API), 451  
     many-to-many relationship, 456-457  
     one-to-many relationship, 453  
     one-to-one relationship, 455  
 Hidden field, 322-323  
 Home page (Bookstore website), 496-497  
 HTML, 6-7  
     generated by asp-for tag helpers, 254-255  
     generated by asp-items tag helper, 256-257  
 HTML elements  
     binding to properties, 58-59  
 HTML helpers, 606-607  
 HTML table classes (Bootstrap), 102-103  
 HTML5 data attributes  
     for alerts, 116-117  
     for button dropdowns, 112-113  
     for navbars, 120-121  
 HtmlAttributeName attribute, 618-619  
 HtmlAttributeNotBound attribute, 622-623  
 HtmlTargetElement attribute, 614-615  
 HTTP, 6-7  
     HTTP GET request, 60-61  
     HTTP pipeline, 16-17  
     HTTP POST request, 60-61  
     HTTP request, 6-7  
         redirecting, 302-305  
     HTTP response, 6-7  
     HTTP status codes, 302-303  
 HttpContext property (Controller), 326-327  
 HttpContextAccessor class  
     mapping a dependency, 561-562  
     mocking for unit testing, 586-587  
     with dependency injection, 564-565

HttpGet attribute, 60-61  
     HttpPost attribute, 60-61  
     HTTPS, 6-7  
     Hypertext Markup Language, *see HTML*  
     Hypertext Transfer Protocol (HTTP), 6-7  
     Hypertext Transfer Protocol Secure (HTTPS), 6-7

---

**I**

    IActionResult interface, 44-45, 202-203, 280-283  
         as a return type of an action method, 236-237  
     IClientModelValidator interface, 418-419  
     Icon (Bootstrap), 110-111  
     id attribute (navbar), 120-121  
     id parameter (action method), 204-205  
     id segment (URL), 204, 205  
     IDE, 20-21  
     Idempotent request, 304-305  
     Identity (ASP.NET Core), 654-655  
         adding to database, 660-661  
         adding to DB context, 658-659  
         configuring middleware, 662-663  
     Identity column, 136-137  
     IdentityDbContext class, 655  
     IdentityDbContext<User> class, 658-659  
     IdentityResult class, 655, 672-673  
     IdentityRole class, 655  
     IdentityUser class, 655, 658-659  
     IEntityTypeConfiguration<T> interface, 444-445  
     IEnumerable<T> type, 472-473  
         returning from data layer, 482-483  
     if-else statement (Razor view), 232-233  
     IHttpContextAccessor interface  
         mapping a dependency, 561-562  
         with dependency injection, 564-565  
     IIS, 8-9  
     IIS Express web server, 50-51  
     Image classes (Bootstrap), 96-97  
     img-fluid class (Bootstrap), 96-97  
     Immediate window, 188-189  
     Include() method (LINQ) 162-163, 474-475  
     Index() action method, 60-61  
     Individual user account authentication, 650-653  
     info class (Bootstrap), 107  
     Inject dependencies, 560-561  
     inject directive (SignInManager object), 664-665  
     Inline condition expression (Razor), 232-233  
     Inline conditional statement (Razor), 232-233  
     Inline expression (Razor), 228-229  
     Inline loop (Razor), 230-231

    InlineData attribute, 574-575  
     InnerHTML property (TagBuilder), 616-617  
     input-validation-error CSS class, 404-405  
     Insert data with EF Core, 146-147  
     Integrated development environment (IDE), 20-21  
     Interface  
         generating for a class, 568  
         implementing in Visual Studio, 578-579  
     Internal Server Error page, 180-181  
     Internet, 4-5  
     Internet Information Services (IIS), 8-9  
     Internet service provider (ISP), 4-5  
     Intranet, 4-5  
     InverseProperty attribute, 451  
         one-to-many relationship, 453  
     Invoke() method (ViewComponent), 632-633  
         with parameters, 634-635  
     InvokeAsync() method (ViewComponent), 632  
     IQueryable object  
         for storing a query, 144-145  
         returning from data layer, 482-483  
     IQueryable<T> type, 472-473  
         return from LINQ, 144-145  
      IRepository interface, 486-487  
     Is<T>() method (It), 583  
     IsAny<T>() method (It), 583  
     IServiceCollection interface, 560-561  
     ISession interface, 326-327  
         extending to work with JSON, 330-332  
     IsInRoleAsync() method (UserManager), 680-681  
     IsNull() method (Assert), 575  
     ISP, 4-5  
     IsRequired() method (Fluent API), 443-445  
     IsRowVersion() method (Fluent API), 478-479  
     IsSignedIn() method (SignInManager), 664-665  
     IsType() method (Assert), 574-575  
     IsValid property (ModelStateDictionary), 406-407  
     IsValid() method (ValidationAttribute), 412-413  
     It class, 582-583  
     ITempDataDictionary interface, implementing for  
         unit testing, 580-581  
     IValidatableObject interface, 416-417

## J

---

    JavaScript Object Notation (JSON), 328-329  
     Join entity, 456-457  
     Join table, 456  
     jQuery library, 84-85  
         for client-side validation, 410-411

jQuery unobtrusive validation library  
 adding custom validation methods, 420-421  
 for client-side validation, 410-411

jQuery validation library  
 adding custom validation methods, 420-421  
 for client-side validation, 410-411

JSON, 328-329

Json() method (Controller), 283  
 with remote validation, 422-423

JsonConvert class, 328-329

JsonIgnore attribute, 329

JsonResult class, 280-281  
 with remote validation, 422-423

jumbotron class (Bootstrap), 96-97

**K**


---

Kebab case, 216-217

Keep() method (TempDataDictionary), 308-309

Kestrel server, 8-9  
 using with Visual Studio, 50-51  
 using with VS Code, 708-709

Key attribute, 442-443  
 one-to-one relationship, 455

Key/value pair  
 cookie, 322-323, 348-349  
 ModelState property, 406-407  
 session state, 322-323  
 TempData, 306-307  
 ViewData, 284-285

Keys property  
 ModelStateDictionary, 406-407  
 ViewDataDictionary, 285

KeyValuePair object, 284-285

**L**


---

LAN, 4-5

Language-Integrated Query (LINQ), 144-147

Last in wins concurrency, 478

Layout (Bookstore website), 496-497

Layout property, 262-263  
 \_ViewStart.cshtml, 238-239  
 for nesting layouts, 264-265

LibMan  
 using with Visual Studio, 84-87  
 using with Visual Studio Code, 718-719

libman.json file, 84-85  
 creating in VS code 718-719

Library Manager, *see LibMan*

Life cycles (dependency), 560-561

light class (Bootstrap), 107

LinkGenerator class, 626-627

Linking entity, 456-457

Linking table, 456

LINQ, 144-147

LINQ expression, 482-483

LINQ methods, 144-145

LINQ to Entities, 472-475

List group (Bootstrap), 114-115

List of items (<select> element), 256-257

list-group class (Bootstrap), 114-115

list-group-item class (Bootstrap), 114-115

Local area network (LAN), 4-5

Locals window, 186-189

Log in, 674-679

Log In/Out button, adding, 664-665

Logging setting (appsettings.json file), 146-147

Lookup data, 138

Loosely coupled controller, 562-563

**M**


---

m class (Bootstrap), 99

macOS issues with this book, 740-741

Many-to-many relationship, 450-451  
 configuring, 456-457

MapAreaControllerRoute() method, 218-219

MapControllerRoute() method, 198-199

MapControllers() method, 210-211

MapDefaultControllerRoute() method, 198-199

Margin classes (Bootstrap), 98-99

Mass assignment attack, 376

MaxAge property (CookieOptions), 349

mb class (Bootstrap), 99

Media query, 82-83

Meta tag, 88-89

Metadata property (ModelExpression), 620-621

Middleware, 16-17

Middleware configuration, 30-31  
 for HTTP request pipeline, 48-49  
 for Identity, 662-663

Migration commands (EF), 446-449

Migration file, 446-447  
 creating, 448-449  
 for creating a database, 142-143  
 for updating a database, 160-161  
 reverting, 448-449

migrations add command (.NET EF Core), 713

Migrations folder, 142-143, 446-448

migrations remove command (.NET EF Core), 713  
 ml class (Bootstrap), 99  
 Mock  
   a repository object, 584-585  
   a TempData object, 584-585  
   an HttpContextAccessor object, 586-587  
 Mock<T> class, 582-583  
 Model, 10-11, 26-27  
   adding, 54-55  
   binding to a view, 58-59  
   displaying in a table, 258-261  
   for a partial view, 630-631  
   passing to a view, 250-251  
 model attribute (partial tag helper), 630-631  
 Model binding, 360-377  
   controlling bound properties, 376-377  
   controlling source of values, 374-375  
   default validation, 398-399  
   retrieving GET and POST data, 362-363  
   with complex types, 364-365  
   with submit buttons, 368-371  
 Model class, 136-137  
   importing, 56-57  
 Model property  
   binding to HTML elements, 254-255  
   displaying in a view, 252-253  
   working with in a tag helper, 620-621  
 Model-level validation messages, 408-409  
 Model-View-Controller (MVC) pattern, 10-11  
 ModelBuilder object, 138-139  
 ModelExpression class, 620-621  
 ModelMetadataType attribute, 470-471  
 Models folder (Bookstore website), 500-501  
 ModelState property (Controller), 74-75, 398-399,  
   406-407  
 ModelStateDictionary class, 406-407  
 ModelValidationState enum, 406-407  
 Moq, 582-587  
   adding to test project, 582-583  
 Movie List app  
   folders and files, 132-133  
   Home controller, 148-149  
   Home/Index view, 148-149  
   Home/Index view with genre data, 162-163  
   Movie controller, 150-151  
   Movie controller with genre data, 164-165  
   Movie/Delete view, 154-155  
   Movie/Edit view, 152-153  
   Movie/Edit view with genre drop-down list,  
     164-165  
   user interface, 130-131

MovieContext class  
   for seeding data, 138-139  
   for seeding related data, 158-159  
 mr class (Bootstrap), 98-99  
 MSTest, 570-571  
 mt class (Bootstrap), 98-99  
 MVC folders, 42-43  
 MVC pattern, 10-11  
 MVC template, 40-41  
 MVC web app, 48-49

## N

---

name attribute (partial tag helper), 631  
 Name property (ModelExpression), 620-621  
 Naming conventions, 24-25  
 Nav (Bootstrap), 118-119  
 nav class (Bootstrap), 119  
 nav-item class (Bootstrap), 118-119  
 nav-link class (Bootstrap), 118-119  
 nav-pills class (Bootstrap), 119  
 nav-tabs class (Bootstrap), 119  
 Navbar (Bootstrap), 120-123  
   Bookstore website, 496-497  
 navbar class (Bootstrap), 120-121  
 navbar-brand class (Bootstrap), 120-121  
 navbar-collapse class (Bootstrap), 120-121  
 navbar-dark class (Bootstrap), 120-121  
 navbar-expand-md class (Bootstrap), 120-121  
 navbar-nav class (Bootstrap), 120-121  
 navbar-right class (Bootstrap), 120-121  
 navbar-toggler class (Bootstrap), 120-121  
 Navigation bars (Bootstrap), 118-123  
 Navigation property, 452-453  
 .NET, 14-15  
 .NET Core, 14-15  
 .NET EF Core commands, 712-713  
 .NET Framework, 14-15  
 Network, 4-5  
 NewtonsoftJson NuGet package, 328-329  
 NFL Teams 1.0 app, 288-297  
 NFL Teams 2.0 app, 310-317  
 NFL Teams 3.0 app, 334-347  
 NFL Teams 4.0 app, 350-355  
 NFL Teams app with model binding, 366-367  
 Non-essential cookie, 324  
 Non-idempotent request, 304-305  
 NonAction attribute, 212-213  
 NotEqual() method (Assert), 575  
 NotMapped attribute, 443

NuGet package  
adding to VS Code project, 714-715  
for EF Core, 134-135  
Identity with EF Core, 658-659  
installing, 134-135  
NuGet Package Manager, 134-135  
NUnit, 570-571  
Number, formatting in a view, 248-249

## O

---

Object property (`Mock<T>`), 582-583  
Object-relational mapping (ORM), 134-135, 440  
ObjectInstance property (`ValidationContext`), 416-417  
offset class (`Bootstrap`), 90-91  
`OnConfiguring()` method (`DbContext`), 440-441  
cleaning for generated DB context, 468-469  
`onDelete()` method (Fluent API), 451, 458-459  
One-to-many relationship, 450-451  
configuring, 452-453  
One-to-one relationship, 450-451  
configuring, 454-455  
`OnModelCreating()` method (`DbContext`), 138-139, 440-441  
applying configuration file, 444-445  
for Identity, 658-659  
Open Web Interface for .NET (OWIN), 654-655  
Optimistic concurrency, 478-479  
`OrderBy()` method (LINQ), 144-145, 472-473  
`OrderByDescending()` method (LINQ), 472-473  
ORM, 134-135, 440  
Over posting attack, 376  
OWIN, 654-655

## P

---

`p` class (`Bootstrap`), 98-99  
Package Manager Console (PMC), 142-143, 446-447  
for creating databases, 733  
Padding classes (`Bootstrap`), 98-99  
Paging  
for Author Catalog of Bookstore website, 510-519  
for Book Catalog of Bookstore website, 520-531  
`ParentTag` property (`HtmlTargetElement`), 614-615  
Partial class, 470-471  
partial tag helper, 605, 628-629  
  attributes, 630-631

Partial view, 628-631  
  creating and using, 628-629  
  for a view component, 632-633  
  passing data to, 630-631  
Password, changing, 696-697  
Password property (`IdentityUser`), 658-659  
Password requirements, 662-663  
PasswordOptions class, 663  
`PasswordSignInAsync()` method (`SignInManager`), 678-679  
Path (directory), 4-5  
Path property (`CookieOptions`), 349  
Pattern (routing)  
  default, 198-201  
  multiple routes, 208-209  
  with static content, 206-207  
`pb` class (`Bootstrap`), 99  
`Peek()` method (`TempDataDictionary`), 308-309  
Persistent cookie, 348-349  
  for authentication, 650-651  
Phone Number property (`IdentityUser`), 659  
PhoneNumberConfirmed property (`IdentityUser`), 659  
Pipeline, 16-17  
`pl` class (`Bootstrap`), 99  
PMC, *see Package Manager Console*  
Popper.js library, 84-85, 112-113  
POST request  
  accessing data using controller, 360-361  
  accessing data using model binding, 362-363  
  preventing resubmission, 304-305  
Post-Redirect-Get pattern, *see PRG pattern*  
PostElement property (`TagHelperOutput`), 616-617  
PowerShell commands (EF), 142-143, 446-447  
`pr` class (`Bootstrap`), 99  
PreElement property (`TagHelperOutput`), 616-617  
Preview mode (VS Code), 706-707  
PRG pattern, 304-305  
  with  `TempData`, 306-307  
primary class (`Bootstrap`), 107  
Primary key, 136-137, 450-451  
`Process()` method (`TagHelper`), 608-609  
`ProcessAsync()` method (`TagHelper`), 608  
Program.cs file, 234-235  
Project (VS Code)  
  creating, 714-715  
  debugging, 720-723  
Project folder (VS Code), 704-706  
Projection, 474-475  
Property (tag helper), 618-619

Property() method (Fluent API), 443-445  
 Property-level validation, 400-401  
   on the client, 410-411  
   with class-level validation, 416-417  
 Property-level validation messages, 408-409  
 Protocol, 4-5  
 pt class (Bootstrap), 99

**Q**

---

Query data, 472-473  
 Query expression, 144-145  
 Query options class (generic), 484-485  
   for Bookstore website, 506-507  
 Query property (Request), 360-361  
 Query string  
   accessing data using controller, 360-361  
   accessing data using model binding, 362-363  
   for maintaining state, 322-323

**R**

---

RAD, 12-13  
 Range attribute, 70-71, 400-401  
 Rapid Application Development (RAD), 12-13  
 Razor  
   code, 28-29, 228-229  
   code block, 46-47  
   inline conditional statement, 232-233  
   inline expression, 228-229  
   inline loop, 230-231  
 Razor expression, 46-47  
 Razor layout, 262-271  
   adding, 66-69  
   applying, 262-263  
   creating, 262-263  
   creating default, 238-239  
   nesting, 264-267  
   view context, 268-269  
 Razor syntax, 228-233  
 Razor view, 24-25  
   adding, 46-47, 66-69  
 Razor view engine, 46-47  
 Razor view imports page, 56-57  
 Razor view start, 66-69  
 Redirect() method (Controller), 283  
 Redirection, 302-305  
 RedirectResult class, 281  
 RedirectToActionResult() method (Controller), 282-283,  
   302-303

RedirectToActionResult class, 280-281  
 Reference, adding to a unit test project, 572-573  
 Register a user, 668-669  
 Register link, 664-665  
 Registration app, 424-435  
 RegularExpression attribute, 400-401  
 Related entities, including in a query, 474-475  
 Relationship  
   between entities, 450-459  
   configuring for deletes, 458-459  
   fully defining, 452-453  
 Relative URL, 246-247  
 rem unit, 92-93  
 Remote attribute, 422-423  
 Remote validation, 422-423  
 Remove() method  
   ISession, 327  
   DbSet, 147, 476-477  
 Remove-Migration command (PowerShell), 446-449  
 RemoveFromRoleAsync() method (UserManager), 680-681  
 Render a web page, 6-7  
 RenderBody() method (Razor layout), 238-239  
 RenderSection() method (Razor layout), 270-271  
 Repository, 486-487  
 Repository class (generic), 486-487  
   for Bookstore website, 508-509  
 Repository object  
   creating fake with code, 578-579  
   creating fake with Moq, 584-585  
 Repository pattern, 486-489  
 Request property (Controller), 348-349, 360-361  
 Required attribute, 70-71, 400-401  
   for configuration, 442-443  
 RequireDigit property (PasswordOptions), 663  
 RequiredLength property (PasswordOptions), 663  
 RequireLowercase property (PasswordOptions), 663  
 RequireNonAlphanumeric property (PasswordOptions), 663  
 RequireUppercase property (PasswordOptions), 663  
 Response property (Controller), 348-349  
 Responsive web design, 82-89  
 Restart command (debugging), 187  
 Returns() method (Mock<T>), 582-583  
 Right-clicking on macOS, 739  
 Role, 654-655  
 role attribute (button group), 108-109  
 RoleManager class, 655, 680-681

RoleManager<T> object, injecting, 666-667  
 Roles, 680-695  
   seeding, 694-695  
 Roles property (RoleManager), 680-681  
 Root directory, 24-25  
 Root folder, 24-25  
 Round trip, 8-9  
 rounded class (Bootstrap), 96-97  
 Route  
   custom, 206-209  
   default, 198-205  
   for maintaining state, 322-323  
   multiple routes, 208-209  
   with static content, 206-207  
 Route attribute, 210-215  
 Route segment  
   accessing data using controller, 360-361  
   accessing data using model binding, 362-363  
 RouteData property (Controller), 360-361  
 RouteDictionary class  
   with filtering, 522-523  
   with paging and sorting, 512-513  
 Routing  
   changing for a controller, 214-215  
   changing for an action method, 210-211  
 row class (Bootstrap), 90-91  
 Rowversion property, 478-479  
 Run a web app, 176-177

**S**


---

Safe navigation operator, 284-285  
 SameSite property (CookieOptions), 349  
 Save data with EF Core, 146-147  
 SaveChanges() method (DbContext), 146-147, 476-477  
 Scaffold-DbContext command (PowerShell), 446-447, 466-467  
 Scope (tag helper), 614-615  
 Scoped life cycle (dependency), 560-561  
 script tags (data validation libraries), 410-411  
 Script-Migration command (PowerShell), 446-447  
 Search engine optimization (SEO), 216-217  
 SearchData class (Bookstore website), 548-549  
 SearchViewModel class (Bookstore website), 548-549  
 secondary class (Bootstrap), 107  
 Section (view), 270-271  
 Secure property (CookieOptions), 349

Seed data,  
   initial, 138-139  
   roles and users, 694-695  
   updated, 158-159  
 Segments of a URL, 200-201  
   accessing from an action method, 208-209  
 Select data with LINQ, 144-146  
 Select related data with LINQ, 162-163  
 Select() method (LINQ), 474-475  
 SelectList() constructor, 256-257  
 SEO, 216-217  
 Separation of concerns, 10-11  
 Serialize an object, 328-329  
 SerializeObject() method (JsonConvert), 328-331  
 Server-side validation, 70-75, 398-409  
   custom, 412-417  
 Session, changing defaults, 324-325  
 Session cookie, 348-349  
   for authentication, 650-651  
 Session property (HttpContext), 326-327  
 Session state, 322-333  
 Set<T> method (DbContext), 486-487  
 SetAttributes() method (TagHelperAttributeList), 608-609  
 SetContent() method (TagHelperContent), 616-617  
 SetInt32() method (ISession), 326-327  
 SetString() method (ISession), 327  
 Setup() method (Mock<T>), 582-583  
 Shared folder, 234-235  
 SignInAsync() method (SignInManager), 672-673  
 SignInManager class, 655, 672-673, 678-679  
 SignInManager<T> object, injecting, 666-667  
 SignOutAsync() method (SignInManager), 672-673  
 Single() method (LINQ), 480-481  
 SingleOrDefault method (LINQ), 480-481  
 Singleton life cycle (dependency), 560-561  
 Skip() method (LINQ), 472-473  
 Slug, 216-217  
   adding to URL, 168-169  
 Sorting  
   for Author Catalog, 510-519  
   for Book Catalog, 520-531  
 Source code editor, 705  
 Source code for the book apps  
   installing on macOS, 738-739  
   installing on Windows, 730-731  
 SQL Server Express LocalDB, 728-729  
 SQL statements, viewing generated, 146-147

SQLite, using with macOS, 740-741  
 Stack trace, 180-181  
 Standard mode (VS Code), 706-707  
 Start/Continue command (debugging), 187  
 Startup.cs file, 30-31, 48-49, 234-235  
     adding a slug to the default route, 168-169  
     configuring for session state, 324-325  
     configuring the default route, 198-199  
     configuring to seed roles and users, 694-695  
     configuring to use generated DB context, 468-469  
     configuring URLs, 166-167  
     password options, 662-663, 668-669  
     with dependency injection, 140-141  
 State, 18-19, 322-323  
 State of an app, 322-323  
 Stateless protocol, 18-19, 322-323  
 Static content (route), 206-207  
 Static web page, 6-7  
 StatusCodeResult class, 281  
 Step through an app  
     Visual Studio, 186-187  
     Visual Studio Code, 722-723  
 Stop debugging  
     Visual Studio, 187  
     Visual Studio Code, 722  
 StringLength attribute, 400-401  
     for configuration, 442-443  
 Strongly-typed view, 58-59  
     with action methods, 60-61  
 Style sheet, attaching, 88-89  
 Submit button  
     posting a name/value pair, 370-371  
     with model binding, 368-371  
 Succeeded property (IdentityResult), 673  
 success class (Bootstrap), 106-107  
 SuppressOutput() method (TagHelperOutput), 624-625  
 switch statement (Razor view), 232-233  
 Synchronous method, 666-667  
 Syntax error, 52-53

## T

---

table class (Bootstrap), 103  
 Table splitting, 454-455  
 table-bordered class (Bootstrap), 102-103  
 table-hover class (Bootstrap), 102-103  
 table-responsive class (Bootstrap), 102-103  
 table-striped class (Bootstrap), 102-103

Tag helpers, 28-29, 604-627  
     adding properties, 618-619  
     built-in, 604-605  
     compared to HTML helpers, 606-607  
     conditional, 624-625  
     controlling scope, 614-615  
     custom, 608-627  
     enabling, 56-57, 238-239  
     for adding elements, 616-617  
     for non-standard HTML elements, 610-611  
     generating URLs, 626-627  
     generating URLs for links, 240-241, 246-247  
     partial, 628-629  
     registering, 604-605  
     with data validation, 408-409  
     with dependency injection, 622-623  
     with extension methods, 612-613  
     working with bound property, 620-621  
 TagBuilder class, 616-617  
 TagHelper class, 608-609  
 TagHelperAttributeList class, 608-609  
 TagHelperContent class, 616-617, 620-621  
 TagHelperOutput class, 608-611, 616-617, 624-625  
 TagMode enumeration, 610-611  
 TagMode property (TagHelperOutput), 610-611  
 TagName property (TagHelperOutput), 610-611  
 Take() method (LINQ), 472-473  
 TCP/IP, 7  
 TempData, for maintaining state, 322-323  
 TempData object  
     creating fake with code, 580-581  
     creating fake with Moq, 584-585  
 TempData property (Controller), 306-309  
 TempDataDictionary class, 308-309  
 Templates (web app)  
     Visual Studio, 40-41  
     Visual Studio Code, 714-715  
 Terminal window (VS Code), 22-23, 710-711  
 Test a web app, 176-183  
 Test Explorer, 576-577  
     for Bookstore.Tests project, 588-589  
 Text classes (Bootstrap), 104-105  
 text-capitalize class (Bootstrap), 105  
 text-center class (Bootstrap), 105  
 text-left class (Bootstrap), 105  
 text-lowercase class (Bootstrap), 105  
 text-right class (Bootstrap), 105  
 text-uppercase class (Bootstrap), 105  
 ThenInclude() method (LINQ), 474-475

Theory attribute, 574-575  
Third-party authentication services, 650-651  
Thread, 666-667  
Tightly coupled controller, 562-563  
Timestamp attribute, 478-479  
ToDo List app, 378-393  
Tokens (for routes), 210-211  
ToList() method (LINQ), 144-145, 473  
ToTable() method (Fluent API), 443  
Tracepoint, 190-191  
Transient life cycle (dependency), 560-561  
Transmission Control Protocol/Internet Protocol (TCP/IP), 7  
True() method (Assert), 575

## U

---

Uniform Resource Locator (URL), 4-5  
Unit, testing, 571  
Unit of work pattern, 488-489  
Unit test  
    running, 576-577  
    writing, 574-575  
Unit test project, 572-573  
Unit testing, 570-587  
    advantages, 570-571  
    process, 571  
Up() method (migration), 142-143, 160-161  
Update data with EF Core, 146-147  
Update() method (DbSet), 146-147, 476-477  
Update-Database command (PowerShell), 142-143, 160, 446-449, 733  
    for adding Identity tables, 660-661  
UpdateAsync() method  
    RoleManager, 681  
    UserManager, 673  
URL, 4-5  
    absolute, 246-247  
    best practices, 216-217  
    fragment, 246-247  
    generating from tag helpers for links, 240-241, 246-247  
    generating in a tag helper, 626-627  
    relative, 246-247  
    segments, 200-201  
    user-friendly, 166-169  
UseAuthentication() method (IAppBuilder), 662-663  
UseAuthorization() method (IAppBuilder), 662-663  
UseEndpoints() method, 198-199

User  
    adding registration fields, 696-697  
    logging in, 674-679  
    registering, 668-673  
User controller, 684-685, 690-691  
User entity (roles), 682-683  
User entity class, 658-659  
User interface (Bookstore website), 496-499, 534-535  
User-friendly URL, 166-169  
UserManager class, 655, 672-673, 680-681  
UserManager<T> object, injecting, 666-667  
UserName property (IdentityUser), 658-659  
UseRouting() method (Startup.cs), 198-199  
Users, seeding, 694-695  
Users property (UserManager), 680-681  
UseSession() method (Startup.cs), 324-325

## V

---

Validate() method (IValidatableObject), 416-417  
ValidateSummary enum, 408-409  
Validation, 70-73  
    class-level, 416-417  
    client-side, 410-411  
    custom client-side, 418-423  
    custom server-side, 412-417  
    data, 398-423  
    model-level, 408-409  
    property-level, 400-401, 408-409  
    property-level with class-level, 416-417  
    remote, 422-423  
    server-side, 70-75, 398-409  
Validation attributes, 70-71  
Validation messages  
    formatting with CSS, 404-405  
    setting, 406-407  
Validation method, adding to jQuery, 420-421  
Validation rule, 70-71  
Validation state, checking, 406-407  
validation-summary-errors class, 404-405  
validation-summary-valid class, 404-405  
ValidationAttribute class, 412-413  
ValidationContext class, 412-413, 416-417  
ValidationResult class, 412-413  
Values property  
    ModelStateDictionary, 406-407  
    RouteData, 360-361  
    ViewDataDictionary, 285  
var keyword, 472-473  
Variable, monitoring, 188-189

Variables window (VS Code), 722-723  
 View, 10-11  
     displaying model properties, 252-253  
     for logging in, 676-677  
     for registering a user, 670-671  
     for working with users and roles, 686-689  
     partial, 628-631  
     strongly-typed, 58-59  
     that explicitly specifies a layout, 262-263  
     with dependency injection, 568-569  
     working with session state, 326-327  
 View component, 632-637  
     creating and using, 632-633  
     simplifying apps, 636-637  
 View model, 298-301  
     benefits, 300-301  
     creating, 298-299  
     for changing a password, 696-697  
     for logging in, 674-675  
     for registering a user, 668-669  
     for users and roles, 683-683  
 View result, 26-27  
 View() method (Controller), 44-45, 282-283  
     for passing a model to a view, 250-251  
     in an action method, 236-237  
     with a strongly-typed view, 60-61  
 ViewBag property (Controller), 44-45, 286-287  
     for displaying a title, 238-239  
 ViewComponent class, 632-633  
 ViewContext attribute, 622-625  
 ViewContext property, 268-269  
     injecting into a tag helper, 622-623  
 ViewData attribute (partial tag helper), 630-631  
 ViewData dictionary (partial view), 630-631  
 ViewData property (Controller), 238-239, 284-285  
     when to use instead of ViewBag, 286-287  
 ViewDataDictionary class, 284-285  
 Viewport, 88-89  
 ViewResult class, 280-281  
 ViewResult object, 44-45, 202-203  
     returning from an action method, 236-237  
 Views folder, 234-235  
     Bookstore website, 502-503  
     partial view, 628-629  
     view component partial view, 632-633  
     with default layout and tag helpers, 242-245  
 Visual Studio, 20-21  
     creating a database, 142-143  
     debugging an app, 176-177, 184-191  
     installing on macOS, 736-737  
     installing on Windows, 728-729

running an app, 50-51, 176-177  
 starting an app, 38-39  
 testing an app, 176-183  
 Visual Studio Code, 22-23, 704-723  
     creating a database, 710-711  
     debugging a project, 720-723  
     installing, 704-705  
     opening and closing a project folder, 704-706  
     running an app, 708-709  
     starting a new project, 714-719  
     stopping an app, 708-709  
     viewing and editing files, 706-707  
 VS Code, *see Visual Studio Code*

## W

---

w class (Bootstrap), 102-103  
 WAN, 4-5  
 warning class (Bootstrap), 106-107  
 Watch window  
     Visual Studio, 188-189  
     Visual Studio Code, 722-723  
 Web app, 4-5  
     debugging, 176-177, 184-191  
     running, 50-51, 176-177  
     starting, 38-39  
     testing, 176-183  
 Web browser, 4-5  
 Web page, 4-5  
     dynamic, 8-9  
     static, 6-7  
 Web server, 4-5  
 Where() method (LINQ), 144-145, 472-473  
 Wide area network (WAN), 4-5  
 Wildcard symbol (@addTagHelper), 604-605  
 Windows PowerShell, 22-23  
 Windows-based authentication, 650-651  
 WithMany() method (Fluent API), 451  
     many-to-many relationship, 456-457  
     one-to-many relationship, 453  
 WithOne() method (Fluent API), 451  
     one-to-many relationship, 452  
     one-to-one relationship, 455  
 Workload (Visual Studio), 728-729  
 Wrapper class (session state), 332-333  
 wwwroot folder, 234-235

## XYZ

---

xUnit framework, 570-571  
 Zen mode (VS Code), 706-707





# 100% Guarantee

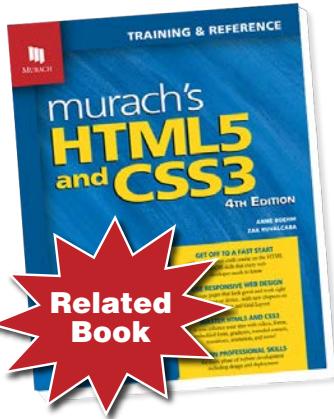
**When you order directly from us, you must be satisfied.** Try our books for 30 days or our eBooks for 14 days. They must work better than any other programming training you've ever used, or you can return them for a prompt refund. No questions asked!



Mike Murach, Publisher



Ben Murach, President



## Have you mastered HTML and CSS?

The more you know about HTML and CSS, the more productive you'll be as a server-side developer. That's why *Murach's HTML5 and CSS3* is a great companion to our ASP.NET book.

## Books for .NET developers

Murach's C# 8.0 ( <i>Coming Spring 2020</i> )	\$59.50
Murach's ASP.NET Core MVC	59.50
Murach's C# 2015	57.50
Murach's ASP.NET 4.6 Web Programming w/ C# 2015	59.50

## Books for database programmers

Murach's SQL Server 2016 for Developers	\$57.50
Murach's Oracle SQL and PL/SQL (2 <sup>nd</sup> Ed.)	54.50
Murach's MySQL (3 <sup>rd</sup> Ed.)	57.50

## Books for Java, C++, and Python programmers

Murach's Java Programming (5 <sup>th</sup> Ed.)	\$59.50
Murach's C++ Programming	59.50
Murach's Python Programming	57.50

## Books for web developers

Murach's HTML5 and CSS3 (4 <sup>th</sup> Ed.)	\$59.50
Murach's JavaScript and jQuery (3 <sup>rd</sup> Ed.)	57.50
Murach's PHP and MySQL (3 <sup>rd</sup> Ed.)	57.50

\*Prices and availability are subject to change. Please visit our website or call for current information.

## We want to hear from you

Do you have any comments, questions, or compliments to pass on to us? It would be great to hear from you! Please share your feedback in whatever way works best.



[www.murach.com](http://www.murach.com)



1-800-221-5528  
(Weekdays, 8 am to 4 pm Pacific Time)



[murachbooks@murach.com](mailto:murachbooks@murach.com)



[twitter.com/MurachBooks](https://twitter.com/MurachBooks)



[facebook.com/murachbooks](https://facebook.com/murachbooks)



[linkedin.com/company/mike-murach-&-associates](https://linkedin.com/company/mike-murach-&-associates)



[instagram.com/murachbooks](https://instagram.com/murachbooks)

## **What software you need for this book**

---

- Visual Studio with the “.NET Core” workload. For Windows, this workload includes
  - .NET Core 3.1
  - ASP.NET Core MVC 3.1
  - C# 8.0
  - A built-in web server called Kestrel
  - A built-in database server called SQL Server Express LocalDB.

The same components are included for macOS except for SQL Server Express LocalDB. To work with databases on macOS, you can use SQLite database files, which don’t require a database server.

- For information about installing Visual Studio with the “.NET Core” workload, please see appendix A (Windows) or appendix B (macOS).

## **The downloadable files for this book**

---

- The source code for all of the applications presented in this book.
- Starting points for the exercises in this book so you can get more practice in less time.
- Solutions for all of the exercises in this book so you can check your work on the exercises.
- For information about downloading and installing these applications, please see appendix A (Windows) or appendix B (macOS).

**[www.murach.com](http://www.murach.com)**