



Building Web Applications with Visual Studio 2017

Using .NET Core and Modern JavaScript
Frameworks

—
Philip Japikse
Kevin Grossnicklaus
Ben Dewey

Apress®

Building Web Applications with Visual Studio 2017

Using .NET Core and Modern
JavaScript Frameworks



Philip Japikse
Kevin Grossnicklaus
Ben Dewey

Apress®

Building Web Applications with Visual Studio 2017

Philip Japikse
West Chester, Ohio
USA

Kevin Grossnicklaus
Ellisville, Missouri
USA

Ben Dewey
Charleston, South Carolina
USA

ISBN-13 (pbk): 978-1-4842-2477-9
DOI 10.1007/978-1-4842-2478-6

ISBN-13 (electronic): 978-1-4842-2478-6

Library of Congress Control Number: 2017947048

Copyright © 2017 by Philip Japikse, Kevin Grossnicklaus and Ben Dewey

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

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

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

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

Cover image designed by Freepik

Managing Director: Welmoed Spahr
Editorial Director: Todd Green
Acquisitions Editor: Todd Green
Development Editor: Laura Berendson
Technical Reviewer: Eric Potter
Coordinating Editor: Jill Balzano
Copy Editor: Kezia Endsley
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit <http://www.apress.com/rights-permissions>.

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

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub via the book's product page, located at www.apress.com/9781484224779. For more detailed information, please visit <http://www.apress.com/source-code>.

Printed on acid-free paper

*To Amy, Coner, Logan, and Skylar. Without your support and patience,
this work would never have happened. Love you guys.*

—Philip Japikse

Contents at a Glance

About the Authors	xix
About the Technical Reviewer	xxi
Acknowledgments	xxiii
Introduction	xxv
■ Part I: Visual Studio 2017 and .NET Core	1
■ Chapter 1: Introducing Entity Framework Core	3
■ Chapter 2: Building the Data Access Layer with Entity Framework Core	49
■ Chapter 3: Building the RESTful Service with ASP.NET Core MVC Services	83
■ Chapter 4: Introducing ASP.NET Core MVC Web Applications	119
■ Chapter 5: Building the SpyStore Application with ASP.NET Core MVC	157
■ Part II: Client-Side Tooling and JavaScript Frameworks	209
■ Chapter 6: JavaScript Application Tools	211
■ Chapter 7: Introduction to TypeScript	241
■ Chapter 8: Angular 2	281
■ Chapter 9: React	329
Index	389

Contents

About the Authors	xix
About the Technical Reviewer	xxi
Acknowledgments	xxiii
Introduction	xxv
■ Part I: Visual Studio 2017 and .NET Core	1
■ Chapter 1: Introducing Entity Framework Core	3
The SpyStore Database	4
Installing Visual Studio 2017 and .NET Core	5
Installing Visual Studio	5
Installing the .NET Core SDKs.....	6
The .NET Core Command Line Interface (CLI).....	8
Creating and Configuring the Solution and Projects	8
Creating the Solution and Projects	8
Changes to the Project Files.....	11
Updating the Target Framework	11
Working with NuGet Packages	13
Manually Restoring Packages	13
Adding the Project References	14
Adding Entity Framework Core.....	14
Adding EF Packages to the SpyStore.DAL Project	14
Installing/Updating Packages Using the SpyStore.DAL.csproj File	16
Adding EF Packages to the SpyStore.Models Project.....	16
Adding Packages to the SpyStore.DAL.Tests Project.....	17

- Building the Foundation 17**
 - Understanding the DbContext Class 17
 - Building the Base Entity Class 22
 - Adding the Category Model Class..... 24
 - Adding the Categories DbSet..... 26
- Migrations 26**
 - Executing EF .NET CLI Commands..... 27
 - Creating the First Migration..... 27
 - Applying the Migration 29
 - Creating Migration SQL Scripts 30
- Understanding CRUD Operations Using Entity Framework..... 31**
 - Creating Records 31
 - Reading Records 31
 - Updating Records 32
 - Deleting Records 33
- Unit Testing EF Core 34**
 - Creating the CategoryTests Class 34
 - Testing EF CRUD Operations 37
- Adding the Core Repository Interface and Base Class 41**
 - Adding the IRepo Interface 41
 - Adding the Base Repository 43
 - Adding the Category Repository 46
- Summary 47**
- Chapter 2: Building the Data Access Layer with Entity Framework Core..... 49**
 - The SpyStore Database 49
 - Navigation Properties and Foreign Keys 50
 - Handling Display Names 51
 - Mixing EF with Stored Procedures and Functions..... 51

Finishing the Model Classes.....	52
Updating the Category Model.....	52
Adding the Product Model.....	52
Adding the Shopping Cart Record Model.....	53
Adding the Order Model.....	54
Adding the Order Detail Model.....	54
Adding the Customer Model.....	55
Updating the StoreContext.....	56
Updating the Database to Match the Model.....	58
Creating the Migration.....	58
Deploying the Migration.....	58
Adding the Stored Procedure and User Defined Function.....	58
Adding a New Migration.....	59
Implementing the Up() Method.....	59
Implementing the Down() Method.....	60
Updating the Database.....	60
Adding the OrderTotal Calculated Field.....	60
Updating the Order Class.....	60
Making OrderTotal a Computed Column.....	60
Adding a New Migration and Update the Database.....	61
Automating the Migrations.....	61
Adding the View Models.....	62
The Product with Category View Model.....	62
The Order Detail with Product Info View Model.....	63
The Order with OrderDetails View Model.....	63
The Cart Record with Product Infor View Model.....	64
Completing the Repositories.....	64
Extending the Interfaces.....	64
Adding/Updating the Repositories.....	67

Initializing the Database with Data.....	75
Creating Sample Data.....	76
Using the Sample Data.....	78
Using the_INITIALIZER in Tests.....	80
Creating NuGet Packages for the Data Access Library.....	81
Setting the NuGet Properties.....	81
Creating the NuGet Packages.....	81
Summary.....	82
■ Chapter 3: Building the RESTful Service with ASP.NET Core MVC Services.....	83
Introducing the MVC Pattern.....	83
The Model.....	83
The View.....	83
The Controller.....	84
Introducing ASP.NET Core MVC Web API.....	84
ASP.NET Core and .NET Core.....	84
Dependency Injection.....	85
Determining the Runtime Environment.....	85
Routing.....	86
Creating the Solution and the Core MVC Project.....	87
Adding the Package Source for the Data Access Layer.....	90
Updating and Adding NuGet Packages.....	90
The ASP.NET Core “Super” Packages.....	91
MVC Projects and Files.....	92
The Program.cs File.....	92
The appsettings.json File(s).....	93
The runtimeconfig.template.json File.....	94
The Startup.cs File.....	94
The Controllers Folder.....	100
The wwwroot Folder.....	100
The web.config File.....	101
The launchsettings.json File.....	101

Controllers and Actions	102
Controllers	102
Actions.....	102
An Example Controller	104
Exception Filters.....	108
Creating the SpyStoreExceptionFilter	108
Adding the Exception Filter for All Actions.....	110
Building the Controllers.....	110
The Category Controller.....	111
The Customer Controller.....	112
The Search Controller	113
The Orders Controller	113
The Product Controller.....	114
The Shopping Cart Controller	115
Using the Combined Solution	118
The Unit Test Solution.....	118
Summary.....	118
■ Chapter 4: Introducing ASP.NET Core MVC Web Applications.....	119
Introducing the “V” in ASP.NET Core MVC	119
Creating the Solution and the Core MVC Project.....	120
Updating and Adding NuGet Packages	122
Routing Revisited	123
The Route Table	123
URL Templates and Default Values	123
MVC Web Applications Projects and Files	124
The Program.cs File.....	124
The appsettings.json File.....	124
The Startup.cs File.....	124
The Controllers Folder	126
The Views Folder	126
The wwwroot Folder.....	126

Controllers, Actions, and Views	129
ViewResults	129
Views	130
Layouts	133
Partial Views	134
Sending Data to Views.....	134
Package Management with Bower	135
Updating and Adding Bower Packages.....	136
Bower Execution.....	136
Bundling and Minification	137
The BundlerMinifier Project.....	137
Configuring Bundling and Minification	137
Visual Studio Integration.....	139
.NET Core CLI Integration.....	141
Creating the Web Service Locator	142
Creating the IWebServiceLocator Interface	142
Creating the WebServiceLocator Class.....	142
Adding the WebServiceLocator Class to the DI Container	143
Creating the WebAPICalls Class to Call the Web API Service.....	143
Creating the IWebApiCalls Interface	143
Creating the Base Class Code.....	144
Creating the WebApiCalls Class.....	148
Adding WebApiCalls Class to the DI Container	151
Adding the Fake Authentication	151
Building the Authentication Helper	151
Adding the AuthHelper Class to the DI Container	152
Creating the Action Filter for the Fake Authentication.....	152
Adding the Action Filter for All Actions	154
Adding the View Models.....	154
Summary.....	156

■ **Chapter 5: Building the SpyStore Application with ASP.NET Core MVC 157**

Tag Helpers 157

- Enabling Tag Helpers 159
- The Form Tag Helper..... 160
- The Anchor Tag Helper..... 160
- The Input Tag Helper..... 160
- The TextArea Tag Helper 161
- The Select Tag Helper..... 161
- The Validation Tag Helpers..... 162
- The Link and Script Tag Helpers 163
- The Image Tag Helper 163
- The Environment Tag Helper..... 163
- Custom Tag Helpers..... 163

Building the Controllers..... 164

- The Products Controller 165
- The Orders Controller 168
- The Shopping Cart Controller 170

Validation..... 176

- Server Side Validation..... 176
- Client Side Validation 180
- Updating the View Models 182

View Components..... 183

- Building the Server Side Code..... 183
- Building the Client Side Code 186
- Invoking View Components..... 186
- Invoking View Components as Custom Tag Helpers 186

Updating and Adding the Views..... 187

- The ViewImports File 187
- The Shared Views 187
- The Cart Views..... 195
- The Orders Views..... 200
- The Products Views 203

Running the Application	205
Using Visual Studio 2017	205
Using the .NET Command Line Interface (CLI).....	206
Using the Combined Solutions.....	206
Summary.....	207
■ Part II: Client-Side Tooling and JavaScript Frameworks	209
■ Chapter 6: JavaScript Application Tools.....	211
What Tools Are We Covering?	211
Node.js	212
Manually Installing Node.js.....	212
Installing Node using the Chocolatey Package Manager.....	213
Setting Up Visual Studio to Use the Latest Version of Node	213
Getting Started with Node.js.....	214
Introduction to NPM.....	215
Saving Project Dependencies.....	217
Executable Packages.....	218
Installing Packages Locally vs. Globally	220
Bower	220
Bower and Visual Studio.....	222
Installing Bower Prerequisites.....	222
Installing Git.....	222
Installing Bower	222
Using Bower	223
Installing Bower Packages	223
Installing Bower Packages Using Visual Studio.....	224
Gulp	227
Installing Gulp.....	228
Copying Files Using Gulp	229
Dependencies in Gulp.....	230
Task Runner Explorer Within Visual Studio	231
Gulp Conclusion.....	232

Module Loaders.....	233
What Is a Module.....	234
SystemJS.....	235
WebPack.....	237
Summary.....	240
■ Chapter 7: Introduction to TypeScript	241
Why TypeScript?	241
TypeScript Basics	242
An Overview of TypeScript Syntax.....	242
Implementing a Basic TypeScript Application.....	253
Setting Up a Sample Project.....	254
Working with TypeScript Files	257
NPM Packages.....	259
Adding TypeScript.....	261
Summary.....	279
■ Chapter 8: Angular 2	281
Creating a New Visual Studio Core Project.....	281
Project Files.....	284
Setting Up the Startup Class.....	284
NPM Install	286
Gulp Setup	287
Typescript Setup.....	287
Main SpyStore App Component Setup.....	288
Creating the Root index.html Page.....	289
Creating the Root App Component.....	291
Creating the App Module	292
Creating the Angular Bootstrap	292

Core Concepts	293
Application Initialization	293
Components.....	294
Services.....	295
Templating	297
Routing	300
Building the SpyStore Angular App	303
Adding Routing	303
Connecting to Services.....	306
Route Parameters.....	311
Search Page	313
Product Details Page	316
Cart Page	320
Checkout.....	326
Summary	328
■ Chapter 9: React	329
Solution Overview	329
Creating a New Visual Studio Core Project.....	330
Project Files.....	332
Setting Up the Startup Class	333
NPM Packages	334
TypeScript Setup	337
Initial Project Folder	338
Webpack.....	342
Introduction to React.....	351
Components.....	351
Application Organization.....	355
Models	356
Services.....	357
Initial Components.....	361

Routing	362
App Component	365
CategoryLinks Component.....	369
Products Component	371
ProductDetail Component.....	376
Cart Component.....	380
CartRecord Component.....	384
Additional Thoughts.....	387
Summary.....	388
Index.....	389

About the Authors



Philip Japikse is an international speaker, Microsoft MVP, ASPInsider, MCSD, CSM, and CSP, and a passionate member of the developer community. He has been working with .NET since the first betas, developing software for over 30 years, and heavily involved in the Agile community since 2005. Phil is co-author of the best selling *C# and the .NET 4.6 Framework* (http://bit.ly/pro_csharp), the Lead Director for the Cincinnati .NET User's Group (<http://www.cinnug.org>), and the Cincinnati Software Architect Group. He also co-hosts the Hallway Conversations podcast (<http://www.hallwayconversations.com>), founded the Cincinnati Day of Agile (<http://www.dayofagile.org>), and volunteers for the National Ski Patrol. Phil enjoys learning new technologies and is always striving to improve his craft. You can follow Phil on Twitter via <http://www.twitter.com/skimedic> and read his blog at <http://www.skimedic.com/blog>.



Kevin Grossnicklaus was at one point in his career the youngster on most development teams. He got his start developing with Visual Studio and managed .NET code during the early beta cycles in 2001. In 2009, Kevin started his own software product development firm called ArchitectNow (www.architectnow.net). At ArchitectNow, Kevin and his team specialize in a wide variety of tools while delivering applications across a variety of cloud and mobile platforms. Born in rural Nebraska, he has spent the last 20 years in St. Louis, Missouri where he lives with his wife Lynda and their three daughters, Alexis, Emily, and Hanna. He is an avid guitar player, fly fisherman, home brewer, and gamer (including everything from retro arcade games, to board games, to role playing games). When he's not spending time on any of those hobbies, he waits patiently for a second season of *Firefly*.



Ben Dewey is a former Microsoft MVP and published author with over 18 years of experience writing applications. He continually strives to create SOLID applications of the highest craftsmanship while paying special attention to clean user experiences (UX). Ben is currently leading the User Experience team at Tallan, Inc. and consults regularly in New York City and around the country on web- and cloud-based technologies. He has also worked to deploy numerous high-quality, engaging apps to the Windows Store. When he's not consulting, Ben is busy training, mentoring, blogging, and speaking at various conferences and community events around the country. Outside of work, Ben spends most of his time playing with his three young kids, working around the house, or, if it's windy, kite surfing. You can find Ben online on Twitter (@bendewey), StackOverflow, GitHub, or on his blog at <http://bendewey.com/>.

About the Technical Reviewer



Eric Potter is a Software Architect for Aptera Software and a Microsoft MVP for Visual Studio and Development Technologies. He works primarily in the .NET web platform, but loves opportunities to try out other stacks. He has been developing high-quality custom software solutions since 2001. At Aptera, he has successfully delivered solutions for clients in a wide variety of industries. He loves to dabble in new and exciting technologies. In his spare time, he loves to tinker with Arduino projects. He fondly remembers what it was like to develop software for the Palm OS. He has an amazing wife and five wonderful children. He blogs at <http://humbletoolsmith.com/> and you can follow him on Twitter as @potteric.

Acknowledgments

Philip Japikse: This book could not have happened without the very talented (and patient) team at Apress. The idea for this book started when .NET Core was still called ASP.NET 5, and the ride from ASP.NET 5 to Visual Studio 2017 has been an interesting one, to say the least. This book also couldn't have happened without my loving wife Amy and all of the time she spent copy editing for me (for free) to keep my words from being a jumbled mess. I also want to thank my co-authors for all of their hard work. The goal of this book is to cover multiple technologies, and without their dedication to the cause, this book would have died a long time ago. Finally, I have to thank my children for their patience and understanding. Now that we are done, we can get out on the slopes and make a few turns!

Kevin Grossnicklaus: First, I'd like to express my extreme gratitude to Phil and Ben for the opportunity to be a part of this book at all. It has been a lot of fun and something I am honored to be a part of. An equally large thank you goes out to the team at Apress for being patient and helping pull everything together. Writing about technologies during their beta cycle is always challenging and getting three authors (including a chronically slow one like myself) to the finish line deserves some type of medal. Next, a shout out to my team at ArchitectNow for pushing me to keep on top of technology just as much now as I did early in my career. I am constantly amazed at all the cool technologies and products we get to use on a daily basis. That said, who knew we would be writing this much JavaScript in 2017? Finally, to my awesome wife Lynda and my three beautiful daughters, Alexis, Emily, and Hanna: Thanks for putting up with all the time I spend working, writing, or with all my other random hobbies. I love you all very much!

Ben Dewey: I'd like to thank a number of people who made this book happen. When Phil came to me with the dream to create a book for .NET developers and architects that would help navigate the real-world challenges and decisions that teams need to make, I was all in. His vision and direction helped shape the complete story that we delivered in this book. I'd also like to thank Apress and their wonderful team for their help and guidance while we chased the changes and releases of the Visual Studio 2017, through multiple RCs. My employer, Tallan (<http://www.tallan.com>), has continued to enable me to grow and reach communities around the country. They have allowed me to attend and speak at conferences throughout the year and have worked with me to foster an environment that respects knowledge and great software. Most importantly, thanks to my family, who put up with me and my long hours to deliver this book and supports me and my passions.

Introduction

This idea for this book came out of a discussion among conference speakers about the problem with keeping up with technology. Not only the rapid fire and revolutionary changes in the .NET ecosystem, but also the proliferation of JavaScript frameworks. I stated to the group:

“There needs to be a book designed to get someone up to speed on ASP.NET 5 and help them make informed decisions about which JavaScript framework (if any) to use. The problem is that the material on the popular JavaScript frameworks is too comprehensive. We need to have a book that gives enough information to enable informed decisions without having to invest weeks learning a framework that might not be a fit. Following the *fail fast* mantra from lean.”

A silence fell over the group. They all looked at me and said, “Great idea! When are you going to write it?”

Thus, this book was born, at least in concept. I knew that it was more than I could tackle on my own, so I reached out to two of my long-time friends, Ben and Kevin, and asked them if they would be interested. I knew they are deeply immersed in many of the JavaScript frameworks and would be a great match to round out the writing team.

The Goals of This Book

After much discussion, we settled on two main goals for this book. The first goal is to bring the .NET developer up to speed with Visual Studio 2017 and .NET Core, including Entity Framework Core, Core MVC Services (formerly Web API), and Core MVC Web Applications. The second goal is to cover a three different JavaScript frameworks (Angular2, Aurelia, and React) as well as client-side build tools and TypeScript. Each of the web applications will consist of the same UI and functionality, and all will use the same RESTful service and database as the backend.

Introducing the SpyStore Database

To keep the sample applications a reasonable size, we settled on a derivative of the IBuySpy database. This database shipped as a sample application in the .Net Framework 1.1 SDK, and I have been using a derivative of it as a test model ever since. The database design (rebranded *SpyStore* for this book) is simple enough to use for clear teaching, but complete enough to be a solid, workable base for writing this book. Figure 1 shows the ERD of the SpyStore database.



Figure 1. The SpyStore database

Introducing the SpyStore Web Site

Based on the SpyStore database and the list of features we wanted to show for each framework, the UI was completely reimaged and created by Ben Dewey, one of our co-authors. The site is very simple, consisting of the following site map.

The Home Page

The home page is also the Product list page, showing the Featured Products. Products can also be displayed for each Category by selecting one of the dynamically created menu items at the top of the page. The top right has links for the Shopping Cart and Order History pages, as well as the Search box. Figure 2 shows the home page listing the featured products.

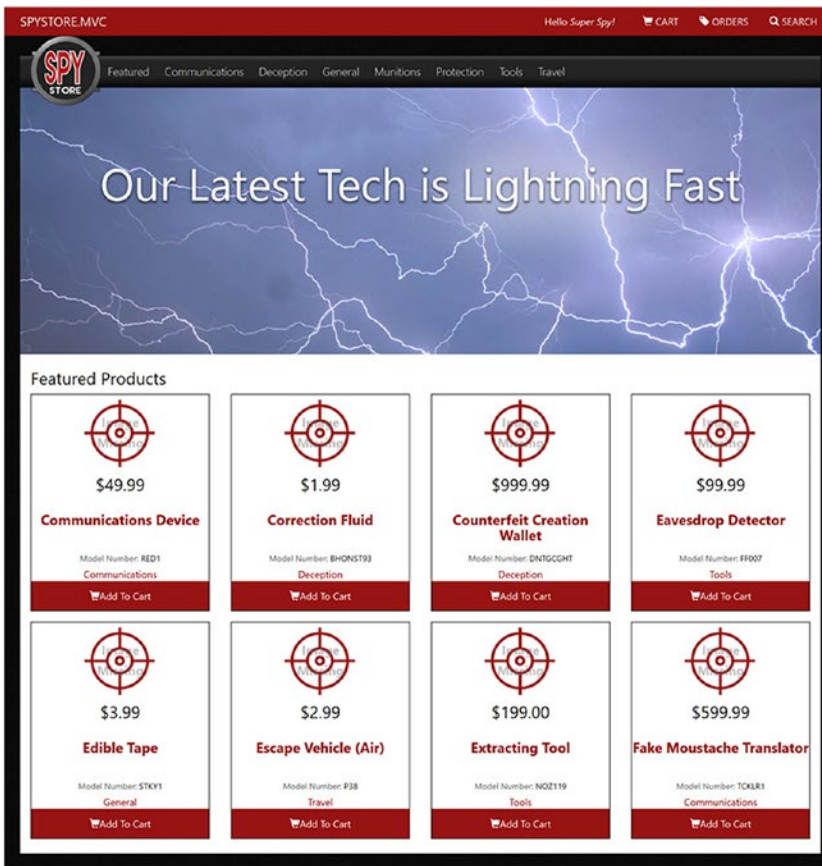


Figure 2. The home page with featured products

The site is also responsive, and it will alter its UI based on the view port. Figure 3 shows the home page as viewed on a mobile device.

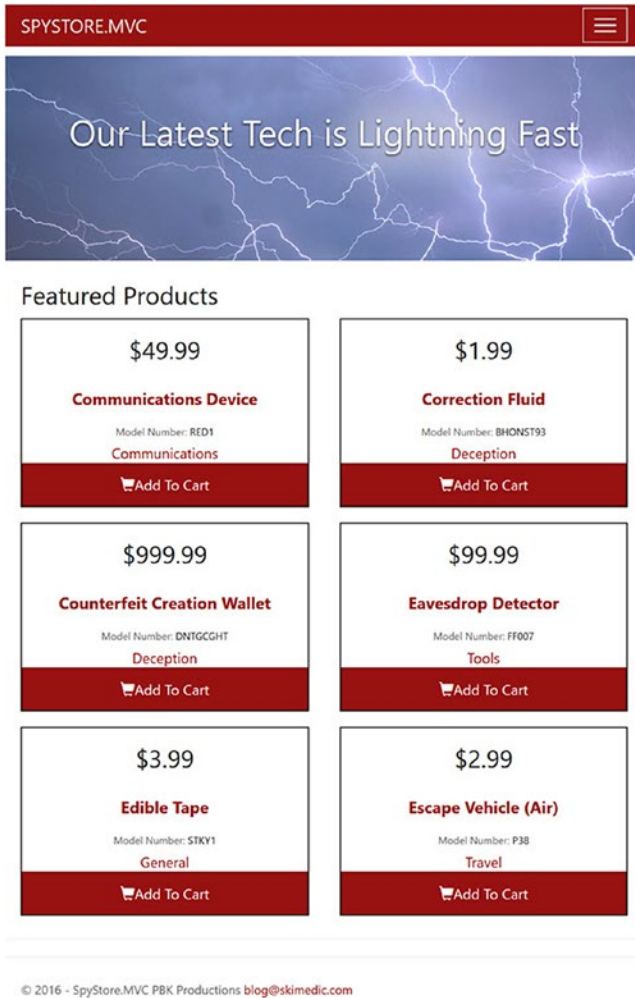


Figure 3. The home page on a mobile device

The Details/Add to Cart Page

The Product Details page doubles as the Add to Cart page. It's shown in standard view in Figure 4 and on a mobile device in Figure 5.

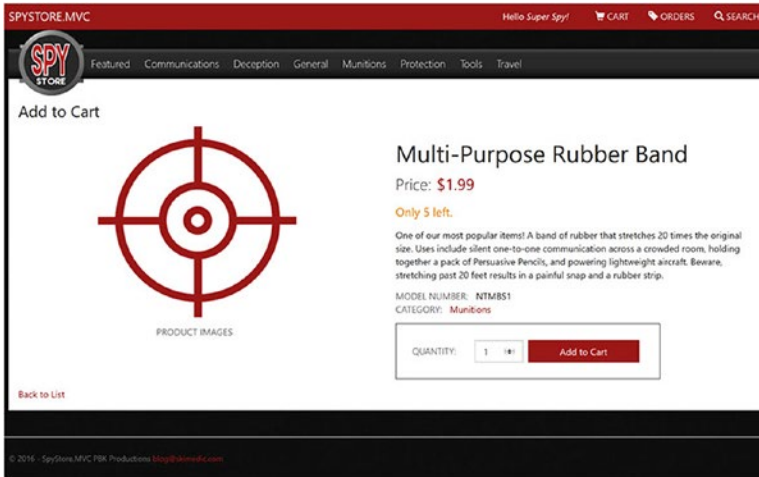


Figure 4. The Product Details/Add to Cart page



Figure 5. The Product Details/Add to Cart page on a mobile device

The Cart Page

The Cart page demonstrates the bulk of the work in the application, with methods to add, delete, and update Shopping Cart records. The Cart page is shown in standard view in Figure 6. The mobile view isn't any different, it's just smaller.

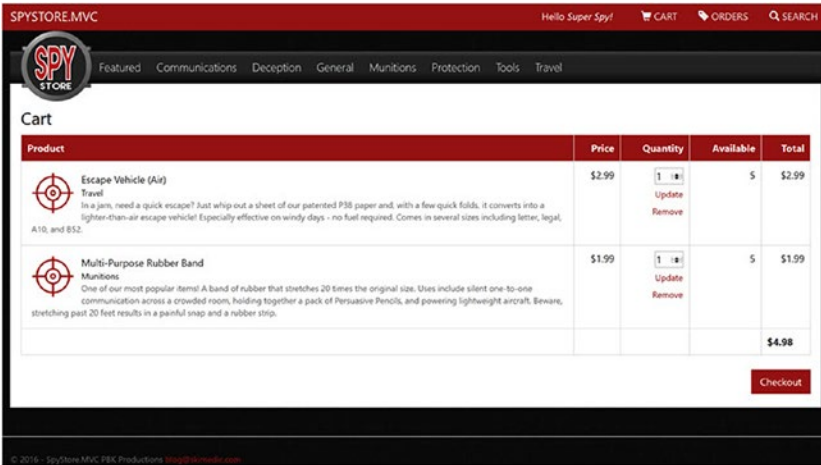


Figure 6. The Cart page

The Order History Page

The Order History page shows all of the top line details for a Customer's orders. The full-screen version is shown in Figure 7 and the mobile device version is shown in Figure 8.

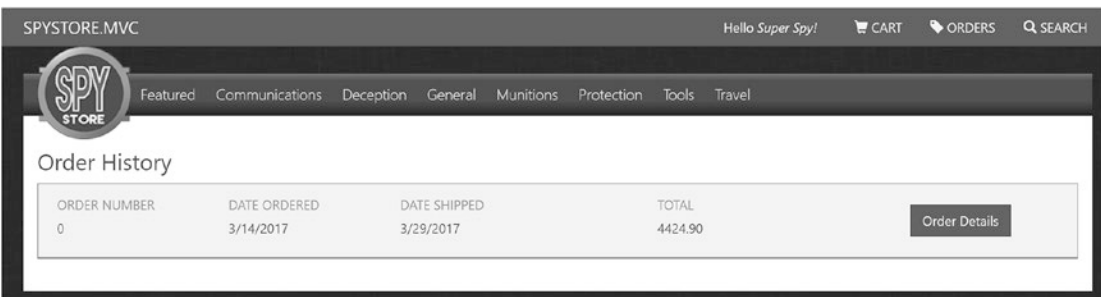


Figure 7. The Order History page

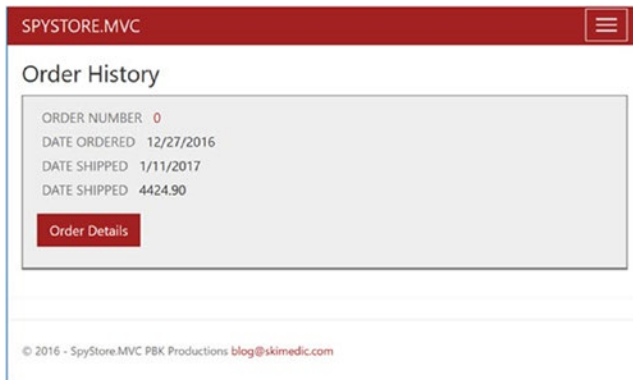


Figure 8. The Order History page on a mobile device

The Order Details Page

The Order Details page shows all of the details for an order, and it is shown in standard view in Figure 9 and on a mobile device in Figure 10.

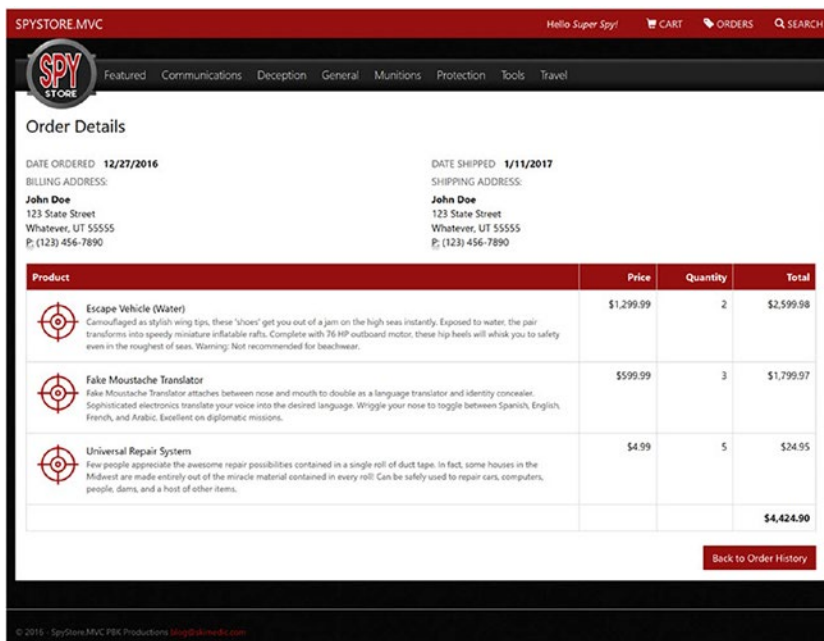


Figure 9. The Order Details page

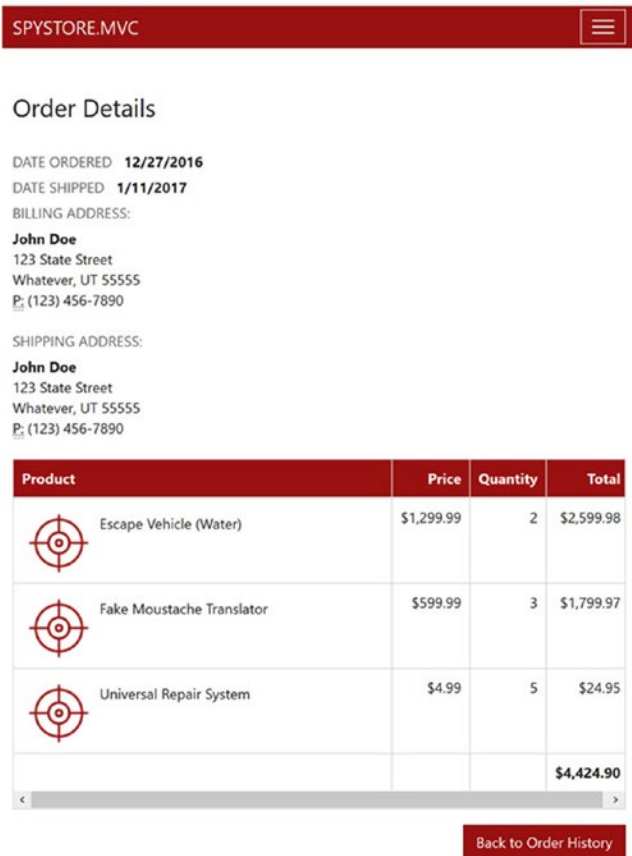


Figure 10. The Order Details page on a mobile device

How This Book Is Organized

This book is organized in two parts, which are aligned with the goals of the book. The first part focuses on Visual Studio 2017 and .NET Core, building the data access layer with Entity Framework Core, the RESTful service with Core MVC Services (formerly called Web API), and finally, the SpyStore web site with ASP.NET Core MVC.

The second part covers client-side build utilities (like Gulp, NPM, and Web Pack), TypeScript, and then builds the SpyStore site using Angular2, Aurelia, and React. The sites use the SpyStore.Service RESTful service built in Part I as the service endpoint.

Part I: Visual Studio 2017 and .NET Core

Part I covers building an end-to-end .NET Core application with Visual Studio 2017.

Chapter 1, “Introducing Entity Framework Core,” begins with installing Visual Studio 2017 and .NET Core SDK and creating the solutions and projects. Then it introduces Entity Framework Core, CRUD operations, and migrations. It then creates a complete data access layer for one table, complete with a repository and unit tests.

Chapter 2, “Building the Data Access Layer with Entity Framework Core,” covers the remaining Entity Framework concepts needed for this application. The rest of the tables from the SpyStore database are added into the data access layer, complete with repositories, migrations, and view models. The final section builds custom NuGet packages from the data access layer.

Chapter 3, “Building the RESTful Service with ASP.NET Core MVC Services,” introduces Core MVC in the context of building RESTful services, then builds the RESTful service for the SpyStore database. The service built in this chapter serves as the service endpoint for all of the web applications built in the rest of the book.

Chapter 4, “Introducing ASP.NET Core MVC Web Applications,” completes the Core MVC story by adding Actions and Views, package management, and bundling and minification. The last sections build core infrastructure for the application.

Chapter 5, “Building the SpyStore Application with ASP.NET Core MVC,” covers two new features in Core MVC—Tag Helpers and View Components. Server-side and client-side validation (including custom validation) are covered, and then the Views, Controllers, and Actions are built to finish the Core MVC Web Application. The final section covers the different ways to run the applications.

This completes the .NET portion of the book.

Part II: Client-Side Tooling and JavaScript Frameworks

Part II expands on the information learned in Part I and introduces architecture patterns needed to build modern JavaScript applications.

Chapter 6, “JavaScript Application Tools,” sets the groundwork for the JavaScript framework chapters and helps you understand the commonalities that exist between the future chapters. There is a core set of knowledge required regardless of which framework your team chooses. This chapter covers Node.js, NPM, Bower, Gulp, SystemJS, and WebPack.

Chapter 7, “Introduction to TypeScript,” introduces you to the core aspects of the TypeScript language. TypeScript will be the language used in later chapters to build the SpyStore interface in Angular and React.

Chapter 8, “Angular 2,” provides a step-by-step walkthrough of building the SpyStore application using Angular 2 and Visual Studio 2017. After guiding the reader through setting up a basic Angular 2 app in Visual Studio, the chapter describes the core architectural components of Angular. After that, the remainder of the chapter focuses on taking the core concepts and applying them to the relevant aspects of the application.

Chapter 9, “React,” walks through implementing the SpyStore application using React and Visual Studio 2017. The development workflow put in place will be built around WebPack and the core structure of the React application will be similar in many regards to that of the Angular solution built in Chapter 8. This will allow developers to compare both frameworks while seeing how they differ in the implementation of the same interface.

The Source Code

All of the source code for the examples in this book is available for download from the Apress site.

PART I



Visual Studio 2017 and .NET Core

CHAPTER 1



Introducing Entity Framework Core

ADO.NET contains everything a .NET developer needs to create data access layers for their applications, but very little (if anything) is done *for* the developer. For example, ADO.NET requires the developer to first create connections, SQL statements, commands, and data readers, and then translate the data from the database into the application's .NET classes that model the domain. When the user is done making changes to the data, the developer then needs to essentially reverse the process to persist any additions, deletions, or changes to the data.

Writing ADO.NET data access code takes time, and it is largely the same code from application to application, just swapping out the SQL statements (or stored procedures) and the model classes. Instead of spending time on data access plumbing, developers should be adding features and differentiators from the competition's applications. Those features could mean the difference between a booming business and a failed startup.

The availability of *Object Relational Mapping* frameworks (commonly referred to as ORMs) in .NET greatly enhanced the data access story by managing the bulk of those data access tasks. The developer creates a mapping between the domain models and the database, and the vast majority of the Create, Read, Update, and Delete (CRUD) operations are handled by the ORM. This leaves the developer free to focus on the business needs of the application.

■ **Note** If that sounds too good to be true, it is. ORMs are not magical unicorns riding in on rainbows. Every decision involves tradeoffs. ORMs reduce the amount of work for developers creating data access layers, but can also introduce performance and scaling issues if used improperly. As with any decision, make sure you understand the pros and cons of a framework before unilaterally adding it to your development stack. I have found that both Entity Framework 6.x and Entity Framework Core are extremely valuable tools in my toolkit. I also only use EF for CRUD operations, and use T-SQL for server-side set operations.

Microsoft's entry into the ORM landscape is *Entity Framework* (or simply, *EF*). Introduced in .NET 3.5 Service Pack 1, the initial versions were much maligned, including a (now) infamous "vote of no confidence" regarding EF version 1.0. The EF team at Microsoft has been hard at work releasing new versions, and since the first version, EF has seen evolutionary as well as revolutionary changes. The current version for the full .NET Framework is EF 6.1.3 and is being used widely throughout the .NET developer community.

Entity Framework Core (commonly referred to as EF Core) is a complete rewrite of Entity Framework based on .NET Core as an open source project on GitHub (<https://github.com/aspnet>). This new version of EF is designed to run on multiple platforms in addition to Windows, such as MacOS and Linux. In addition to enabling different platforms, the rewrite enables additional features that couldn't be reasonably added into previous versions of EF. This includes supporting non-relational data stores and in-memory databases (very helpful for unit testing), deploying to phones and tablets, and reducing the EF footprint of the install by modularization of the framework.

A big change from prior versions of EF (for some of you) is that support for the Entity Designer is not going to be added—EF Core only supports the Code First development paradigm, either from an existing database or by creating new one. If you are currently using Code First, you can safely ignore the previous sentence.

When .NET Core and the related frameworks (including ASP.NET Core and EF Core) went RTM in June 2016, EF received a fair amount of criticism for the list of missing features in the product (compared to EF 6.x). The features that *were* released were solid and performed significantly faster than their EF 6.x counterparts, but if your application needed any of the missing features, EF 6.x was the only choice. Version 1.1 for EF Core was released November 14, 2016, and significantly narrowed the gap between EF 6.x and EF Core. EF Core 1.1 is the version used in this book.

■ **Note** The full comparison of EF 6 and the current production version of EF is available here: <https://docs.microsoft.com/en-us/ef/efcore-and-ef6/features>, and the roadmap for EF Core is available on GitHub here: <https://github.com/aspnet/EntityFramework/wiki/Roadmap>.

Two chapters isn't nearly enough to cover all of Entity Framework Core, and this isn't meant to be an extensive reference for EF Core. But it covers enough for most (if not all) line of business applications, and includes what you need to know to effectively use EF Core in your projects.

■ **Note** If you didn't read the intro, the tl;dr version is this: This book covers building a fake e-commerce site we are calling *SpyStore*. It is loosely based on the IBuySpy database that shipped as a sample app with the .NET 1.1 SDK. The first two chapters build the data access layer and the next chapter builds a RESTful service that exposes the data access layer, and then the next two build the user interface in Core MVC. The rest of the book focuses on frontend technologies, including build tools, TypeScript, and then different JavaScript frameworks, including Angular 2, React, and Aurelia.

The SpyStore Database

In this chapter, you begin to build the data access library for the SpyStore database. The database is extremely simple: six related tables, two computed columns, one stored procedure, one user defined function, Identity columns for primary keys, and timestamps for concurrency checking. Figure 1-1 shows the Entity Relationship Diagrams (ERD) for the SpyStore database.



Figure 1-1. The SpyStore database

You will build this database using the Code First pattern. For reference, the complete database creation script `SpyStoreDatabase.sql` is included in the downloadable code for this chapter.

Note The database and web sites created in this book are not full featured e-commerce sites, but simply demo code useful for showing the techniques and concepts of the frameworks and languages covered. As such, we have ignored important subjects like security, authentication, authorization, and to a large extent, error handling. If it seems like a simplistic design that has gaps, that is intentional!

Installing Visual Studio 2017 and .NET Core

.NET Core and its related frameworks currently ship out of band from Visual Studio, so you need to install Visual Studio and .NET Core separately. The experience will be significantly improved when the tooling is fully baked.

Installing Visual Studio

If you haven't already installed Visual Studio 2017, you can get the free community edition at <https://www.visualstudio.com/en-us/visual-studio-homepage-vs.aspx>. All of the .NET samples in this book will work with the free community edition as well the paid versions. The install experience for VS2017 is very different than previous versions. Features are broken into workloads. Make sure you select the "ASP.NET and web development" and the ".NET Core cross platform development" workloads to include .NET Core, ASP.NET Core, and Entity Framework Core, as shown in Figure 1-2.

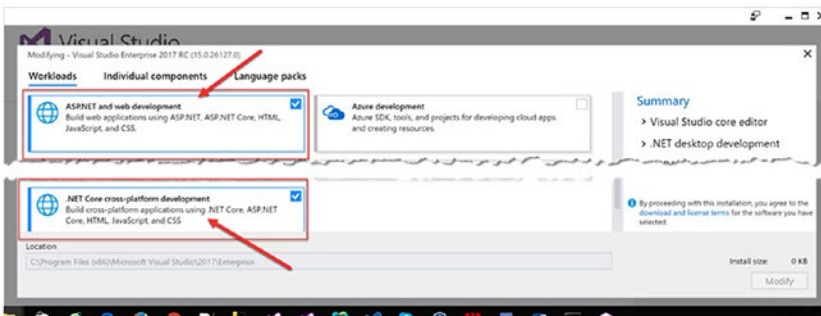


Figure 1-2. Installing Visual Studio 2017

■ **Note** This book is written using January 28, 2017 Release Candidate of Visual Studio 2017 (release notes and downloads are available here: <https://www.visualstudio.com/en-us/news/releasenotes/vs2017-relnotes>) and Preview 4 of the .NET Core tooling (installed with Visual Studio 2017). The most recent .NET Core SDK (both x86 and x64) are available here: <https://github.com/dotnet/core/blob/master/release-notes/rc3-download.md>. Historically, the RC tag is applied when a product is ready to be fully released except for bug fixes and security fixes. However, this isn't guaranteed, and it's possible that if you are using the fully released versions, your images and workflow might be slightly different in regards to Visual Studio and Visual Studio .NET Core tooling.

Visual Studio Code is a new tool (also free) that runs on Linux and Mac and works with .NET Core. Additionally, there is an early version of Visual Studio 2017 for the Mac. While it's not used to build the .NET Core apps in this book, all of the work in this book can be developed with either Visual Studio Code or Visual Studio 2017 for the Mac.

Installing the .NET Core SDKs

Microsoft set up a home page for everything .NET at <http://www.dot.net>. From here all released versions of .NET (including .NET Core) are accessible. This book is using .NET Core Version 1.1, which at the time of this writing is the current release.

■ **Note** With .NET Core, Microsoft is using a different approach to release. The Long Term Support (LTS) version is supported for at least one year after the next LTS release and are distinguished as Major Versions (e.g. 1.0, 2.0). They will only be updated with critical fixes. Version 1.0.3 is the LTS release in production. The current version is the latest released Minor Version (e.g. 1.1, 2.1), and will be supported for three months after the next Minor Version release. The complete story on the support lifecycle can be found here: <https://www.microsoft.com/net/core/support>.

To install the .NET SDKs (both x86 and x64 versions need to be installed), download them from here: <https://github.com/dotnet/core/blob/master/release-notes/rc3-download.md>. After installing them, to confirm the version that is currently installed, enter the following commands in a command prompt:

```
dotnet
dotnet --info
```

The `dotnet` command shows the version of .NET Core on your machine (your output may be different based on the version installed):

```
C:\Users\skime>dotnet
```

```
Microsoft .NET Core Shared Framework Host
```

```
Version : 1.1.0
Build   : 928f77c4bc3f49d892459992fb6e1d5542cb5e86
```

```
Usage: dotnet [common-options] [[options] path-to-application]
```

Common Options:

```
--help           Display .NET Core Shared Framework Host help.
--version        Display .NET Core Shared Framework Host version.
```

Options:

```
--fx-version <version>  Version of the installed Shared Framework to use to run the
                           application.
--additionalprobingpath <path>  Path containing probing policy and assemblies to probe for.
```

Path to Application:

The path to a .NET Core managed application, dll or exe file to execute.

If you are debugging the Shared Framework Host, set 'COREHOST_TRACE' to '1' in your environment.

To get started on developing applications for .NET Core, install the SDK from: <http://go.microsoft.com/fwlink/?LinkID=798306&clcid=0x409>

The `dotnet -info` command shows the version of the .NET Core tooling on your machine (your output may be different based on the version installed):

```
C:\Users\skime>dotnet --info
```

```
.NET Command Line Tools (1.0.0-rc3-004530)
```

Product Information:

```
Version:           1.0.0-rc3-004530
Commit SHA-1 hash: 0de3338607
```

Runtime Environment:

```
OS Name:           Windows
OS Version:        10.0.14986
```

OS Platform: Windows
RID: win10-x64
Base Path: C:\Program Files\dotnet\sdk\1.0.0-rc3-004530

At the time of this writing, the current version of the .NET Core *framework* is 1.1.0, and the version of the *tooling* for .NET Core is version 1.0.0-rc3-004530.

The .NET Core Command Line Interface (CLI)

The commands just used are part of the .NET Core Command Line Interface, or CLI. .NET Core places an emphasis on command line support, and in many places, the command line tooling is better than the GUI tooling in Visual Studio. The commands all start with the invoker `dotnet`, followed by a specific command. Each framework in the .NET Core ecosystem has its own commands, including EF Core. You will use the Core CLI throughout the .NET chapters in this book.

Creating and Configuring the Solution and Projects

Separation of Concerns (SoC) is a design principle in software development where applications are partitioned based on distinct responsibilities. This leads to more maintainable code and cleaner implementations. In this book, the data access layer, RESTful service, and each UI will be developed following this principle.

The data access layer will be developed in its own solution following a repository pattern. The repositories shield the calling code from having to understand specifics about how the data access works. The solution consists of two projects, one to hold the application models and the other to hold all of the EF Core code. This further separation enables Core MVC applications to reference the strongly typed models without needing all of the EF Core code as well, as additional projects are added to the solution for testing.

Creating the Solution and Projects

Start by creating a new solution and Console Application project. Typically, a data access layer is built using just class libraries, but due to an issue in the current version of .NET Core, migrations only work on console application projects. From the Start Screen in Visual Studio, click on More Project Templates... under New Project, as shown in Figure 1-3.

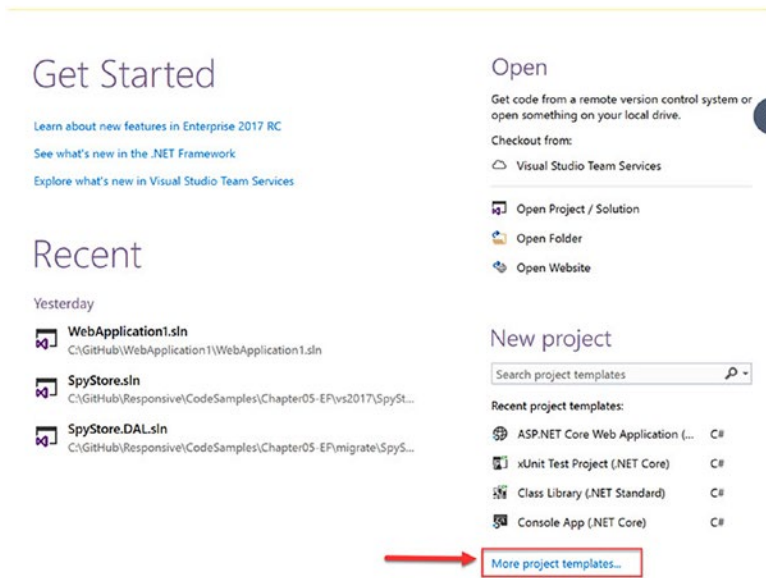


Figure 1-3. Creating the *SpyStore.DAL* project and solution

In the New Project dialog box, select the **Console App (.NET Core)** template located in **Installed** ► **Visual C#** ► **.NET Core**. Name the project *SpyStore.DAL*, as shown in Figure 1-4. Click OK to create the project.

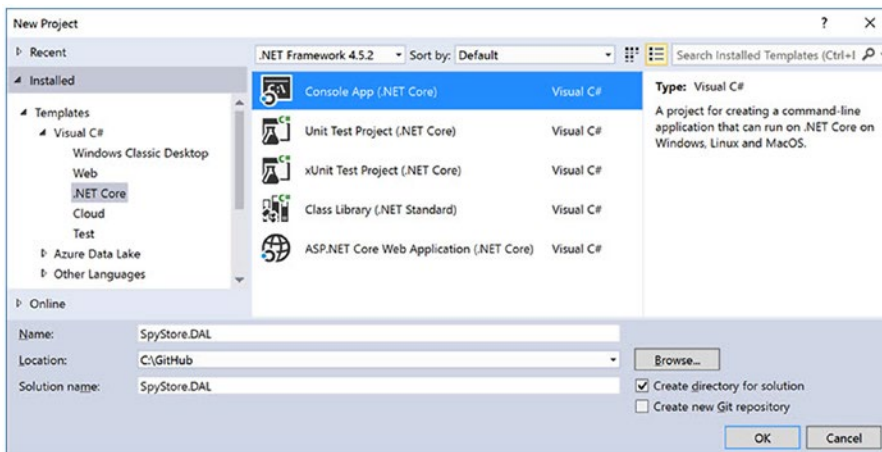


Figure 1-4. Creating the *SpyStore.DAL* project and solution

Open `Program.cs` and delete the following line from the `Main` method:

```
Console.WriteLine("Hello World!");
```

The file should contain the following after the editing:

```
using System;
class Program
{
    static void Main(string[] args)
    {
    }
}
```

Note Previous versions of .NET allowed for data access libraries to be Class Library projects. Core is different. If you can't execute `dotnet run` against a project, you also can't execute `dotnet ef`. This is because .NET Core doesn't have a globally installed framework, but is installed on a per project basis. Due to this, in order to run migrations for the data access library, the project type must be a Console Application.

Next, add a class library to the solution to hold the models. Right-click on the solution name and select `Add ► New Project...`, as shown in Figure 1-5.

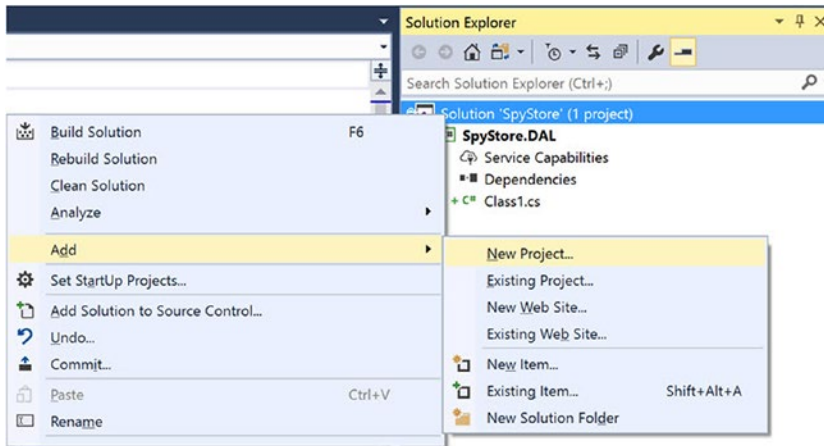


Figure 1-5. Adding a new project

Select the `Class Library (.NET Standard)` template and name the project `SpyStore.Models`, as shown in Figure 1-6. As with prior versions of Visual Studio, the new project is automatically added into the solution. Click `OK` to create the new project and close the wizard.

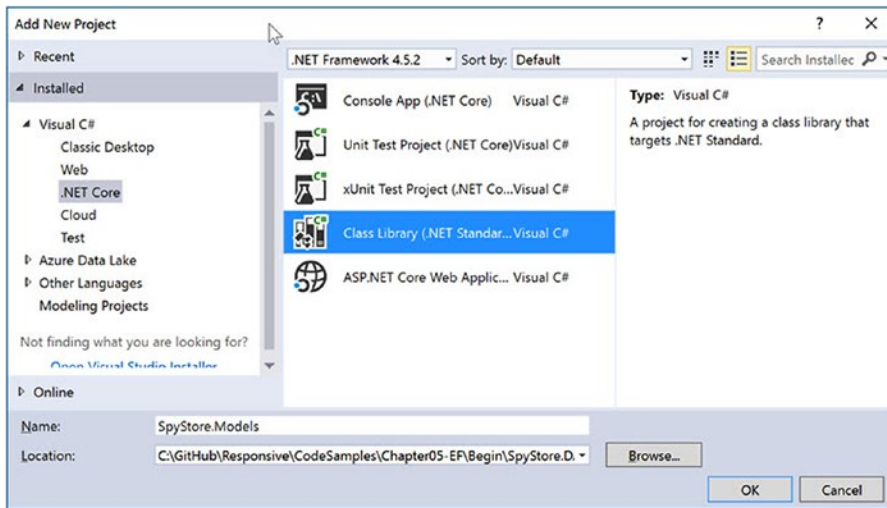


Figure 1-6. Adding the *SpyStore.Models* project

Finally, add the unit test project. Even though there are two different templates for unit test projects, you are going to build one from scratch. Right-click on the solution and select **Add ► New Project...**, then select the **Console App (.NET Core)** template and name the project `SpyStore.DAL.Tests`. Click **OK** to create the new project and close the wizard.

Changes to the Project Files

If you've been tracking the .NET Core story, there have been a myriad of changes. What started as ASP.NET5 had grown to .NET Core, ASP.NET Core, and EF Core. The command line tools changed from DNX and DNU to DOTNET. The project files started as `project.json` and `*.xproj` and then (with VS2017) reverted back to `*.csproj`. The good news is that the drastic changes are winding down with the release of VS2017, and the landscape is beginning to stabilize. The removal of `project.json` brings .NET Core projects back to being MSBuild compatible and aligns them with the rest of the .NET family in regards to tooling.

The Visual Studio 2017 `<projectname>.csproj` file has some nice updates over previous versions of Visual Studio. One of the biggest changes is the replacement of individual file listings with project directories, also referred to as *file globbing*. This carries forward one of the major benefits of the `project.json` file and the folder structure scoping of .NET Core projects. Also, all of the NuGet package information for the project is contained in this file (instead of a separate packages file), and finally, additional nodes were added for the .NET Core command line tooling (replacing the "tools" node in `project.json`).

VS2017 enables editing the project file directly through Visual Studio. This is a much-improved workflow over the previous method of editing the file in an external text editor and then reloading the project.

Updating the Target Framework

The default templates in Visual Studio target the LTS version of .NET Core, which is currently 1.0.3. The projects all need to be updated to 1.1. Right-click on the `SpyStore.DAL` project and select **Properties**. On the **Application** tab, change the **Target Framework** to `NETCoreApp 1.1`, as shown in Figure 1-7. Do this for the `SpyStore.DAL.Tests` project as well.

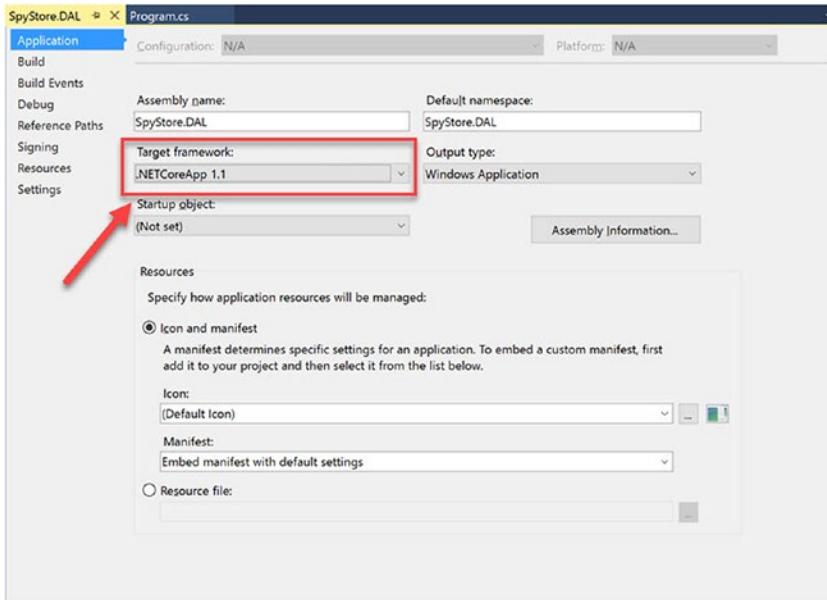


Figure 1-7. Updating the target framework to NETCoreApp 1.1

Next, right-click on the `SpyStore.Models` project and select Properties. For this project, update the target framework to NETStandard 1.6, as shown in Figure 1-8.

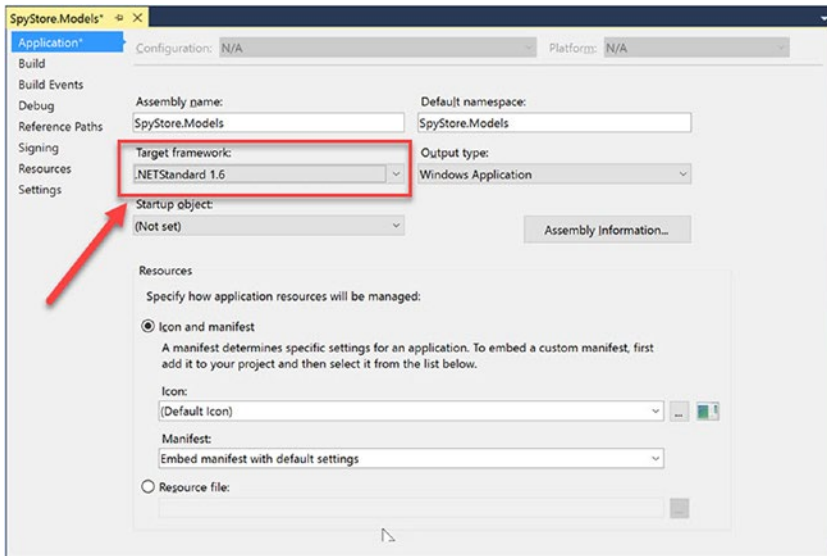


Figure 1-8. Updating the target framework to NETStandard 1.6

Working with NuGet Packages

EF Core and the other .NET Core frameworks are distributed as NuGet packages. When a new project is created using Visual Studio, the actual NuGet packages aren't included, just a reference to those packages, so the packages need to be restored. Most source code control systems also don't include the package files, so a restore needs to happen when projects are loaded from source code control. This is a good thing, since it saves considerable space in the template files and source code control systems. It also means that you must have Internet access to work with .NET Core projects.

Visual Studio executes a package restore when a new project is created. A message to this effect is displayed in the Solution Explorer window briefly when the new project wizard finished. Visual Studio also executes a restore when a change is detected in the project file. Packages can also be restored on demand.

■ **Note** NuGet is updated at a fairly rapid cadence. If you receive errors while restoring, updating, or adding packages, try updating your version of NuGet Package Manager by downloading the latest version from <https://www.nuget.org/>.

Manually Restoring Packages

Packages can be restored manually from Package Manager Console, from the command prompt, or through Visual Studio using the NuGet Package Manager GUI.

Restoring Packages from the CLI

You can restore the NuGet packages in your solution (or project) using the .NET Core CLI by entering `dotnet restore` in a standard command prompt. The command restores the packages for every `*.csproj` file that it finds in the directory tree at and below where the command was executed. In other words, if you enter the command at the solution level, the packages in all of the projects in your solution will be restored (if they are in the same directory tree, of course). If you enter it at a project level, only the packages for that project are restored.

Restoring with Package Manager Console

You can also force a package refresh through Package Manager Console (PMC) in Visual Studio. To restore packages using PMC, first open it by selecting **View** ► **Other Windows** ► **Package Manager Console**. The console window will load in the bottom of the Visual Studio workspace by default.

Enter the same command (`dotnet restore`) at the prompt. The same rules apply—every project in the directories below where the command is executed will be updated. The default project setting in PMC doesn't matter in this case—it's the current directory being used by the PMC. You can check the current directory by typing `dir` at a PMC prompt and changing the current directory using the `cd` command.

Adding the Project References

The `SpyStore.DAL` project needs to reference the `SpyStore.Models` project. Add the reference by right-clicking the `SpyStore.DAL` project in Solution Explorer, select **Add ► Reference**, and then click on **Projects** in the left side of the Reference Manager. Check the box next to `SpyStore.Models`, as shown in Figure 1-9. Click **OK** to add the reference.

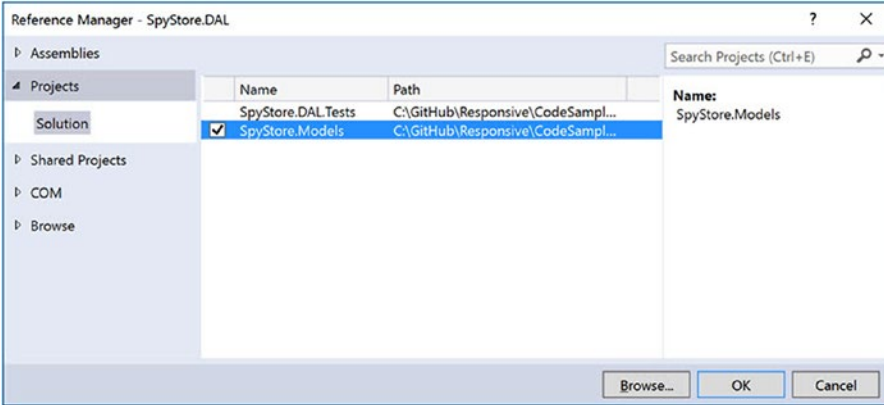


Figure 1-9. Adding the project reference

The `SpyStore.DAL.Tests` project needs to reference the `SpyStore.DAL` and `SpyStore.Models` projects. Add the references by right-clicking the `SpyStore.DAL.Tests` project in Solution Explorer, selecting **Add ► Reference**, and then selecting the `SpyStore.Models` and `SpyStore.DAL` projects in the References Manager dialog. Click **OK** after selecting the projects to finish adding the references.

Adding Entity Framework Core

The required EF Core packages can be added to a project by manually updating the `*.csproj` file, using PMC, or using the NuGet Package Manager. The simplest way is to use NuGet Package Manager.

Adding EF Packages to the SpyStore.DAL Project

Adding packages with NuGet Package Manager is just as easy as updating the package versions just completed. Open NuGet Package Manager by right-clicking on the `SpyStore.DAL` project and selecting **Manage NuGet Packages**. Make sure the **Include Prerelease** box is checked, then click on **Browse** in the top-left corner and enter `Microsoft.EntityFrameworkCore` into the search box. Install all of the `Microsoft.EntityFrameworkCore` packages shown in Table 1-1. Change the search to `System.Linq.Queryable` and add the final package from the table.

Table 1-1. *Entity FrameworkCore Packages*

Package	Version
Microsoft.EntityFrameworkCore	The core package of the framework 1.1.0
Microsoft.EntityFrameworkCore.Design	The shared design time components of EF Core tools 1.1.0
Microsoft.EntityFrameworkCore.Relational	Shared components for relational database providers 1.1.0
Microsoft.EntityFrameworkCore.Relational.Design	Shared design time components for relational database providers 1.1.0
Microsoft.EntityFrameworkCore.SqlServer	Microsoft SQL Server database provider for EF Core 1.1.0
Microsoft.EntityFrameworkCore.SqlServer.Design	Design time EF Core functionality for MS SQL Server 1.1.0
Microsoft.EntityFrameworkCore.Tools	Visual Studio Commands for EF Core 1.0.0-msbuild3-final

This package needs to be installed by hand:

Microsoft.EntityFrameworkCore.Tools.DotNet (see note)	.NET Core SDK command line tools for EF Core	1.0.0-msbuild3-final
---	---	----------------------

■ **Note** At the time of this writing, the `Microsoft.EntityFrameworkCore.Tools.DotNet` package can't be properly installed properly through the NuGet package manager. The package must be installed by manually updating the project file, as you will do next. Depending on when you are reading this book, the versions will most likely be updated from Table 1-1. When selecting packages, pay close attention in NuGet Package Manager to make sure that you are picking the correct versions, especially when working with pre-release packages.

Installing/Updating Packages Using the SpyStore.DAL.csproj File

You can also update and install NuGet packages directly in the `SpyStore.DAL.csproj` file. Right-click on the `SpyStore.DAL` project and select `Edit SpyStore.DAL.csproj`, as shown in Figure 1-10.

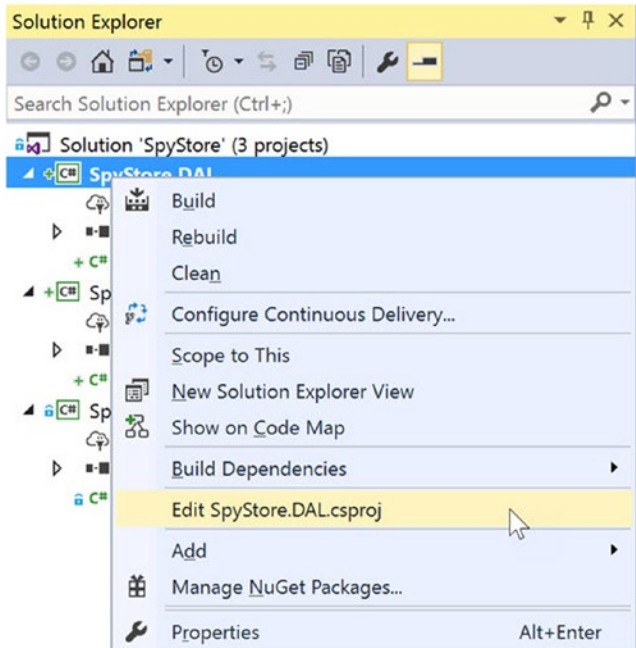


Figure 1-10. Editing the project file

Tooling references are placed in their own item group. .NET Core CLI tools are added with a `DotNetCliToolReference` tag. Add a new `ItemGroup` and then add the following markup. This enables the `dotnet ef` command line functions that will be utilized later.

```
<ItemGroup>
  <DotNetCliToolReference Include="Microsoft.EntityFrameworkCore.Tools.DotNet">
    <Version>1.0.0-msbuild3-final</Version>
  </DotNetCliToolReference>
</ItemGroup>
```

Future releases of VS2017 will probably provide a user interface to do this, but for now, manually entering these values is part of the joy of working with pre-release software!

Adding EF Packages to the SpyStore.Models Project

The `SpyStore.Models` project only needs two packages added:

- `Microsoft.EntityFrameworkCore` (v1.1.0)
- `Microsoft.EntityFrameworkCore.SqlServer` (v1.1.0)

To add them, open NuGet Package Manager and repeat the previous steps to add the two packages.

Adding Packages to the SpyStore.DAL.Tests Project

The `SpyStore.DAL.Tests` project needs the following packages added:

- `xUnit` (at least 2.2.0-beta5-build3474)
- `xunit.runner.visualstudio` (at least 2.2.0-beta5-build1225)
- `Microsoft.NET.Test.Sdk` (v15.0.0-preview-20170125-04))
- `Microsoft.EntityFrameworkCore` (v1.1.0)
- `Microsoft.EntityFrameworkCore.Design` (v1.1.0)
- `Microsoft.EntityFrameworkCore.Relational` (v1.1.0)
- `Microsoft.EntityFrameworkCore.SqlServer` (v1.1.0)

To add them, open NuGet Package Manager and repeat the previous steps to add the required packages.

Building the Foundation

The process that will be followed to create the data access layer is fairly standard, and one that I've mastered since my adoption of ORMs.

1. Create the context class.
2. Create a base entity class.
3. Create and configure model class(es).
4. Add the model to the context as a `DbSet<T>`.
5. Create a migration and run it to update the database.
6. Add a strongly typed repository for the model.

Creating a base entity class is optional, but commonly done to hold common properties across all entities. Adding a custom repository implementation is also optional, and largely dependent on the architecture of the project. The `DbContext` provides some Unit of Work and Repository implementation details, so technically, including any repository functionality beyond that is unnecessary. However, it is common for developers to add their own repository implementation to enable tighter control of the CRUD process and increase code reuse.

Understanding the DbContext Class

The MSDN Library defines the `DbContext` instance as a combination of the Unit of Work and Repository patterns. It provides access to the model collections in your data model (populating the collections from the data store when needed), exposes CRUD functionality on your collections (in addition to the functions exposed through the `DbSet<T>` specialized collection classes), and groups changes into a single call as a unit of work. Table 1-2 and 1-3 list some of the properties and methods of the `DbContext` used in this book.

Table 1-2. *Some of the Properties Exposed by DbContext*

DbContext Property	Definition
Database	Provides access database related information and functionality, including execution of SQL statements.
Model	The metadata about the shape of entities, the relationships between them, and how they map to the database.
ChangeTracker	Provides access to information and operations for entity instances this context is tracking.

Table 1-3. *Some of the Methods Exposed by DbContext*

DbContext Method	Definition
AddAdd<TEntity>	Begins tracking the entity. Will be inserted in the data store when SaveChanges is called. If type isn't specified, the DbContext will determine which DbSet the entity should be added into. Async versions are also available. Note: The DbSet<T> versions are typically used.
AddRange	Adds a list of entities to the change tracker instead of just one at a time. Entities do not have to all be of the same type. An async version is also available. Note: The DbSet<T> version is typically used.
Find<TEntity>	Searches the change tracker for an entity of the given by primary key. If it is not found, the data store is queried for the entity. An async version is also available. Note: The DbSet<T> version is typically used.
RemoveRemove<TEntity>	Prepares an entity for deletion from the data store. Will be deleted from the data store when SaveChanges is called. If type isn't specified, the DbContext will determine which DbSet the entity should be removed from. Async versions are also available. Note: The DbSet<T> versions are typically used.
RemoveRange	Prepares a list of entities to be deleted from the data store instead of just one at a time. Entities do not have to all be of the same type. An async version is also available. Note: The DbSet<T> version is typically used.
UpdateUpdate<TEntity>	Prepares an entry to be updated in the data store. Will be updated when SaveChanges is called. If the type isn't specified, the DbContext will determine which DbSet the entity belongs to. Async versions are also available. Note: The DbSet<T> versions are typically used.
SaveChangesSaveChangesAsync	Saves all entity changes to the database and returns the number of records affected.
Entry(T entity)	Provides access to change tracking information and operations (such as changing the EntityState) for the entity.
OnConfiguring	A builder used to create or modify options for this context. Executes each time a DbContext instance is created.
OnModelCreating	Called when a model has been initialized, but before it's finalized. Methods from the Fluent API are placed in this method to finalize the shape of the model.

EF Core data access layers need a custom class that derives from the `DbContext` base class. The next section covers building the `StoreContext` class.

Creating the `StoreContext` Class

To get started, create a new folder in the `SpyStore.DAL` project named `EF`. In this folder, add a new class named `StoreContext.cs`. Add the following namespaces to the class:

```
using Microsoft.EntityFrameworkCore;
using Microsoft.EntityFrameworkCore.Infrastructure;
using SpyStore.Models.Entities;
```

Inherit from `DbContext` and add the following two constructors, as shown here:

```
namespace SpyStore.DAL.EF
{
    public class StoreContext : DbContext
    {
        public StoreContext()
        {
        }
        public StoreContext(DbContextOptions options) : base(options)
        {
        }
    }
}
```

The second constructor takes an instance of the `DbContextOptions` class. The options that can be passed in include settings for the database provider, connection resiliency, and any other options specific to your application. The options are typically passed in through a dependency injection (DI) framework, such as is used in ASP.NET Core. This allows automated tests to inject a different instance for testing purposes (such as the `InMemory` provider) as well as configure EF Core to use development or production servers without changing any of the data access code.

EF Core has a fallback mechanism in case the parameterless constructor is used to create the `StoreContext` class, the `OnConfiguring` event handler.

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
}
```

The event fires after a new context class is constructed but before any methods can be called on the class. The `DbContextOptionsBuilder` class has a property called `IsConfigured` that's set to true if the context was configured in the constructor, and false if not. By checking this property value in the `OnConfiguring` event handler, the developer can ensure that the context is configured. While dependency injection is the preferable way to create a new instance of a `DbContext` class, the `OnConfiguring` handler is useful for situations such as unit testing. It is also used in determining the default database type and connection

string to target when creating and applying migrations. Add the following code to the `StoreContext` class (updating the connection string to match your machine's configuration):

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    if (!optionsBuilder.IsConfigured)
    {
        optionsBuilder.UseSqlServer(
@"Server=(localdb)\mssqllocaldb;Database=SpyStore;Trusted_Connection=True;MultipleActive
ResultSets=true;");
    }
}
```

Unlike prior versions of EF, the modularity of EF Core requires an application to add the database providers and configure them for use. In this example, the context is configured to use the SQL Server database provider, using the LocalDB instance and the database `SpyStore`.

Understanding the `DbSet<T>` Collection Type

Each model collection is represented by a `DbSet<T>` specialized collection property in the `DbContext` class. Each `DbSet<T>` property contains all of the like entities (e.g., `Category` records) tracked in the context. Table 1-4 lists the methods available on `DbSet<T>` that are used in this data access layer.

Table 1-4. Some of the Methods Exposed by `DbSet<T>`

<code>DbSet<T></code> Method	Definition
Add/AddRange	Begins tracking the entity/entities in the <code>Added</code> state. Item(s) will be added when <code>SaveChanges</code> is called. Async versions are available as well.
Find	Searches for the entity in the <code>ChangeTracker</code> by primary key. If not found, the data store is queried for the object. An async version is available as well.
Update/Update	Begins tracking the entity/entities in the <code>Modified</code> state. Item(s) will be updated when <code>SaveChanges</code> is called. Async versions are available as well.
Remove/Remove	Begins tracking the entity/entities in the <code>Deleted</code> state. Item(s) will be removed when <code>SaveChanges</code> is called. Async versions are available as well.
Attach/AttachRange	Begins tracking the entity/entities in the <code>Unchanged</code> state. No operation will execute when <code>SaveChanges</code> is called. Async versions are available as well.

■ **Note** This book is only presenting the synchronous versions of the EF methods, and leaves it to the reader to determine if they should call the asynchronous versions of the `DbSet<T>` and `DbContext` methods. Using the asynchronous methods is a minor syntactical change. Just make sure that you thoroughly test everything.

Connection Resiliency

When dealing with any database, but especially with cloud-based databases (such as SQL Azure), applications benefit from a retry strategy in case of a connectivity hiccup. These hiccups are usually referred to as *transient errors*. EF Core Version 1.1 added the ability to define a *connection strategy*, long available in EF 6. A connection strategy defines a set of errors that should invoke a retry operation, the number of retries, and the amount of time to wait between the retries.

The SQL Server provider for EF Core provides a `SqlServerRetryExecutionStrategy` class. The class defaults the retry count to 5 with a delay between retries of 30 seconds. This connection strategy uses the `SqlServerTransientExceptionDetector` to determine if the action should be retried. The `EnableRetryOnFailure` method is a handy shortcut to configure the retry limit and retry wait time to the default values, as well as the ability to add errors to the retry list.

Update the `OnConfiguring` to the following:

```
protected override void OnConfiguring(DbContextOptionsBuilder optionsBuilder)
{
    if (!optionsBuilder.IsConfigured)
    {
        optionsBuilder.UseSqlServer(
@"Server=(localdb)\mssqllocaldb;Database=SpyStore;Trusted_Connection=True;MultipleActiveResultSets=true;", options => options.EnableRetryOnFailure());
    }
}
```

Custom Connection Strategies

A custom connection strategy can also be created. Create a new class in the EF directory of the `SpyStore.DAL` project named `MyConnectionStrategy.cs`. Add a `using` for `Microsoft.EntityFrameworkCore.Storage` and derive the class from `ExecutionStrategy`. Add the following code:

```
using System;
using Microsoft.EntityFrameworkCore.Storage;

namespace SpyStore.DAL.EF
{
    public class MyExecutionStrategy : ExecutionStrategy
    {
        public MyExecutionStrategy(ExecutionStrategyContext context) :
            base(context, ExecutionStrategy.DefaultMaxRetryCount,
                ExecutionStrategy.DefaultMaxDelay)
        {
        }

        public MyExecutionStrategy(
            ExecutionStrategyContext context,
            int maxRetryCount,
            TimeSpan maxRetryDelay) : base(context, maxRetryCount, maxRetryDelay)
        {
        }
    }
}
```

```

protected override bool ShouldRetryOn(Exception exception)
{
    return true;
}
}
}

```

Both constructors require an `ExecutionStrategyContext` (which comes from the `DbContextOptionsBuilder`). The first constructor defaults the retry count and retry delay to 5 and the 30 seconds, respectively, just like the default `SqlServerRetryExecutionStrategy`. The `ShouldRetryOn` method returns true if EF should retry the operation, and false if it should not. In this example, it always returns true, so as long as there are retry counts left, it will always retry. Not a good implementation for production, but useful for testing error handling.

Adding the custom strategy to a context is as simple as replacing the `EnableRetryOnFailure` with the following code:

```

contextOptionsBuilder.UseSqlServer(connectionString,
    options => options.ExecutionStrategy(c=> new MyExecutionStrategy(c)));

```

Building the Base Entity Class

All of the tables in the database have a primary key named `Id` and a `Timestamp` field named `TimeStamp`. Since all models have these properties, they will be placed in a base class. EF Core pulls properties in base classes and derived classes together when shaping the database tables.

■ **Note** At the time of this writing, *complex types* are not yet supported in EF Core. This is a feature of EF 6.x where a class is used as a property on a model and pulled into the database appropriately. This is a technique used frequently in domain driven design. This feature is on the roadmap for EF Core, but nothing has been announced as to when it will be delivered. For now, EF Core supports inheritance, but not complex types.

In the `SpyStore.Models` project, delete the `Class1.cs` file that was auto-generated by the template. Add a new folder in the project named `Entities` by right-clicking on the `SpyStore.Models` project and selecting `Add ► New Folder`. Create a new folder under the `Entities` folder named `Base`. In the `Base` folder, add a new class named `EntityBase.cs` by right-clicking on the folder and selecting `Add ► Class`. Add the following namespaces to the class:

```

using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

```

Make the class abstract (since it will only ever be inherited from) and add two properties—`Id (int)` and `TimeStamp (byte[])`—as follows:

```
namespace SpyStore.Models.Entities.Base
{
    public abstract class EntityBase
    {
        public int Id { get; set; }
        public byte[] TimeStamp { get; set; }
    }
}
```

If this looks simple, it is! EF model classes are just Plain Old CLR Objects (POCOs). Shaping of the database tables and fields and mapping the classes to the database is done through the EF built-in conventions, data annotations (from `System.ComponentModel`), and the EF Fluent API. Each of these will be covered as needed in this book.

■ **Note** There isn't enough space in this book to cover EF conventions, data annotations, and Fluent API in their entirety. To explore all of the options available in shaping entities and the backing database, the full documentation is at the following URL: <https://docs.microsoft.com/en-us/ef/core/modeling/>.

Entity Framework Conventions

There are many conventions built into EF, and if you understand them, they are very beneficial in reducing the amount of manual configuration needed to properly shape the database. However, there is a lot to know, and that can cause problems. For example, the primary key convention creates a primary key for a table when EF finds a field named `Id` or `<typename>Id` (such as `CustomerId` on a class named `Customer`). For SQL Server, if the datatype of the property is `int` or `GUID`, the values will be generated by the server when a new record is added. Any other datatype will need to be configured by the developer. Each database provider has its own variations of this convention as well.

If the application relied only on EF conventions to shape the database, each developer would need to know and understand that convention. And that's just one convention! The more "secret sauce" a project has, the greater the opportunity for developer confusion and errors. Instead of relying solely on conventions, a common practice (and one used in this book) is to be explicit as possible when defining entities by using data annotations and/or the Fluent API. For the base class, the explicit definition will be done through data annotations, covered in the next section.

Data Annotations Support in EF Core

Data annotations are applied as .NET attributes and are a key mechanism for shaping the database. Unlike the EF conventions, data annotations are explicit and are much more developer friendly. Table 1-5 lists some of the more common data annotations that are available in EF Core (and used in this book).

Table 1-5. *Some of the Data Annotations Supported by EF Core*

Data Annotation	Definition
Table	Defines the table name for the model class. Allows specification of the schema as well (as long as the data store supports schemas).
Column	Defines the column name for the model property.
Key	Defines the primary key for the model. Not necessary if the key property is named <code>Id</code> or combines the class name with <code>Id</code> , such as <code>OrderId</code> . Key fields are implicitly also <code>[Required]</code> .
Required	Declares the property as not nullable.
ForeignKey	Declares a property that is used as the foreign key for a navigation property.
InverseProperty	Declares the navigation property on the other end of a relationship.
MaxLength	Specifies the max length for a string property.
TimeStamp	Declares a type as a rowversion in SQL Server and adds concurrency checks to database operations.
DatabaseGenerated	Specifies if the field is database generated or not. Takes a <code>DatabaseGeneratedOption</code> value of <code>Computed</code> , <code>Identity</code> , or <code>None</code> .
NotMapped	Specifies that EF needs to ignore this property in regards to database fields.
DataType	Provides for a more specific definition of a field than the intrinsic datatype.

Adding Data Annotations to the EntityBase Class

For primary key fields, the `Key` and `DatabaseGenerated` attributes replace the EF conventions. They don't add any functionality, but are explicit and obvious to another developer. As detailed in the previous table, the `Key` attribute identifies the property as the primary key. The `DatabaseGenerated` attribute with the `DatabaseGeneratedOption` of `Identity` instructs SQL Server to generate the value when new records are added. Update the `Id` property by adding the following code:

```
[Key, DatabaseGenerated(DatabaseGeneratedOption.Identity)]
public int Id { get; set; }
```

The next data annotation to add is the `TimeStamp` attribute. In SQL Server, rowversion fields get updated by the server every time a record is added or saved. The CLR datatype that represents a rowversion is a byte array of length 8. Update the `TimeStamp` property to the following, and the base class is complete:

```
[Timestamp]
public byte[] TimeStamp { get; set; }
```

Adding the Category Model Class

Each of the tables listed in the Figure 1-1 will be represented by a .NET class (the simplest modeling mechanism for EF project), starting with the `Category` table.

■ **Note** This chapter focuses on the `Category` table and works through the entire process of building the `DbContext`, creating and running EF migrations, unit testing, and creating a repository. In the next chapter, the final EF Core concepts are covered, and the remaining tables and repositories are added, completing the data access layer.

In the `Entities` folder, add a new class and name it `Category.cs`. Add the following namespaces to the class:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using SpyStore.Models.Entities.Base;
```

Update the class to inherit from `EntityBase` and add a string property called `CategoryName`, as shown in the following code:

```
namespace SpyStore.Models.Entities
{
    public class Category : EntityBase
    {
        public string CategoryName { get; set; }
    }
}
```

Just like the `EntityBase` class, data annotations will be used to shape the table that gets created by EF from the `Category` class. For the `CategoryName` field, add the `DataType` and `MaxLength` annotations. The `DataType` annotation refines the type beyond the high-level string datatype, and the `MaxLength` annotation sets the size of the text field in SQL Server. The `MaxLength` attribute is also used in Core MVC validations, covered later in this book. Update the code in the `Category.cs` class to the following:

```
public class Category : EntityBase
{
    [DataType(DataType.Text), MaxLength(50)]
    public string CategoryName { get; set; }
}
```

The convention in EF Core for table names is based on the name of the `DbSet<T>` property in the `DbContext` class (covered in the next section). This can be changed (including specifying a database schema) by using the `Table` attribute. Update the code in the `Category.cs` class to the following:

```
[Table("Categories", Schema = "Store")]
public class Category : EntityBase
{
    [DataType(DataType.Text), MaxLength(50)]
    public string CategoryName { get; set; }
}
```

This completes the `Category` class for now. When the additional tables are added to the model, the navigation property for the `Product` class will be added to `Category` class.

Adding the Categories DbSet

The `DbContext` is the core of using EF Core as a data access library. Each model class needs to be added as a `DbSet<T>` property in the context. Add the following to the end of the `StoreContext.cs` class:

```
public DbSet<Category> Categories { get; set; }
```

Any instances of the `Category` class contained in the `DbSet<Category>` collection property can be acted on by the `StoreContext` and EF for CRUD operations.

■ **Note** Technically, any class that is reachable from a class contained in a `DbContext`'s `DbSet<T>` is reachable by EF. This EF convention can cause confusion and code issues, so all of the classes in this data access library are added directory into the `StoreContext` class as `DbSet<T>` properties.

Now that the `Category` class, its base class, and the `StoreContext` are done, it's time to update the database by creating and executing the first migration.

Migrations

When application models are added, deleted, or changed, the physical data store needs to be updated to match the shape of the application data model. Entity Framework migrations provide a convenient mechanism to do just that. In true code-first scenarios (where the database is created from the .NET classes), the process is very straightforward.

1. Create, delete, or update the model classes and/or the context class.
 2. Create a new migration.
 3. Apply that migration to the database.
- or
4. Create the script file for the migration for execution elsewhere (such as SQL Server Management Studio).

■ **Note** Although not covered in this book, you can also use EF Core with existing databases. While still called “code first” (probably not the best name), code first from an existing database works in reverse of the code first steps shown here. Changes are made to an existing database, and then the `DbContext` and model classes are scaffolded from that database. This is a great way to learn the data annotations and EF Fluent API available in EF Core—you build a database in SQL Server Management Studio, and then reverse engineer it into EF code. It is also invaluable if you are in a situation where someone else controls the database your application is using. To find out more, enter `dotnet ef dbcontext scaffold -h` at the command prompt (as shown later in this section) to get help on scaffolding your database.

Each migration builds from the previous migration, and a project can have as many migrations as needed. The more migrations created, the more granular the approach. It is not uncommon to have a series of migrations for each context class, and each team will settle on its preferred workflow.

When a migration is created, EF examines model classes (and their data annotations) that can be reached from the context through the `DbSet<T>` properties, any related classes (and their data annotations), and any Fluent API commands (covered later in this chapter). From all of this information, EF constructs the current shape of the data model. EF then compares this shape with the existing database and creates the necessary change set to migrate the database to the current code-based model. This change set is saved in a migration file that is named using a timestamp and the friendly name provided by the developer. Executing the change set updates the database. This is all detailed in the following sections.

Executing EF .NET CLI Commands

Creating and executing a migration are .NET Core CLI commands added by the EF tooling. The commands can be executed from the Package Manager Console or from the command line. In either case, the command must be run from the directory that contains the `.csproj` file for the project containing the context class and the `Microsoft.EntityFrameworkCore.Tools.DotNet` `DotNetCliToolReference`.

Open Package Manager Console and check the current directory by entering the `dir` command at the prompt. When PMC opens, the default directory is the solution directory. Navigate to the `SpyStore.DAL` directory by entering `cd SpyStore.DAL` at the command prompt.

Once the proper directory is set, run any of the EF commands by typing `dotnet ef` at the prompt followed by the specific command (migrations, for example), including any necessary arguments and options. Help is available for all of the commands by adding `-h` (or `--help`) after the command. For example, to get the list of available top-level EF commands, type:

```
dotnet ef -h
```

The help shows three available commands:

Commands:

```
database    Commands to manage your database
dbcontext   Commands to manage your DbContext types
migrations  Commands to manage your migrations
```

Each successive command also provides help. To get help for adding a migration, enter:

```
dotnet ef migrations add -h
```

Creating the First Migration

When migrations are created, they default to a directory named `Migrations`. To change that, use the `-o` parameter. It's generally a good idea to specify the context as well with the `-c` parameter and the fully qualified name of the context. If one isn't specified, the default context (or the only context EF can find in the current project directory) will be used. In this example project, there is only one context, but real business applications frequently have more than one, and getting in the habit of specifying the context will potentially be required in those larger projects.

To create the migration for the `StoreContext`, enter the following command at the command prompt:

```
dotnet ef migrations add Initial -o EFMigrations -c SpyStore.DAL.EF.StoreContext
```

Executing the first migration creates the EF\Migrations directory (if it doesn't already exist) and adds three files to the directory:

- 20161113015453_Initial.cs (your timestamp value will be different)
- 20161113015453_Initial.Designer.cs (your timestamp value will be different)
- StoreContextModelSnapshot.cs (updated if already exists)

The files describing the changes needed are the two with names containing a timestamp and the migration name. This naming convention enables EF to replay all migrations in the correct order, as each subsequent migration will have another, later timestamp as the first part of the name.

Open the 20161113015453_Initial.cs class. This class has two methods, named Up and Down. The Up method is for applying the changes to the database, and the Down method is to roll back the changes. The file should look like this (notice the class name does not contain the timestamp value):

```
public partial class Initial : Migration
{
    protected override void Up(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.EnsureSchema(name: "Store");

        migrationBuilder.CreateTable(
            name: "Categories",
            schema: "Store",
            columns: table => new
            {
                Id = table.Column<int>(nullable: false)
                    .Annotation("SqlServer:ValueGenerationStrategy",
                        SqlServerValueGenerationStrategy.IdentityColumn),
                CategoryName = table.Column<string>(maxLength: 50, nullable: true),
                TimeStamp = table.Column<byte[]>(rowVersion: true, nullable: true)
            },
            constraints: table =>
            {
                table.PrimaryKey("PK_Categories", x => x.Id);
            });
    }

    protected override void Down(MigrationBuilder migrationBuilder)
    {
        migrationBuilder.DropTable(
            name: "Categories",
            schema: "Store");
    }
}
```

The 20161113015453_Initial.Designer.cs class is the cumulative shape of the database, including this migration and all previous migrations.

The StoreContextModelSnapshot.cs class is the current shape of the database according to all of the migrations.

■ **Note** Previous versions of EF created a hash of the database and stored it in the `__MigrationHistory` table. Conflicts in database shape were usually discovered the hard way with developers stepping on each other. The new approach greatly enhances the team development workflow, since the full database definition is stored as a text file in the project instead of a hash in the database. This enables change detection and merge capabilities using modern version control software.

Removing a Migration

In previous versions of EF, you could remove any specific migration (even if it was in the middle of the migration chain). This option is no longer available. The `dotnet ef migrations remove` command removes the last migration only, so if you need to remove the third out of five total, you need to execute the `remove` command three times. This makes sense, since the `ModelSnapshot` is updated when a migration is removed. Removing a migration in the middle would leave the `ModelSnapshot` in an inconsistent state.

Applying the Migration

By default, migrations are not automatically applied (auto applying migrations is covered later). The `dotnet ef database update` command is used to promote the changes to the data store. If the name of a migration is entered with the command, EF will apply all pending migrations up to and including the named migration. If no name is entered, all pending migrations will be applied. If a 0 is entered instead of a name, all migrations will be rolled back. The context to use can be specified the same way as when creating migrations, and if one isn't specified, the default context (or the only context EF can find in the current project directory) will be used. To apply the `Initial` migration you just created, enter the following command into Package Manager Console:

```
dotnet ef database update Initial -c SpyStore.DAL.EF.StoreContext
```

The `update` command first checks to see if the database exists based on the connection string contained in the context's `DbContextOptions`. If the database doesn't exist, EF will create it. EF then checks for a table named `__EFMigrationsHistory`. If it doesn't exist, it will also be created. This table stores the migration name and EF version for all applied migrations. EF next checks the list of applied migrations against the list of pending migrations included in the command entered and then applies them, in order. When the process is complete, any migrations that were applied are entered into the `__EFMigrationsHistory` table.

After executing the command, you will see all of the SQL commands executed in the PMC (or command window). If the migration was successfully applied, you will see a single word:

```
Done.
```

Somewhat anticlimactic considering all of the work that was completed!

Viewing the Database

To view the created database, open Server Explorer (if it's not visible, select it under the View menu in VS2017). Create a new database connection using the values from your connection string (similar to Figure 1-11).

The screenshot shows the 'Add Connection' dialog box with the following configuration:

- Data source:** Microsoft SQL Server (SqlClient)
- Server name:** (localdb)\mssqllocaldb
- Log on to the server:**
 - Authentication:** Windows Authentication
 - User name:** (empty)
 - Password:** (empty)
 - Save my password
- Connect to a database:**
 - Select or enter a database name: SpyStore
 - Attach a database file: (empty) [Browse...]
 - Logical name: (empty)

Buttons at the bottom: Test Connection, OK, Cancel, and an Advanced... button.

Figure 1-11. Creating the connection for the SpyStore database

Once the connection is created, you can view the Customers table in Server Explorer to confirm that the table was created successfully.

Creating Migration SQL Scripts

If you are in an environment where database updates cannot be applied directly by developers, EF Core has a process for creating SQL scripts for migrations. This is also valuable if you want to check what the migration will do before actually applying it to your database. The start and end migrations can be specified to limit the scope of the script. If neither is specified, the script will start at the first migration and run to the last migration.

To create a SQL script for the Initial migration, enter the following command:

```
dotnet ef migrations script 0 Initial -o migration.sql -c SpyStore.DAL.EF.StoreContext
```

The output is a SQL script that can be run against your database to reflect the changes in your model. Note that the script creates the `__EFMigrationHistory` table if it doesn't exist, but not the database.

Understanding CRUD Operations Using Entity Framework

Now that the framework is in place and the migration has been applied, it's time to put it all together and understand how CRUD operations are executed using Entity Framework. The `DbSet<Category>` property of the `StoreContext` is the gateway into and out of the `Category` table in the database. You will see all of this in action later in this chapter.

Creating Records

Creating a new category record in the database through EF is a three-step process:

1. Create a new instance of the `Category` class and set the properties accordingly.
2. Add it to the `Categories` `DbSet<Category>` collection property of the `StoreContext`.
or
Add it to the `StoreContext`.
3. Call `SaveChanges()` on the `StoreContext`.

When creating a new instance of the `Category` class, the `Id` and `Timestamp` properties should not be populated since they are set by SQL Server.

When a new instance of a class is added to the appropriate `DbSet<T>`, two main things happen. The class is added to the context's `ChangeTracker` and the `EntityState` of the instance is set to `EntityState.Added`. The `ChangeTracker` (as the name suggests) tracks all entities that are added, updated, or set for deletion in the context. You can also add an instance to the context directly. If this method is used, the context determines the type of the instance and adds it to the correct `DbSet<T>`.

The final step is to call `SaveChanges` on the context. When this is called, the context gets all of the changed entities from the `ChangeTracker`, determines the actions needed (based on the entity state), and executes each of the changes against the database in a single transaction. If any of the changes fail, all of them fail as a unit. If they all succeed, the `EntityState` of each instance is changed to `EntityState.Unchanged`, and the `Id` and `Timestamp` properties (as well as any other database generated properties in your model) are populated with the values from the database.

Reading Records

Reading records from the database involves executing a LINQ statement against the `DbSet<T>`. The data read from the database is used to create new instances of the classes (in this case `Category` instances) and adds them to the `DbSet<T>` and, by default, tracks them using the `ChangeTracker`.

No-Tracking Queries

Records can be read for read-only use. This can be more performance, as the `ChangeTracker` doesn't monitor the records. To read records as untracked, add `AsNoTracking()` to the LINQ statement as in the following:

```
context.Categories.AsNoTracking().ToList();
```

For all queries on a context instance to be no-tracking, change the `QueryTrackingBehavior` property of the `ChangeTracker` as follows:

```
context.ChangeTracker.QueryTrackingBehavior=QueryTrackingBehavior.NoTracking;
```

Updating Records

There are two methods to update records in the database. The first is to work with a tracked entity, and the steps are as follows:

1. Read the record to be changed from the database and make any desired edits.
2. Call `Update` on the `Categories DbSet<Category>`.
or
Call `Update` on the `StoreContext`.
3. Call `SaveChanges()` on the `StoreContext`.

Without concurrency checking, if the `CategoryName` has been updated, a query similar to the following is executed to update the record:

```
UPDATE Store.Categories SET CategoryName = "Foo" WHERE Id = 5
```

Concurrency Checking

When there is a `Timestamp` attribute on a property in the class, the where clauses on `Updates` and `Deletes` are updated to include this field. In this case, it's the `Timestamp` field. For example, the update statement from before is updated to this:

```
UPDATE Store.Categories SET [CategoryName] = 'Communications'  
WHERE Id = 5 and Timestamp = 0x0000000000017076
```

The `Timestamp` property in SQL Server is updated every time a record is added or updated. If the record referenced by the `ChangeTracker` was changed in the database by another user or process, the `Timestamp` value in the tracked entity won't match the value in the database. The where clause can't locate the record, and the record is left unchanged. This doesn't cause an exception in SQL Server, it just doesn't affect any records. This causes a difference between the number of expected changes in the `ChangeTracker` and the actual number of changes executed.

When the number of expected changes doesn't match the number of actual changes, EF throws a `DbUpdateConcurrencyException`, and the entire transaction is rolled back. The entities that were not updated or deleted are contained in the `Entries` property of the `DbUpdateConcurrencyException`.

Updating Using Entity State

The second method involves setting the `EntityState` on a model instance directly. The steps are as follows:

1. Make sure the entity is not already being tracked in the context.
2. Create a new instance of the `Category` class.
3. Set all of the fields to the correct values. Ensure that the `Id` and `Timestamp` fields match the values of the record to be updated.
4. Set the `EntityState` to `EntityState.Modified`.
5. Call `SaveChanges()` on the `StoreContext`.

The advantage of this is that if the entity isn't already being tracked, it doesn't have to be read from the database just to be updated. This method is best used for stateless scenarios where all of the properties are available. Note: If the entity is being tracked, this method doesn't work.

Deleting Records

There are two main methods for deleting records from the database. The first method tracks very closely with updating records.

1. Read the record to be changed from the database.
2. Call `Remove` on the `Categories DbSet<Category>`.
or
Call `Remove` on the `StoreContext`.
3. Call `SaveChanges()` on the `StoreContext`.

Similar to when it's updating, EF generates a query similar to the following (and will throw a `DbUpdateConcurrencyException` if nothing is deleted):

```
DELETE FROM Store.Categories WHERE Id = 5 and timestamp = 0x0000000000017076
```

Deleting Using Entity State

The second method involves setting the `EntityState` on a model instance directly. The steps are as follows:

1. Make sure the entity is not already being tracked in the context.
2. Create a new instance of the `Category` class.
3. Set the `Id` and `Timestamp` fields to the values of the record to be deleted.
4. Set the `EntityState` to `EntityState.Deleted`.
5. Call `SaveChanges()` on the `StoreContext`.

Similar to updating using `EntityState`, the advantage of this is that if the entity isn't already being tracked, it doesn't have to be read from the database just to be deleted. This method is commonly used in stateless web scenarios. Note: If the entity is being tracked, this method doesn't work.

Unit Testing EF Core

It's time to add unit tests to ensure that all of the work you have done so far actually does what is expected. There are many unit testing frameworks available for .NET developers, and the choice depends largely on personal preference. My favorite unit testing framework is xUnit (<http://xunit.github.io>), which also supports .NET Core.

The tests that follow are designed to test that the `StoreContext` can perform CRUD operations for the `Categories` table in the database. From an academic point of view, that means they aren't actually unit tests, but *integration* tests. While technically correct, I will simply refer to them as *unit tests* (or just *tests*) throughout this book. One can also argue that there isn't a need to add unit (or integrations) tests just to test that EF does what it is supposed to do. That is correct as well. But what these tests are really doing is validating *our* understanding and implementation of EF, not necessarily testing EF itself. In short, there are a wide range of opinions on testing, and being different doesn't mean being wrong.

■ **Note** Automated testing is an extremely important part of development. Due to space limitations (in an already long set of chapters on EF Core), there are only a few tests for EF covered in print. There are many more tests in the downloadable code, and following the samples shown in this section, I leave the rest of the testing for you, the reader, to implement.

Creating the CategoryTests Class

For the unit tests, I am following the Constructor and Dispose pattern in xUnit as documented here: <http://xunit.github.io/docs/shared-context.html#constructor>, which requires each test class to implement `IDisposable`. This pattern is useful if you need to run code before and/or after every test—code in the class constructor is run before every test, and code in the `Dispose` method is run after every test.

Add a new folder named `ContextTests` to the root of the `SpyStore.DAL.Tests` project, and add a new class into that folder named `CategoryTests.cs`. Add (or update) the `using` statements to include all of the following:

```
using System;
using SpyStore.DAL.EF;
using SpyStore.Models.Entities;
using Xunit;
using Microsoft.EntityFrameworkCore;
using System.Linq;
```

Next, implement `IDisposable` and create a constructor.

```
public class CategoryTests : IDisposable
{
    public CategoryTests()
    {
    }
    public void Dispose()
    {
    }
}
```

The default for xUnit is to run all tests in a class in serial and to test across classes in parallel. Since all of the tests in this project are testing against the same database, running them serially keeps them isolated from all other tests. To accomplish this, add the class level `Collection` attribute and add the text "SpyStore.DAL" into the attribute's constructor. All classes that share the `Collection` attribute and the same key have their tests run in serial. Add the constructor like this:

```
[Collection("SpyStore.DAL")]
public class CategoryTests : IDisposable
{
    //omitted for brevity
}
```

Add a private read-only variable to hold an instance of the `StoreContext` and instantiate it in the constructor. Dispose of it in the `Dispose` method. This causes every test to get a newly instantiated `StoreContext`, and after each test, the `StoreContext` is disposed. Update the code to match the following (note the new using statement):

```
using SpyStore.DAL.EF;
public class CategoryTests : IDisposable
{
    private readonly StoreContext _db;
    public CategoryTests()
    {
        _db = new StoreContext();
    }
    public void Dispose()
    {
        _db.Dispose();
    }
}
```

■ **Note** Creating a new instance of a store context before each use should be done with caution and critical thought. There is a cost associated with creating a new instance of a `DbContext`. For testing purposes, however, it's best to have a clean context for every test, so creating a new one is the best approach.

The final setup code is to make sure any records that were added to the database are cleared out and sequences reseeded. Add the following using statement to the top of the file:

```
using Microsoft.EntityFrameworkCore;
```

Next, create a method called `CleanDatabase` as follows:

```
private void CleanDatabase()
{
    _db.Database.ExecuteSqlCommand("Delete from Store.Categories");
    _db.Database.ExecuteSqlCommand($"DBCC CHECKIDENT (\\"Store.Categories\\", RESEED, -1);");
}
```

Add a call to this method in the constructor and the `Dispose` method. Calling in the `Dispose` executes after each test. The method is added into the constructor in case another test fails and leaves the database in an inconsistent state, as follows:

```
public CategoryTests()
{
    _db = new StoreContext();
    CleanDatabase();
}
public void Dispose()
{
    CleanDatabase();
    _db.Dispose();
}
```

Creating and Running the First Test

xUnit uses the `[Fact]` attribute to mark tests. This is analogous to the `[Test]` attribute used in other test frameworks (there is also a `[Theory]` attribute for data driven tests, but that isn't covered in this chapter). The first test will simply demonstrate xUnit and running tests in Visual Studio. Add the following code to the `CategoryTests.cs` class:

```
[Fact]
public void FirstTest()
{
    Assert.True(true);
}
```

To execute the test, open the Test Explorer window by selecting `Test > Windows > Test Explorer`. When you rebuild the solution, unit tests in your solution will show up in the Test Explorer window, as shown in Figure 1-12.

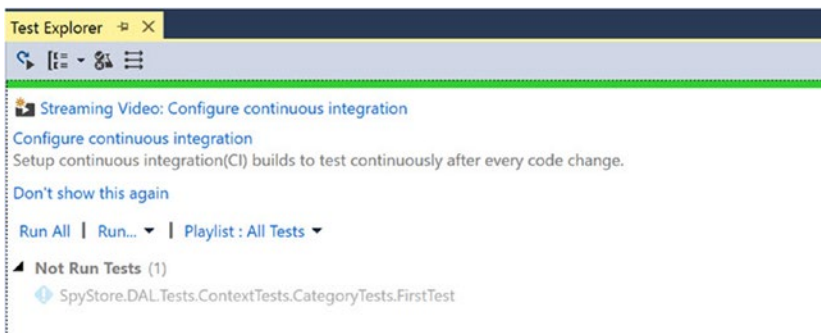


Figure 1-12. The Test Explorer window

To run the tests, click on `Run All` or right-click the test name and select `Run Selected Tests`. If all goes well, the test will go green. If not, you can double-click on the test name, and Visual Studio will take you directly to the test code.

Testing EF CRUD Operations

The next step is to add tests to demonstrate the CRUD functionality of EF. The following sections put into action the concepts discussed in the previous section. Start by adding the following using statement to the `CategoryTests.cs` file:

```
using SpyStore.Models.Entities;
```

Test Adding a Category Record

This test covers adding a new record into the database using the `DbSet<T>` property on the `StoreContext`. Add the following code to the `CategoryTests.cs` class:

```
[Fact]
public void ShouldAddACategoryWithDbSet()
{
    var category = new Category { CategoryName = "Foo" };
    _db.Categories.Add(category);
    Assert.Equal(EntityState.Added, _db.Entry(category).State);
    Assert.True(category.Id < 0);
    Assert.Null(category.TimeStamp);
    _db.SaveChanges();
    Assert.Equal(EntityState.Unchanged, _db.Entry(category).State);
    Assert.Equal(0, category.Id);
    Assert.NotNull(category.TimeStamp);
    Assert.Equal(1, _db.Categories.Count());
}
```

The first part of the test creates a new `Category` instance and then adds it to the `DbSet<Category>` collection through the `Add()` method. The next line confirms that the `EntityState` is set to `EntityState.Added`. The `Id` gets a temporary value (less than zero) since it is assigned from the database. The temporary value for the `Id` depends on how many have been added and not yet persisted (each entity gets a unique negative value). The `TimeStamp` property is null since it hasn't been populated.

As discussed earlier, calling `SaveChanges()` on the `StoreContext` executes the add SQL statement(s), and if successful, sets the `EntityState` for all processed records to `EntityState.Unchanged`, and the database generated fields are populated. The next three asserts verify this. The final assert verifies that there is one record in the `Categories` table.

■ **Note** The avid tester will probably notice that I am not following one of the core philosophies in unit testing in that I essentially have two actions (`Add` and `SaveChanges`) mixed in with a series of asserts. This is typically not a good idea, but the purpose of the tests covered in this chapter are for teaching EF Core, not for production development.

For the next test, instead of adding the new record through the `DbSet<Category>` property, add the new record directly through the `StoreContext` class. That is literally the only change. Copy the last test into a new one named `ShouldAddCategoryWithContext` and change the second line to the following:

```
_db.Add(category);
```

The entire test is shown here:

```
[Fact]
public void ShouldAddACategoryWithContext()
{
    var category = new Category { CategoryName = "Foo" };
    _db.Add(category);
    Assert.Equal(EntityState.Added, _db.Entry(category).State);
    Assert.True(category.Id < 0);
    Assert.Null(category.TimeStamp);
    _db.SaveChanges();
    Assert.Equal(EntityState.Unchanged, _db.Entry(category).State);
    Assert.Equal(0, category.Id);
    Assert.NotNull(category.TimeStamp);
    Assert.Equal(1, _db.Categories.Count());
}
```

Run the test and verify that it works as well. For the rest of the tests, you will only be using the `DbSet<T>` property and not replicating tests just to show how to call the CRUD methods on the context class.

Test Retrieving All Category Records

This test is a simple example showing the equivalent of this SQL statement:

```
SELECT * FROM STORE.Categories Order By CategoryName
```

Add a new Fact with the following code:

```
[Fact]
public void ShouldGetAllCategoriesOrderedByName()
{
    _db.Categories.Add(new Category { CategoryName = "Foo" });
    _db.Categories.Add(new Category { CategoryName = "Bar" });
    _db.SaveChanges();
    var categories = _db.Categories.OrderBy(c=>c.CategoryName).ToList();
    Assert.Equal(2, _db.Categories.Count());
    Assert.Equal("Bar", categories[0].CategoryName);
    Assert.Equal("Foo", categories[1].CategoryName);
}
```

The Fact starts off by adding two `Category` records into the database. LINQ statements in EF start from the `DbSet<T>`, and in this example, there isn't a `where` clause but there is an `OrderBy`. The actual query isn't executed until the list is iterated over or `ToList` is called. In this test, when `ToList` is called, the query is executed and the data is retrieved. The asserts check to make sure that the `Category` records were indeed retrieved and in the correct order.

Test Updating a Category Record

This test demonstrates modifying a record in the database. Add the following code into your test class:

```
[Fact]
public void ShouldUpdateACategory()
{
    var category = new Category { CategoryName = "Foo" };
    _db.Categories.Add(category);
    _db.SaveChanges();
    category.CategoryName = "Bar";
    _db.Categories.Update(category);
    Assert.Equal(EntityState.Modified, _db.Entry(category).State);
    _db.SaveChanges();
    Assert.Equal(EntityState.Unchanged, _db.Entry(category).State);
    StoreContext context;
    using (context = new StoreContext())
    {
        Assert.Equal("Bar", context.Categories.First().CategoryName);
    }
}
```

After creating and saving a new `Category` record, the instance is updated. The test confirms that the `EntityState` is set to `EntityState.Modified`. After `SaveChanges` is called, the test confirms that the state is returned to `EntityState.Unchanged`. The final part of the test creates a new `StoreContext` instance (to avoid any side effects due to caching) and confirms that the `CategoryName` was changed. The assert uses the `First` LINQ method, which is equivalent to adding `Top 1` to a SQL query. The `First` LIN method also executes the query immediately, so there isn't a need to iterate over the results or call `ToList`. This makes total sense, since there isn't a list!

It's important to note that `Update()` only works for persisted records that have been changed. If you try to call `Update()` on an entity that is new, EF will throw an `InvalidOperationException`. To show this, add another `Fact` that contains the following code:

```
[Fact]
public void ShouldNotUpdateANonAttachedCategory()
{
    var category = new Category { CategoryName = "Foo" };
    _db.Categories.Add(category);
    category.CategoryName = "Bar";
    Assert.Throws<InvalidOperationException>(() => _db.Categories.Update(category));
}
```

This uses the xUnit `Assert` that confirms an exception of the specified type was thrown. If the exception is thrown, the test passes. If it doesn't get thrown, the test fails.

Test Deleting a Category Record Using Remove

Create a new Fact in your test class and enter the following code:

```
[Fact]
public void ShouldDeleteACategory()
{
    var category = new Category { CategoryName = "Foo" };
    _db.Categories.Add(category);
    _db.SaveChanges();
    Assert.Equal(1, _db.Categories.Count());
    _db.Categories.Remove(category);
    Assert.Equal(EntityState.Deleted, _db.Entry(category).State);
    _db.SaveChanges();
    Assert.Equal(EntityState.Detached, _db.Entry(category).State);
    Assert.Equal(0, _db.Categories.Count());
}
```

This test adds a record to the `Categories` table and asserts that it was saved. Next, `Remove` is called on the `DbSet<T>`, and then the deletion is executed by calling `SaveChanges`. Note: `Remove` only works on tracked entities. If the instance was not being tracked, it would have to be loaded from the database into the `ChangeTracker` to be deleted, or you can delete it using `EntityState` (shown in the next section). The final asserts verify that the record was deleted.

Test Deleting a Record Using EntityState

To show how to use `EntityState` to delete a record, create a new Fact and enter the following code:

```
[Fact]
public void ShouldDeleteACategoryWithTimestampData()
{
    var category = new Category { CategoryName = "Foo" };
    _db.Categories.Add(category);
    _db.SaveChanges();
    var context = new StoreContext();
    var catToDelete = new Category { Id = category.Id, TimeStamp = category.TimeStamp };
    context.Entry(catToDelete).State = EntityState.Deleted;
    var affected = context.SaveChanges();
    Assert.Equal(1, affected);
}
```

The test begins by saving a new `Category` record to the database. The test then creates a new `StoreContext` to avoid any caching issues, creates a new instance of a `Category`, and assigns the `Id` and `TimeStamp` values from the saved record. The entity has the `EntityState` set to `EntityState.Deleted` and `SaveChanges` is called to persist the change. The final assert verifies that there was one affected record.

Testing Concurrency Checking

To show how EF uses the `TimeStamp`, create a new Fact and enter the following code:

```
[Fact]
public void ShouldNotDeleteACategoryWithoutTimestampData()
{
    var category = new Category { CategoryName = "Foo" };
    _db.Categories.Add(category);
    _db.SaveChanges();
    var context = new StoreContext();
    var catToDelete = new Category { Id = category.Id };
    context.Categories.Remove(catToDelete);
    var ex = Assert.Throws<DbUpdateConcurrencyException>(() => context.SaveChanges());
    Assert.Equal(1, ex.Entries.Count);
    Assert.Equal(category.Id, ((Category)ex.Entries[0].Entity).Id);
}
```

The test is almost exactly the same as the previous test, except that the `TimeStamp` value isn't assigned to the new `Category` instance. When `SaveChanges` is called, EF throws a `DbUpdateConcurrencyException`. The exception exposes the entities in error through the `Entries` property, allowing for properly handling concurrency issues based on application needs.

Adding the Core Repository Interface and Base Class

A common data access design pattern is the *repository pattern*. As described by Martin Fowler (<http://www.martinfowler.com/eaCatalog/repository.html>), the core of this pattern is to mediate between the domain and data mapping layers. This helps to eliminate duplication of code. Having specific repositories and interfaces will also become useful when you implement ASP.NET Core MVC in Chapters 3 and 5.

■ **Note** The code shown in this section is not meant to be a model for the official implementation of the pattern. It is my interpretation of the pattern mixed with years of using EF in production.

Adding the IRepo Interface

My version of the repository pattern starts with a core repository for the common methods that will be used on all derived repositories. This core interface (called `IRepo`) will be supported by an abstract class that encapsulates the `DBSet<T>` and `DbContext` methods, plus some additional convenience methods.

To create the `IRepo` interface, start by adding a new folder to the `SpyStore.DAL` project named `Repos`, then add a new folder named `Base` under the new `Repos` folder. Right-click on the `Base` folder and select `Add ► New Item`. Select the `Interface` template, as shown in Figure 1-13.

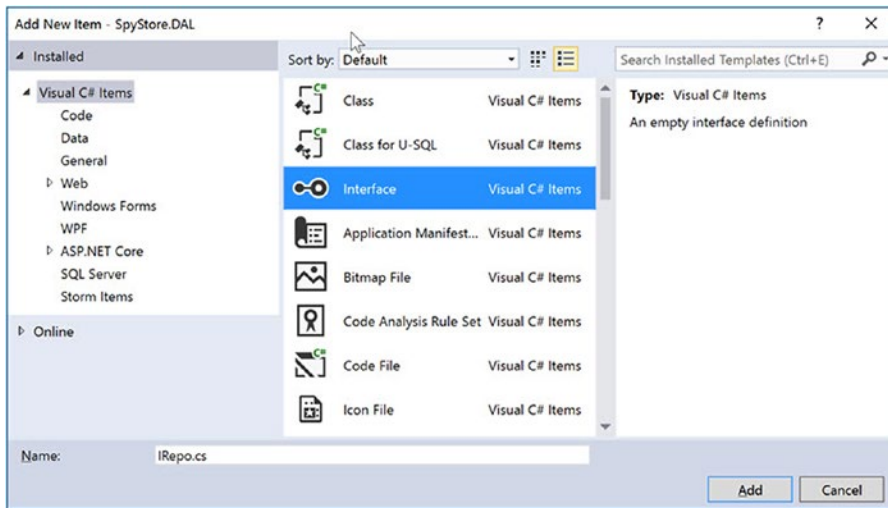


Figure 1-13. Adding the *IRepo* interface

In the new interface file, add the following namespaces:

```
using System.Collections.Generic;
using SpyStore.Models.Entities.Base;
```

Update the *IRepo.cs* interface to a generic interface, limiting the type of the parameter to classes derived from *EntityBase*, like this:

```
public interface IRepo<T> where T : EntityBase
```

The following methods should (mostly) look familiar from Tables 1-3 and 1-4. Update the *IRepo.cs* interface to the following code:

```
public interface IRepo<T> where T : EntityBase
{
    int Count { get; }
    bool HasChanges { get; }
    T Find(int? id);
    T GetFirst();
    IEnumerable<T> GetAll();
    IEnumerable<T> GetRange(int skip,int take);
    int Add(T entity, bool persist = true);
    int AddRange(IEnumerable<T> entities, bool persist = true);
    int Update(T entity, bool persist = true);
    int UpdateRange(IEnumerable<T> entities, bool persist = true);
    int Delete(T entity, bool persist = true);
    int DeleteRange(IEnumerable<T> entities, bool persist = true);
    int Delete(int id, byte[] timeStamp, bool persist = true);
    int SaveChanges();
}
```

All of the interfaces methods will be detailed when they are implemented in the base repository class.

Adding the Base Repository

Next, add a class to the `Repos\Base` directory named `RepoBase`. This class will implement the common functionality defined in the interface. Update the using statements in the `RepoBase` class to the following:

```
using System;
using System.Collections.Generic;
using Microsoft.EntityFrameworkCore;
using SpyStore.DAL.EF;
using SpyStore.Models.Entities.Base;
using System.Linq;
using Microsoft.EntityFrameworkCore.ChangeTracking;
using Microsoft.EntityFrameworkCore.Storage;
```

The base repo needs to implement `IDisposable` and `IRepo<T>`, again limiting the type of `T` to `EntityBase`.

```
public abstract class RepoBase<T> : IDisposable, IRepo<T> where T : EntityBase, new()
```

Create a protected variable for the `StoreContext`, instantiate it in the constructors, and dispose of it in the `Dispose` method. Update the code to the following:

```
public abstract class RepoBase<T> : IDisposable, IRepo<T> where T : EntityBase, new()
{
    protected readonly StoreContext Db;
    protected RepoBase()
    {
        Db = new StoreContext();
        Table = Db.Set<T>();
    }
    protected RepoBase(DbContextOptions<StoreContext> options)
    {
        Db = new StoreContext(options);
        Table = Db.Set<T>();
    }
    bool _disposed = false;
    public void Dispose()
    {
        Dispose(true);
        GC.SuppressFinalize(this);
    }
    protected virtual void Dispose(bool disposing)
    {
        if (_disposed) return;
        if (disposing)
        {
            // Free any other managed objects here.
            //
        }
        Db.Dispose();
        _disposed = true;
    }
}
```

The second constructor takes a `DbContextOptions` instance to support dependency injection. This will be used by ASP.NET Core MVC in Chapter 3. Next, add a property to hold the strongly typed `DbSet<T>` and another to expose the `StoreContext` to outside classes. The `Table` property is a convenience property to clean up the code needing the specific `DbSet<T>` property on the `StoreContext`. Add the following code:

```
protected DbSet<T> Table;
public StoreContext Context => Db;
```

With the `StoreContext` set, use the `Set<T>` function to assign the correct `DbSet<T>` to the `Table` variable. Update both constructors to the following code:

```
protected RepoBase()
{
    Db = new StoreContext();
    Table = Db.Set<T>();
}
protected RepoBase(DbContextOptions<StoreContext> options)
{
    Db = new StoreContext(options);
    Table = Db.Set<T>();
}
```

A significant advantage to encapsulating the `DbSet<T>` and `DbContext` operations in a repository class is wrapping the call to `SaveChanges`. This enables centralized error handling of calls to make changes to the database. Add the following code to the `RepoBase` class:

```
public int SaveChanges()
{
    try
    {
        return Db.SaveChanges();
    }
    catch (DbUpdateConcurrencyException ex)
    {
        //A concurrency error occurred
        //Should handle intelligently
        Console.WriteLine(ex);
        throw;
    }
    catch (RetryLimitExceededException ex)
    {
        //DbResiliency retry limit exceeded
        //Should handle intelligently
        Console.WriteLine(ex);
        throw;
    }
    catch (Exception ex)
    {
        //Should handle intelligently
        Console.WriteLine(ex);
        throw;
    }
}
```

As discussed previously, a `DbUpdateConcurrencyException` is thrown when the number of records updated in the database doesn't match the number of records the `ChangeTracker` expects to be updated. The `RetryLimitExceededException` is thrown when the max retries in the `ConnectionString` is exceeded. The final exception handler is a generic catch-all exception handler. Note that none of the handlers actually do any error handling. They are just here to demonstrate catching specific errors.

The next block of code to be added wraps the matching `Create`, `Update`, and `Delete` calls on the specific `DbSet<T>` property. The default implementation immediately calls `SaveChanges`, but there is an option to not call `SaveChanges` immediately in order to batch the updates to the database in a single transaction. All of the methods are marked `virtual` to allow for downstream overriding. Add the following code:

```
public virtual int Add(T entity, bool persist = true)
{
    Table.Add(entity);
    return persist ? SaveChanges() : 0;
}
public virtual int AddRange(IEnumerable<T> entities, bool persist = true)
{
    Table.AddRange(entities);
    return persist ? SaveChanges() : 0;
}
public virtual int Update(T entity, bool persist = true)
{
    Table.Update(entity);
    return persist ? SaveChanges() : 0;
}
public virtual int UpdateRange(IEnumerable<T> entities, bool persist = true)
{
    Table.UpdateRange(entities);
    return persist ? SaveChanges() : 0;
}
public virtual int Delete(T entity, bool persist = true)
{
    Table.Remove(entity);
    return persist ? SaveChanges() : 0;
}
public virtual int DeleteRange(IEnumerable<T> entities, bool persist = true)
{
    Table.RemoveRange(entities);
    return persist ? SaveChanges() : 0;
}
```

As explained earlier, this code is deleting a record using the `Id` and `TimeStamp` properties for an entity that is tracked by the `ChangeTracker`. Add the following method to find if the entity to be deleted is being tracked, and if so, return it:

```
internal T GetEntryFromChangeTracker(int? id)
{
    return Db.ChangeTracker.Entries<T>().Select((EntityEntry e) => (T)e.Entity)
        .FirstOrDefault(x => x.Id == id);
}
```

The Delete method first uses that method to check if the entity is being tracked. If it is, and the TimeStamps match, the entity is removed. If the entity is being tracked and the TimeStamps don't match, an exception is thrown. If the entity isn't being tracked, a new entity is created and tracked with the EntityState of Deleted.

```
public int Delete(int id, byte[] timeStamp, bool persist = true)
{
    var entry = GetEntryFromChangeTracker(id);
    if (entry != null)
    {
        if (entry.TimeStamp == timeStamp)
        {
            return Delete(entry, persist);
        }
        throw new Exception("Unable to delete due to concurrency violation.");
    }
    Db.Entry(new T { Id = id, TimeStamp = timeStamp }).State = EntityState.Deleted;
    return persist ? SaveChanges() : 0;
}
```

The next series of methods return records using LINQ statements. The Find method searches the ChangeTracker first. If the entity being sought is found, that instance gets returned. If not, the record is retrieved from the database.

```
public T Find(int? id) => Table.Find(id);
```

The GetFirst and GetAll methods are very straightforward. Both use the default sort order for the table and are listed here:

```
public T GetFirst() => Table.FirstOrDefault();
public virtual IEnumerable<T> GetAll() => Table;
```

The GetRange method is used for chunking. The base public implementation uses the default sort order. The internal method takes an IQueryable<T> to allow downstream implementations to change the sort order or filters prior to the chunking. This will be demonstrated in the next section.

```
internal IEnumerable<T> GetRange(IQueryable<T> query, int skip, int take)
=> query.Skip(skip).Take(take);
public virtual IEnumerable<T> GetRange(int skip, int take)
=> GetRange(Table, skip, take);
```

The following two properties are added as a developer convenience:

```
public bool HasChanges => Db.ChangeTracker.HasChanges();
public int Count => Table.Count();
```

Adding the Category Repository

The next step is to add the Category specific implementation of RepoBase. Add a new class to the Repos directory and name it CategoryRepo.cs. Inherit BaseRepo<Category> and make sure the using statements match the following:

```
using System.Collections.Generic;
using System.Linq;
using Microsoft.EntityFrameworkCore;
using SpyStore.DAL.EF;
using SpyStore.DAL.Repos.Base;
using SpyStore.Models.Entities;
```

Add the two constructors to match the base constructors, as follows:

```
namespace SpyStore.DAL.Repos
{
    public class CategoryRepo : RepoBase<Category>
    {
        public CategoryRepo(DbContextOptions<StoreContext> options)
            : base(options)
        {
        }
        public CategoryRepo()
        {
        }
    }
}
```

The `GetAll()` and `GetRange()` methods override the base method to sort by `CategoryName`.

```
public override IEnumerable<Category> GetAll()
    => Table.OrderBy(x => x.CategoryName);
public override IEnumerable<Category> GetRange(int skip, int take)
    => GetRange(Table.OrderBy(x => x.CategoryName), skip, take);
```

This repository is essentially done, except for a small bit of additional functionality involving the `Product` class that will be added in the next class. The class is very small, and adding repositories will be as trivial as creating the `CategoryRepo`, since most of the heavy lifting is being done by the `RepoBase<T>` class. This code reuse demonstrates another advantage of using repositories.

■ **Source Code** The `SpyStore.DAL` solution with additional unit tests can be found in the `Chapter 01` subdirectory of the download files.

Summary

In this chapter, you learned about Entity Framework (EF) and Object Relational Mappers, and how they can help the development of an application. You learned about .NET Core and Entity Framework Core (EF Core), the new cross-platform versions of the popular .NET and EF frameworks. The opening section was closed out with a review of the database used by the data access layer.

The next section began with installing Visual Studio 2017 and .NET Core, then created the solution and projects. Next, you used NuGet to update the default packages and add the additional packages, both for EF Core and xUnit.

After getting the tooling installed and the projects created, the chapter walked you through implementing everything needed to use the Category model in your project. This included adding the `EntityBase` and `Category` model classes, configuring them with data annotations, and adding the `StoreContext` instance of the `DbContext`. Finally, you created a migration and executed it to create the database and the database tables.

The next section discussed the theory behind EF Core and CRUD operations. Then, unit tests were added to confirm not only that everything works as expected, but also that our understanding of EF Core is accurate.

The final section built the base repository, encapsulating a significant portion of the CRUD code and wrapping `SaveChanges` for centralized error handling. Finally, the `CategoryRepo` was added, overriding the `GetAll` and `GetRange` methods to retrieve the records ordered by the `CategoryName` field.

The next chapter will complete the entire object model, add the other repositories, and create the tooling to seed the data with sample data for testing.

CHAPTER 2

Building the Data Access Layer with Entity Framework Core

The previous chapter introduced Entity Framework Core and Visual Studio 2017 and built an end-to-end data access layer for the Categories table. This chapter completes the data access layer and covers the final EF concepts used in this book—navigation properties, stored procedures, and user defined functions.

The SpyStore Database

As a refresher, the full SpyStore database is shown in Figure 2-1.

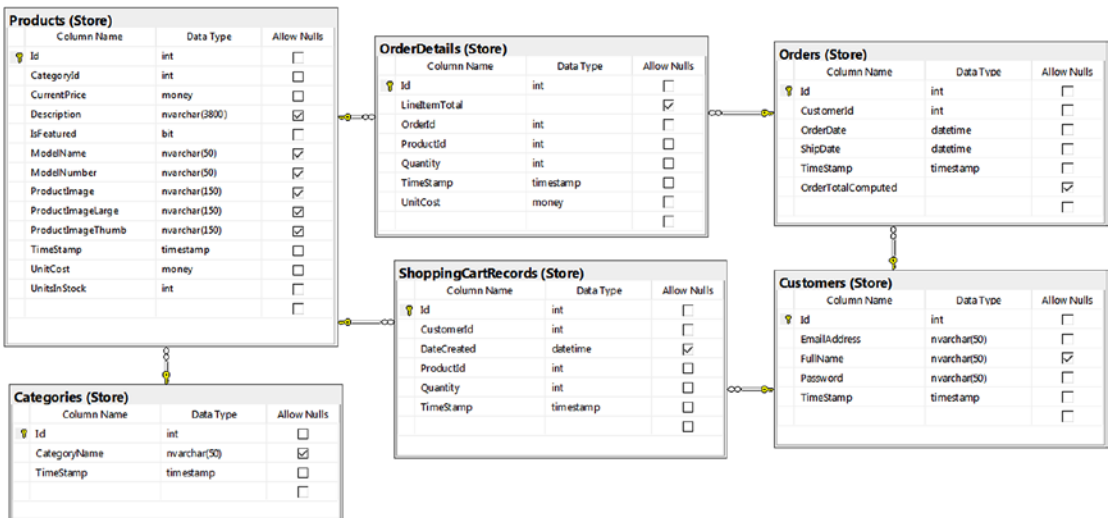


Figure 2-1. The SpyStore database

Navigation Properties and Foreign Keys

As the name suggests, *navigation properties* represent how model classes relate to each other and allow traversing the object model. On the database side, navigation properties are translated into foreign key relationships between tables. The most common navigation properties represent one-to-many relationships. On the one side, a class adds a `List<T>` of the child. On the many side, the class adds a property of the parent type. For example, the `Blogs` table has a one-to-many relationship to the `Posts` table. This would be represented in the `Blog` class as follows:

```
public class Blog
{
    //omitted for brevity
    public int BlogId {get; set;}
    public List<Post> Posts {get; set;}
}

public class Post
{
    //omitted for brevity
    public int BlogId {get;set;}
    public Blog Blog {get; set;}
}
```

By convention, the EF will look for a property on the child class (`Post`) with a name matching the `<NavigationPropertyName>Id` (plus some other formats—see the following note). EF also looks for the primary key on the parent class (`Blog`) that matches the name. If everything lines up, the property on the child becomes the foreign key to the parent class. If the foreign key property is not a nullable type, the relationship is set to delete the child when the parent is deleted (cascade delete). If it is a nullable (`int?` for example), the child will not be deleted when the parent is deleted. EF sets up the `Blog` table as the parent, due to the `List<Post>` property.

■ **Note** For more information on build relationships (including one-one and many-many), and the full details of the conventions, refer to the help documentation located at the following URL: <https://docs.microsoft.com/en-us/ef/core/modeling/relationships>.

Relationships can be defined explicitly with data annotations as well as EF's Fluent API. The `ForeignKey` annotation identifies the property on the child that is the foreign key for the navigation property to the parent. The `InverseProperty` annotation identifies the property on the child that navigates back to the parent. In the `Blog` and `Post` classes, neither of these annotations are required, as the EF conventions already define the relationships. One advantage to using them is that the names can be arbitrary—for example, the foreign key in the `Post` table could be named `FooId`. They also become required when the conventions fall down in complex cases.

This book explicitly uses these attributes on every table for clarity, just as the `Key` annotation was used on the `EntityBase` class. Refactoring the `Blog` and `Post` tables with the annotations would look like this (note the use of the `nameof` function to avoid magic strings):

```
public class Blog
{
    //omitted for brevity
    public int BlogId {get; set;}
}
```

```

    [InverseProperty(nameof(Post.Blog))]
    public List<Post> Posts {get; set;}
}

public class Post
{
    //omitted for brevity
    public int BlogId {get;set;}
    [ForeignKey(nameof(BlogId))]
    public Blog Blog {get; set;}
}

```

Handling Display Names

Although not used by EF, the `Display` attribute is used by ASP.NET Core MVC in display templates to set friendly names for fields. For example, the `CategoryName` property could be set to have the friendly name of “Category”. There are several differing camps on where they should be applied. One way is to set them on the base models themselves. This has the advantage of being defined once throughout the entire system. Detractors feel that this is mixing concerns between the data layer and the user interface layer. Supporters feel that it centralizes all of the information about the model. The other option is to create view models for the UI and apply the attributes there. I leave it to the reader to determine the best option for your project. In this sample code, the display names will be set in the data access layer.

For example, to set the display name for the `CategoryName` property, update the property like this:

```

[Display(Name = "Category")]
public string CategoryName {get; set; }

```

Mixing EF with Stored Procedures and Functions

The purchase process for a customer’s cart involves adding a new record into the `Orders` table, moving records from the `ShoppingCartRecords` table into the `OrderDetails` table, and wrapping it all up into a transaction to make sure all of the changes succeed or fail as a unit.

Additionally, the `Orders` table should have a computed column that sums up the total for all of the `OrderDetail` records. Computed columns in SQL Server can be based on a user defined function to do just this. Unfortunately, there isn’t any way with data annotation or the Fluent API to do this.

Set operations like these are not the best use of EF. Code can be written to create the inserts and deletions for the order process, and LINQ statements can be created to sum up the total cost of an order, but EF was designed and optimized for CRUD operations. There are many application scenarios that are better served by using stored procedures and user defined functions, and these are just two examples.

SQL Server Management Studio is a great tool for coding stored procedures and user defined functions, but how does that code get incorporated into the existing .NET Core projects? A manual workflow needs to be executed so all developers on the team get the T-SQL code installed. This can cause friction, and should be an automatic process, not only for getting the original code, but also for getting any updates. Fortunately, custom T-SQL statements can be executed in a migration the same way that EF created DDL. You will add the stored procedure and function later in this chapter.

Finishing the Model Classes

The vast majority of the remaining tables and fields follow the same pattern as the `Category` class in the last chapter, so there shouldn't be any surprises, just a fair amount of code. Where there is a variation it will be explained, but for the most part the next sections just show all of the code necessary to finish the data models. Because they all interact through navigation properties, the project won't compile until all of the classes have been entered.

Updating the Category Model

The earlier `Category` example was very simplistic for teaching purposes. Now it's time to add the `Product` navigation property into the class. Update the `Category.cs` class to the following code:

```
using System.Collections.Generic;
using SpyStore.Models.Entities.Base;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;

namespace SpyStore.Models.Entities
{
    [Table("Categories", Schema = "Store")]
    public class Category : EntityBase
    {
        [DataType(DataType.Text), MaxLength(50)]
        public string CategoryName { get; set; }
        [InverseProperty(nameof(Product.Category))]
        public List<Product> Products { get; set; } = new List<Product>();
    }
}
```

Adding the Product Model

Create a new class in the `SpyStore.Models` project in the `Entities` folder named `Product.cs`. Update the code to the following:

```
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using SpyStore.Models.Entities.Base;

namespace SpyStore.Models.Entities
{
    [Table("Products", Schema = "Store")]
    public class Product : EntityBase
    {
        [MaxLength(3800)]
        public string Description { get; set; }
        [MaxLength(50)]
        public string ModelName { get; set; }
    }
}
```

```

public bool IsFeatured { get; set; }
[MaxLength(50)]
public string ModelNumber { get; set; }
[MaxLength(150)]
public string ProductImage { get; set; }
[MaxLength(150)]
public string ProductImageLarge { get; set; }
[MaxLength(150)]
public string ProductImageThumb { get; set; }
[DataType(DataType.Currency)]
public decimal UnitCost { get; set; }
[DataType(DataType.Currency)]
public decimal CurrentPrice { get; set; }
public int UnitsInStock { get; set; }
[Required]
public int CategoryId { get; set; }
[ForeignKey(nameof(CategoryId))]
public Category Category { get; set; }
[InverseProperty(nameof(ShoppingCartRecord.Product))]
public List<ShoppingCartRecord> ShoppingCartRecords { get; set; }
    = new List<ShoppingCartRecord>();
[InverseProperty(nameof(OrderDetail.Product))]
public List<OrderDetail> OrderDetails { get; set; } = new List<OrderDetail>();
}
}

```

Adding the Shopping Cart Record Model

Create a new class in the `SpyStore.Models` project in the `Entities` folder named `ShoppingCartRecord.cs`. Update the code to the following:

```

using System;
using System.ComponentModel;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using SpyStore.Models.Entities.Base;

namespace SpyStore.Models.Entities
{
    [Table("ShoppingCartRecords", Schema = "Store")]
    public class ShoppingCartRecord : EntityBase
    {
        [DataType(DataType.Date)]
        public DateTime? DateCreated { get; set; }
        public int CustomerId { get; set; }
        [ForeignKey(nameof(CustomerId))]
        public Customer Customer { get; set; }
        public int Quantity { get; set; }
        [NotMapped, DataType(DataType.Currency)]
        public decimal LineItemTotal { get; set; }
        public int ProductId { get; set; }
    }
}

```

```

    [ForeignKey(nameof(ProductId))]
    public Product Product { get; set; }
}
}

```

Adding the Order Model

Create a new class in the `SpyStore.Models` project in the `Entities` folder named `Order.cs`. Update the code to the following:

```

using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using SpyStore.Models.Entities.Base;

namespace SpyStore.Models.Entities
{
    [Table("Orders", Schema = "Store")]
    public class Order : EntityBase
    {
        public int CustomerId { get; set; }
        [DataType(DataType.Date)]
        [Display(Name = "Date Ordered")]
        public DateTime OrderDate { get; set; }
        [DataType(DataType.Date)]
        [Display(Name = "Date Shipped")]
        public DateTime ShipDate { get; set; }
        [ForeignKey("CustomerId")]
        public Customer Customer { get; set; }
        [InverseProperty("Order")]
        public List<OrderDetail> OrderDetails { get; set; } = new List<OrderDetail>();
    }
}

```

An additional calculated field will be added for the `OrderTotal` later in this chapter.

Adding the Order Detail Model

Create a new class in the `SpyStore.Models` project in the `Entities` folder named `OrderDetail.cs`. Update the code to the following:

```

using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using SpyStore.Models.Entities.Base;

namespace SpyStore.Models.Entities
{
    [Table("OrderDetails", Schema = "Store")]
    public class OrderDetail : EntityBase

```

```

{
    [Required]
    public int OrderId { get; set; }
    [Required]
    public int ProductId { get; set; }
    [Required]
    public int Quantity { get; set; }
    [Required, DataType(DataType.Currency), DisplayName(Name = "Unit Cost")]
    public decimal UnitCost { get; set; }
    [DisplayName(Name = "Total")]
    public decimal? LineItemTotal { get; set; }
    [ForeignKey(nameof(OrderId))]
    public Order Order { get; set; }
    [ForeignKey(nameof(ProductId))]
    public Product Product { get; set; }
}
}

```

The `LineItemTotal` field will be created as a calculated field using the Fluent API (discussed later in this chapter). The `DatabaseGenerated(DatabaseGeneratedOption.Computed)` attribute could have been added to indicate to EF that the field is computed by the database, but the Fluent API will take care of this.

Adding the Customer Model

Create a new class in the `SpyStore.Models` project in the `Entities` folder named `Customer.cs`. Update the code to the following:

```

using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using SpyStore.Models.Entities.Base;

namespace SpyStore.Models.Entities
{
    [Table("Customers", Schema = "Store")]
    public class Customer : EntityBase
    {
        [DataType(DataType.Text), MaxLength(50), DisplayName(Name = "Full Name")]
        public string FullName { get; set; }
        [Required]
        [EmailAddress]
        [DataType(DataType.EmailAddress), MaxLength(50), DisplayName(Name = "Email Address")]
        public string EmailAddress { get; set; }
        [Required]
        [DataType(DataType.Password), MaxLength(50)]
        public string Password { get; set; }
        [InverseProperty(nameof(Order.Customer))]
        public List<Order> Orders { get; set; } = new List<Order>();
    }
}

```

```

    [InverseProperty(nameof(ShoppingCartRecord.Customer))]
    public virtual List<ShoppingCartRecord> ShoppingCartRecords { get; set; }
        = new List<ShoppingCartRecord>();
    }
}

```

The `EmailAddress` attribute is used for validation by ASP.NET Core MVC and ignored by SQL Server.

Updating the StoreContext

There are two steps left before the data model is completed. The first is to add the new model classes as `DbSet<T>` properties in the `StoreContext`. The second is to finish design of the models using the Fluent API.

Adding the `DbSet<T>` Properties for the Models

Open `StoreContext.cs` in the `SpyStore.DAL` project and add a `DbSet<T>` for each of the models, as follows (including the existing line for the `Categories` `DbSet<Category>`):

```

public DbSet<Category> Categories { get; set; }
public DbSet<Customer> Customers { get; set; }
public DbSet<OrderDetail> OrderDetails { get; set; }
public DbSet<Order> Orders { get; set; }
public DbSet<Product> Products { get; set; }
public DbSet<ShoppingCartRecord> ShoppingCartRecords { get; set; }

```

Finishing the Model with the Fluent API

The Fluent API is another mechanism to shape your database. Fluent API code is placed in the override of the `OnModelCreating()` method in your `DbContext` class. To get started, open `StoreContext.cs` in the `SpyStore.DAL` project and add the following code:

```

protected override void OnModelCreating(ModelBuilder modelBuilder)
{
}

```

In this method, the `ModelBuilder` parameter exposes the Fluent API to further define the models and the resulting database.

Creating the Unique Index for Email Addresses on the Customer Table

The application requirements specify that each email address in the `Customer` table be unique. To create a unique index on the `EmailAddress` field, enter the following code in the `OnModelCreating()` method:

```

modelBuilder.Entity<Customer>(entity =>
{
    entity.HasIndex(e => e.EmailAddress).HasName("IX_Customers").IsUnique();
});

```


Setting Default Values on the Order Table

In the `Order` model, the application requirements specify that both the `OrderDate` and `ShipDate` fields have default values of the current date. Add the following code to the `OnModelCreating` method:

```
modelBuilder.Entity<Order>(entity =>
{
    entity.Property(e => e.OrderDate)
        .HasColumnType("datetime")
        .HasDefaultValueSql("getdate()");
    entity.Property(e => e.ShipDate)
        .HasColumnType("datetime")
        .HasDefaultValueSql("getdate()");
});
```

Creating the Computed Column for Order Details

Each record in the `OrderDetail` model has a quantity and a unit cost, and as mentioned, the `LineItemTotal` field needs to be calculated by the database by multiplying the `Quantity` and `UnitCost` fields. Both the `UnitCost` and `LineItemTotal` fields should be set to the SQL Server money type. Add the following code to the `OnModelCreating` method:

```
modelBuilder.Entity<OrderDetail>(entity =>
{
    entity.Property(e => e.LineItemTotal)
        .HasColumnType("money")
        .HasComputedColumnSql("[Quantity]*[UnitCost]");
    entity.Property(e => e.UnitCost).HasColumnType("money");
});
```

Specifying the SQL Server Money Type for the Product Table

The only changes to the `Product` model is setting the `UnitCost` and `CurrentPrice` fields to SQL Server money type. Add the following code to the `OnModelCreating` method:

```
modelBuilder.Entity<Product>(entity =>
{
    entity.Property(e => e.UnitCost).HasColumnType("money");
    entity.Property(e => e.CurrentPrice).HasColumnType("money");
});
```

Updating the Shopping Cart Record Model

There are three updates to the `ShoppingCartRecord` model to satisfy the application requirements. First, create a unique index based on the records `Id`, `CustomerId`, and `ProductId` properties. This also demonstrates how to create an index based on multiple properties. Next, set the default value for the

DateCreated value to the current date, and finally, set the default value for the Quantity field to 1. To do this, add the following code to the OnModelCreating method:

```
modelBuilder.Entity<ShoppingCartRecord>(entity =>
{
    entity.HasIndex(e => new {ShoppingCartRecordId = e.Id, e.ProductId, e.CustomerId })
        .HasName("IX_ShoppingCart")
        .IsUnique();
    entity.Property(e => e.DateCreated)
        .HasColumnType("datetime")
        .HasDefaultValueSql("getdate()");
    entity.Property(e => e.Quantity).HasDefaultValue(1);
});
```

Updating the Database to Match the Model

The next step is to publish all of the changes into the database.

Creating the Migration

To create the migration, open Package Manager Console, change the working directory to SpyStore.DAL, and run the following command:

```
dotnet ef migrations add RemainingTables -o EF\Migrations -c SpyStore.DAL.EF.StoreContext
```

This adds another migration class into the EF\Migrations directory in the SpyStore.DAL project. Just like the first migration you created earlier in this chapter, the Up() method contains the code to update the database to match the current models. The Down() method will revert the status to the previous migration.

Deploying the Migration

The final step is to execute the migration. Enter the following at the command prompt in the Package Manager Console:

```
dotnet ef database update RemainingTables -c SpyStore.DAL.EF.StoreContext
```

This updates your database to match the model and adds a new record into the __EFMigrationHistory table.

Adding the Stored Procedure and User Defined Function

Now that the database is mostly completed, it's time to add the stored procedure and user defined function (referred to earlier in the chapter) by creating a new migration, manually updating the code, and deploying the changes to the database.

Adding a New Migration

Migrations can be created at any time, even though nothing has changed in the model. If nothing has changed, the `Up` and `Down` methods will be empty, but all of the infrastructure needed to add custom T-SQL is contained in the migration file. Open Package Manager Console (PMC), change to the `SpyStore.DAL` directory, and create a new migration. Use the following commands to do this (if PMC is already in the `SpyStore.DAL` directory, skip the first command):

```
cd .\SpyStore.DAL
dotnet ef migrations add TSQL -o EF\Migrations -c SpyStore.DAL.EF.StoreContext
```

Implementing the `Up()` Method

In the `Up()` method, use the `Sql()` method of the `MigrationBuilder` object to execute the SQL necessary to build your stored procedure. Add the following to the `Up()` method:

```
protected override void Up(MigrationBuilder migrationBuilder)
{
    string sql = "CREATE FUNCTION Store.GetOrderTotal ( @OrderId INT ) " +
        "RETURNS MONEY WITH SCHEMABINDING " +
        "BEGIN " +
        "DECLARE @Result MONEY; " +
        "SELECT @Result = SUM([Quantity]*[UnitCost]) FROM Store.OrderDetails " +
        " WHERE OrderId = @OrderId; RETURN @Result END";
    migrationBuilder.Sql(sql);

    sql = "CREATE PROCEDURE [Store].[PurchaseItemsInCart] " +
        " (@customerId INT = 0, @orderId INT OUTPUT) AS BEGIN " +
        " SET NOCOUNT ON; " +
        " INSERT INTO Store.Orders (CustomerId, OrderDate, ShipDate) " +
        "     VALUES(@customerId, GETDATE(), GETDATE()); " +
        " SET @orderId = SCOPE_IDENTITY(); " +
        " DECLARE @TranName VARCHAR(20);SELECT @TranName = 'CommitOrder'; " +
        " BEGIN TRANSACTION @TranName; " +
        " BEGIN TRY " +
        "     INSERT INTO Store.OrderDetails (OrderId, ProductId, Quantity, UnitCost) " +
        "     SELECT @orderId, ProductId, Quantity, p.CurrentPrice " +
        "     FROM Store.ShoppingCartRecords scr " +
        "     INNER JOIN Store.Products p ON p.Id = scr.ProductId " +
        "     WHERE CustomerId = @customerId; " +
        "     DELETE FROM Store.ShoppingCartRecords WHERE CustomerId = @customerId; " +
        "     COMMIT TRANSACTION @TranName; " +
        " END TRY " +
        " BEGIN CATCH " +
        "     ROLLBACK TRANSACTION @TranName; " +
        "     SET @orderId = -1; " +
        " END CATCH; " +
        "END;";
    migrationBuilder.Sql(sql);
}
```

The preceding code first creates a user defined function for calculating the order total based on an `OrderId`. The next block creates the stored procedure to process an order.

Implementing the `Down()` Method

Update the `Down` method to remove the function and stored procedure and return the database to the state it was in before the migration:

```
protected override void Down(MigrationBuilder migrationBuilder)
{
    migrationBuilder.Sql("DROP FUNCTION [Store].[GetOrderTotal]");
    migrationBuilder.Sql("DROP PROCEDURE [Store].[PurchaseItemsInCart]");
}
```

Updating the Database

The final step is to execute the migration. Enter the following at the command prompt in Package Manager Console (or the command line):

```
dotnet ef database update
```

Adding the `OrderTotal` Calculated Field

Now that the user defined function is in place, the `OrderTotal` field can be added to the `Order` class. This order has to be followed, since SQL Server will not allow an existing field to be created as a computed column. It has to be set on creation of the column.

Updating the `Order` Class

Open the `Order.cs` class and add the following property definition:

```
[Display(Name = "Total")]
public decimal? OrderTotal { get; set; }
```

Making `OrderTotal` a Computed Column

The final change is changing the column definition using the Fluent API. Open the `StoreContext.cs` class and update the `Order` definition block to the following:

```
modelBuilder.Entity<Order>(entity =>
{
    entity.Property(e => e.OrderDate)
        .HasColumnType("datetime")
        .HasDefaultValueSql("getdate()");

    entity.Property(e => e.ShipDate)
        .HasColumnType("datetime")
        .HasDefaultValueSql("getdate()");
}
```

```
entity.Property(e => e.OrderTotal)
    .HasColumnType("money")
    .HasComputedColumnSql("Store.GetOrderTotal([Id])");
});
```

Adding a New Migration and Update the Database

Open Package Manager Console (PMC), change to the `SpyStore.DAL` directory, and create a new migration. Use the following commands to do this:

```
dotnet ef migrations add Final -o EF\Migrations -c SpyStore.DAL.EF.StoreContext
```

Update the database with the following command:

```
dotnet ef database update
```

This completes the `SpyStore` database.

Automating the Migrations

This project won't have any more migrations, so it's time to automate the execution of the migrations for other developers on your team. The `Database.Migrate` function first makes sure the database exists (and creates it if not), then runs any pending migrations for the context.

Open the `StoreContext.cs` file and add the `try...catch` block and the `Database.Migrate` command into the second constructor. This is the constructor that ASP.NET MVC uses through dependency injection. The first constructor doesn't need it since the data initializer (developed later in this chapter) calls `Database.Migrate`. Your updated code will look like this:

```
public StoreContext()
{
}
public StoreContext(DbContextOptions options) : base(options)
{
    try
    {
        Database.Migrate();
    }
    catch (Exception)
    {
        //Should do something intelligent here
    }
}
```

■ **Note** There is also a `Database.EnsureCreated` function. This function ensures the database exists, and then creates the tables and fields based on the `ModelSnapshot`, but does not run any migrations. Therefore, in this project, the stored procedure and user defined function would not get created.

Adding the View Models

View models are used to shape the data into a form that is more useful for presenting the data. They are typically created by combining parts of two or more existing models into one class. The view models will be used by ASP.NET Core MVC in later chapters.

The Product with Category View Model

This view model combines the `Product` and `Category` data into one class. It also doubles as a base class for the other view models that need `Product` and `Category` information together. This is used on the home page and the search results pages. Create a new folder in the `SpyStore.Models` project named `ViewModels`. In this folder, create a new folder named `Base`, add a new class named `ProductAndCategoryBase.cs`, and update the code to the following:

```
using System.ComponentModel.DataAnnotations;
using SpyStore.Models.Entities.Base;

namespace SpyStore.Models.ViewModels.Base
{
    public class ProductAndCategoryBase : EntityBase
    {
        public int CategoryId { get; set; }
        [Display(Name = "Category")]
        public string CategoryName { get; set; }
        public int ProductId { get; set; }
        [MaxLength(3800)]
        public string Description { get; set; }
        [MaxLength(50)]
        [Display(Name = "Model")]
        public string ModelName { get; set; }
        [Display(Name="Is Featured Product")]
        public bool IsFeatured { get; set; }
        [MaxLength(50)]
        [Display(Name = "Model Number")]
        public string ModelNumber { get; set; }
        [MaxLength(150)]
        public string ProductImage { get; set; }
        [MaxLength(150)]
        public string ProductImageLarge { get; set; }
        [MaxLength(150)]
        public string ProductImageThumb { get; set; }
        [DataType(DataType.Currency), Display(Name = "Cost")]
        public decimal UnitCost { get; set; }
        [DataType(DataType.Currency), Display(Name = "Price")]
        public decimal CurrentPrice { get; set; }
        [Display(Name="In Stock")]
        public int UnitsInStock { get; set; }
    }
}
```

The Order Detail with Product Info View Model

The order detail screen shows product information as well, so the next view model is the Order Detail records combined with the product information. Add a new file to the ViewModels folder named OrderDetailWithProductInfo.cs and update the code to match this:

```
using System.ComponentModel.DataAnnotations;
using System.ComponentModel.DataAnnotations.Schema;
using SpyStore.Models.Entities.Base;
using SpyStore.Models.ViewModels.Base;

namespace SpyStore.Models.ViewModels
{
    public class OrderDetailWithProductInfo : ProductAndCategoryBase
    {
        public int OrderId { get; set; }
        [Required]
        public int Quantity { get; set; }
        [DataType(DataType.Currency), Display(Name = "Total")]
        public decimal? LineItemTotal { get; set; }
    }
}
```

The Order with OrderDetails View Model

When viewing an order online, the order details contain product and category information as well. Add a new class to the ViewModels folder named OrderWithDetailsAndProductInfo.cs and update the code to the following:

```
using System;
using System.Collections.Generic;
using System.ComponentModel.DataAnnotations;
using SpyStore.Models.Entities.Base;

namespace SpyStore.Models.ViewModels
{
    public class OrderWithDetailsAndProductInfo : EntityBase
    {
        public int CustomerId { get; set; }
        [DataType(DataType.Currency), Display(Name = "Total")]
        public decimal? OrderTotal { get; set; }
        [DataType(DataType.Date)]
        [Display(Name = "Date Ordered")]
        public DateTime OrderDate { get; set; }
        [DataType(DataType.Date)]
        [Display(Name = "Date Shipped")]
        public DateTime ShipDate { get; set; }
        public IList<OrderDetailWithProductInfo> OrderDetails { get; set; }
    }
}
```

The Cart Record with Product Infor View Model

The final view model combines the ShoppingCartRecord, Product, and Category tables. Add a new class to the ViewModels folder named CartRecordWithProductInfo.cs and update the code to the following:

```
using System;
using System.ComponentModel.DataAnnotations;
using SpyStore.Models.Entities.Base;
using SpyStore.Models.ViewModels.Base;

namespace SpyStore.Models.ViewModels
{
    public class CartRecordWithProductInfo : ProductAndCategoryBase
    {
        [DataType(DataType.Date), Display(Name = "Date Created")]
        public DateTime? DateCreated { get; set; }
        public int? CustomerId { get; set; }
        public int Quantity { get; set; }
        [DataType(DataType.Currency), Display(Name = "Line Total")]
        public decimal LineItemTotal { get; set; }
    }
}
```

Completing the Repositories

Now that the database and the view models are in place, it's time to complete the repositories.

Extending the Interfaces

ASP.NET Core MVC has built-in support for Dependency Injection (DI). Instead of passing instances of the repositories to the controllers, everything is referenced as an interface. Each individual repository will need its API represented as an interface, building on the IRepo<T> interface. To get started, add a new directory in the Repos directory named Interfaces.

Adding the ICategoryRepo Interface

Add a new file in the Interfaces directory named ICategoryRepo.cs. Update the code to match the following:

```
using System.Collections.Generic;
using SpyStore.DAL.Repos.Base;
using SpyStore.Models.Entities;

namespace SpyStore.DAL.Repos.Interfaces
{
    public interface ICategoryRepo : IRepo<Category>
    {
        IEnumerable<Category> GetAllWithProducts();
        Category GetOneWithProducts(int? id);
    }
}
```


Adding the IProductRepo Interface

Add a new file to the Interfaces directory named `IProductRepo.cs`. Update the code to match the following:

```
using System.Collections.Generic;
using SpyStore.DAL.Repos.Base;
using SpyStore.Models.Entities;
using SpyStore.Models.ViewModels.Base;

namespace SpyStore.DAL.Repos.Interfaces
{
    public interface IProductRepo : IRepo<Product>
    {
        IEnumerable<ProductAndCategoryBase> Search(string searchString);
        IEnumerable<ProductAndCategoryBase> GetAllWithCategoryName();
        IEnumerable<ProductAndCategoryBase> GetProductsForCategory(int id);
        IEnumerable<ProductAndCategoryBase> GetFeaturedWithCategoryName();
        ProductAndCategoryBase GetOneWithCategoryName(int id);
    }
}
```

Adding the ICustomerRepo Interface

There aren't any additional methods for the Customer class, but a custom interface is going to be added anyway for consistency. Add a new file in the Interfaces directory named `ICustomerRepo.cs`. Update the code to match the following:

```
using SpyStore.DAL.Repos.Base;
using SpyStore.Models.Entities;

namespace SpyStore.DAL.Repos.Interfaces
{
    public interface ICustomerRepo : IRepo<Customer>
    {
    }
}
```

Adding the IOrderRepo Interface

Add a new file in the Interfaces directory named `IOrderRepo.cs`. Update the code to match the following:

```
using System.Collections.Generic;
using SpyStore.DAL.Repos.Base;
using SpyStore.Models.Entities;
using SpyStore.Models.ViewModels;
```

```
namespace SpyStore.DAL.Repos.Interfaces
{
    public interface IOrderRepo :IRepo<Order>
    {
        IEnumerable<Order> GetOrderHistory(int customerId);
        OrderWithDetailsAndProductInfo GetOneWithDetails(int customerId, int orderId);
    }
}
```

Adding the IOrderDetailRepo Interface

Add a new file in the Interfaces directory named `IOrderDetailRepo.cs`. Update the code to match the following:

```
using System.Collections.Generic;
using SpyStore.DAL.Repos.Base;
using SpyStore.Models.Entities;
using SpyStore.Models.ViewModels;

namespace SpyStore.DAL.Repos.Interfaces
{
    public interface IOrderDetailRepo :IRepo<OrderDetail>
    {
        IEnumerable<OrderDetailWithProductInfo> GetCustomersOrdersWithDetails(int customerId);
        IEnumerable<OrderDetailWithProductInfo> GetSingleOrderWithDetails(int orderId);
    }
}
```

Adding the IShoppingCartRepo Interface

Add a new file in the Interfaces directory named `IShoppingCartRepo.cs`. Update the code to match the following:

```
using System.Collections.Generic;
using SpyStore.DAL.Repos.Base;
using SpyStore.Models.Entities;
using SpyStore.Models.ViewModels;

namespace SpyStore.DAL.Repos.Interfaces
{
    public interface IShoppingCartRepo :IRepo<ShoppingCartRecord>
    {
        CartRecordWithProductInfo GetShoppingCartRecord(int customerId, int productId);
        IEnumerable<CartRecordWithProductInfo> GetShoppingCartRecords(int customerId);
        int Purchase(int customerId);
        ShoppingCartRecord Find(int customerId, int productId);
        int Update(ShoppingCartRecord entity, int? quantityInStock, bool persist = true);
        int Add(ShoppingCartRecord entity, int? quantityInStock, bool persist = true);
    }
}
```

Adding/Updating the Repositories

Each of the repositories derives from `RepoBase<T>` to gain access to the base functionality. They also need to implement their model specific interface that you just added to the project. Each repository is discussed in detail in the next sections.

Updating the Category Repository

Open the `CategoryRepo.cs` file in the `Repos` directory. Add the following using statement to the top of the file:

```
using SpyStore.DAL.Repos.Interfaces;
```

Add the `ICategoryRepo` interface to the class definition and add the following two methods to implement the `ICategoryRepo` interface:

```
public Category GetOneWithProducts(int? id) =>
    Table.Include(x => x.Products).SingleOrDefault(x => x.Id == id);

public IEnumerable<Category> GetAllWithProducts() =>
    Table.Include(x => x.Products).ToList();
```

Both of these methods use the `Include` extension method to eagerly fetch the related `Product` records for the `Category(s)` being retrieved. EF initiates a `JOIN` operation and runs a query similar to the following SQL query (as gathered by SQL Server Profiler):

```
exec sp_executesql N'SELECT [p].[Id], [p].[CategoryId], [p].[CurrentPrice], [p].
[Description], [p].[IsFeatured], [p].[ModelName], [p].[ModelNumber], [p].[ProductImage],
[p].[ProductImageLarge], [p].[ProductImageThumb], [p].[TimeStamp], [p].[UnitCost], [p].
[UnitsInStock]
FROM [Store].[Products] AS [p]
INNER JOIN (
    SELECT DISTINCT TOP(2) [x].[Id]
    FROM [Store].[Categories] AS [x]
    WHERE [x].[Id] = @_id_0
) AS [x] ON [p].[CategoryId] = [x].[Id]
ORDER BY [x].[Id]',N'@_id_0 int',@_id_0=1
```

Adding the Product Repository

Start by adding a new file into the `Repos` directory named `ProductRepo.cs`. Next, update the code to the following:

```
using System.Collections.Generic;
using System.Linq;
using Microsoft.EntityFrameworkCore;
using SpyStore.DAL.EF;
using SpyStore.DAL.Repos.Base;
using SpyStore.DAL.Repos.Interfaces;
using SpyStore.Models.Entities;
using SpyStore.Models.ViewModels.Base;
```

```

namespace SpyStore.DAL.Repos
{
    public class ProductRepo : RepoBase<Product>, IProductRepo
    {
        public ProductRepo(DbContextOptions<StoreContext> options) : base(options)
        {
            Table = Context.Products;
        }
        public ProductRepo() : base()
        {
            Table = Context.Products;
        }
        public override IEnumerable<Product> GetAll() => Table.OrderBy(x => x.ModelName);

        internal ProductAndCategoryBase GetRecord(Product p, Category c)
            => new ProductAndCategoryBase()
            {
                CategoryName = c.CategoryName,
                CategoryId = p.CategoryId,
                CurrentPrice = p.CurrentPrice,
                Description = p.Description,
                IsFeatured = p.IsFeatured,
                Id = p.Id,
                ModelName = p.ModelName,
                ModelNumber = p.ModelNumber,
                ProductImage = p.ProductImage,
                ProductImageLarge = p.ProductImageLarge,
                ProductImageThumb = p.ProductImageThumb,
                TimeStamp = p.TimeStamp,
                UnitCost = p.UnitCost,
                UnitsInStock = p.UnitsInStock
            };

        public IEnumerable<ProductAndCategoryBase> GetProductsForCategory(int id)
            => Table
                .Where(p => p.CategoryId == id)
                .Include(p => p.Category)
                .Select(item => GetRecord(item, item.Category))
                .OrderBy(x => x.ModelName);
        public IEnumerable<ProductAndCategoryBase> GetAllWithCategoryName()
            => Table
                .Include(p => p.Category)
                .Select(item => GetRecord(item, item.Category))
                .OrderBy(x => x.ModelName);
        public IEnumerable<ProductAndCategoryBase> GetFeaturedWithCategoryName()
            => Table
                .Where(p => p.IsFeatured)
                .Include(p => p.Category)
                .Select(item => GetRecord(item, item.Category))
                .OrderBy(x => x.ModelName);
        public ProductAndCategoryBase GetOneWithCategoryName(int id)

```

```

=> Table
    .Where(p => p.Id == id)
    .Include(p => p.Category)
    .Select(item => GetRecord(item, item.Category))
    .SingleOrDefault();
public IEnumerable<ProductAndCategoryBase> Search(string searchString)
=> Table
    .Where(p => p.Description.ToLower().Contains(searchString.ToLower())
        || p.ModelName.ToLower().Contains(searchString.ToLower()))
    .Include(p => p.Category)
    .Select(item => GetRecord(item, item.Category))
    .OrderBy(x => x.ModelName);
}
}

```

The `GetAll()` method overrides the base `GetAll()` method to return the `Product` records in order of the `ModelName` property. The `GetProductsForCategory()` method returns all of the `Product` records for a specific `Category Id`, ordered by `ModelName`.

The internal `GetRecord` method is used to project the LINQ results in the following methods (`GetAllWithCategoryName()`, `GetFeaturedWithCategoryName()`, `GetOneWithCategoryName()`, and `Search()`) to create the `ProductAndCategoryBase` view model.

Adding the Customer Repository

The `CustomerRepo` provides the core features of the `BaseRepo` class and implements the `ICustomerRepo` interface. Add a new class in the `Repos` directory named `CustomerRepo.cs` and update the code to the following:

```

using System.Collections.Generic;
using System.Linq;
using Microsoft.EntityFrameworkCore;
using SpyStore.DAL.EF;
using SpyStore.DAL.Repos.Base;
using SpyStore.DAL.Repos.Interfaces;
using SpyStore.Models.Entities;

namespace SpyStore.DAL.Repos
{
    public class CustomerRepo : RepoBase<Customer>, ICustomerRepo
    {
        public CustomerRepo(DbContextOptions<StoreContext> options) : base(options)
        {
        }
        public CustomerRepo() : base()
        {
        }
        public override IEnumerable<Customer> GetAll() => Table.OrderBy(x => x.FullName);
        public override IEnumerable<Customer> GetRange(int skip, int take)
            => GetRange(Table.OrderBy(x => x.FullName), skip, take);
    }
}

```

Adding the OrderDetail Repository

The `OrderDetail` repository adds a new LINQ extension into your toolbox: `ThenInclude`. This method (introduced in EF Core) can simplify long LINQ queries as the queries walk the domain model. Add a new class in the `Repos` directory named `OrderDetailRepo.cs` and update the code to the following:

```
using System.Collections.Generic;
using System.Linq;
using Microsoft.EntityFrameworkCore;
using SpyStore.DAL.EF;
using SpyStore.DAL.Repos.Base;
using SpyStore.Models.Entities;
using SpyStore.Models.ViewModels;
using SpyStore.DAL.Repos.Interfaces;

namespace SpyStore.DAL.Repos
{
    public class OrderDetailRepo : RepoBase<OrderDetail>, IOrderDetailRepo
    {
        public OrderDetailRepo(DbContextOptions<StoreContext> options) : base(options)
        {
        }
        public OrderDetailRepo()
        {
        }
        public override IEnumerable<OrderDetail> GetAll()=> Table.OrderBy(x => x.Id);
        public override IEnumerable<OrderDetail> GetRange(int skip, int take)
            => GetRange(Table.OrderBy(x => x.Id), skip, take);

        internal IEnumerable<OrderDetailWithProductInfo> GetRecords(IQueryable<OrderDetail> query)
            => query.Include(x => x.Product).ThenInclude(p => p.Category)
                .Select(x => new OrderDetailWithProductInfo
                {
                    OrderId = x.OrderId,
                    ProductId = x.ProductId,
                    Quantity = x.Quantity,
                    UnitCost = x.UnitCost,
                    LineItemTotal = x.LineItemTotal,
                    Description = x.Product.Description,
                    ModelName = x.Product.ModelName,
                    ProductImage = x.Product.ProductImage,
                    ProductImageLarge = x.Product.ProductImageLarge,
                    ProductImageThumb = x.Product.ProductImageThumb,
                    ModelNumber = x.Product.ModelNumber,
                    CategoryName = x.Product.Category.CategoryName
                })
                .OrderBy(x => x.ModelName);

        public IEnumerable<OrderDetailWithProductInfo> GetCustomersOrdersWithDetails(
            int customerId)
            => GetRecords(Table.Where(x => x.Order.CustomerId == customerId));
    }
}
```

```

    public IEnumerable<OrderDetailWithProductInfo> GetSingleOrderWithDetails(int orderId)
        => GetRecords(Table.Where(x => x.Order.Id == orderId));
    }
}

```

The internal `GetRecords` method starts with an `IQueryable<OrderDetail>` and then uses a projection to populate a list of `OrderDetailWithProductInfo`. The `GetCustomersOrdersWithDetails` and `GetSingleOrderWithDetails` methods create the initial query (searching by `CustomerId` or `OrderId`, respectively).

Adding the Order Repository

The `OrderRepo` class leverages the `OrderDetailRepo` class to get the `OrderDetail` records into the view model. Add a new class in the `Repos` directory named `OrderRepo.cs` and update the code to the following:

```

using System.Collections.Generic;
using System.Linq;
using Microsoft.EntityFrameworkCore;
using SpyStore.DAL.EF;
using SpyStore.DAL.Repos.Base;
using SpyStore.DAL.Repos.Interfaces;
using SpyStore.Models.Entities;
using SpyStore.Models.ViewModels;

namespace SpyStore.DAL.Repos
{
    public class OrderRepo : RepoBase<Order>, IOrderRepo
    {
        private readonly IOrderDetailRepo _orderDetailRepo;
        public OrderRepo(DbContextOptions<StoreContext> options, IOrderDetailRepo
orderDetailRepo)
            : base(options)
        {
            _orderDetailRepo = orderDetailRepo;
        }
        public OrderRepo(IOrderDetailRepo orderDetailRepo)
        {
            _orderDetailRepo = orderDetailRepo;
        }
        public override IEnumerable<Order> GetAll()=> Table.OrderByDescending(x => x.OrderDate);
        public override IEnumerable<Order> GetRange(int skip, int take)
            => GetRange(Table.OrderByDescending(x => x.OrderDate), skip, take);
        public IEnumerable<Order> GetOrderHistory(int customerId)
            => Table.Where(x => x.CustomerId == customerId)
                .Select(x => new Order
                {
                    Id = x.Id,
                    TimeStamp = x.TimeStamp,
                    CustomerId = customerId,
                    OrderDate = x.OrderDate,
                }
            );
    }
}

```

```

        OrderTotal = x.OrderTotal,
        ShipDate = x.ShipDate,
    });
public OrderWithDetailsAndProductInfo GetOneWithDetails(int customerId, int orderId)
=> Table
    .Include(x => x.OrderDetails)
    .Where(x => x.CustomerId == customerId && x.Id == orderId)
    .Select(x => new OrderWithDetailsAndProductInfo
        {
            Id = x.Id,
            CustomerId = customerId,
            OrderDate = x.OrderDate,
            OrderTotal = x.OrderTotal,
            ShipDate = x.ShipDate,
            OrderDetails = _orderDetailRepo.GetSingleOrderWithDetails(orderId).ToList()
        })
    .FirstOrDefault();
}
}

```

The constructors also take `IOrderDetailRepo` as a parameter. This will be instantiated by the DI Framework in ASP.NET Core (or manually in unit test classes). The `GetOrderHistory` gets all of the orders for a customer, removing the navigation properties from the result set. The `GetOneWithDetails()` method uses `OrderDetailRepo` to retrieve the `OrderDetails` with `Product` and `Category` information, which is then added to the view model.

Adding the `InvalidQuantityException`

The `ShoppingCartRepo` will throw a custom exception when trying to add more items to the cart than are available in inventory. Add a new directory named `Exceptions` in the `SpyStore.DAL` project. In this directory, add a new class named `InvalidQuantityException.cs`. Make the class public and derive from the `System.Exception` type. Then stub out the three main constructors, as follows:

```

public class InvalidQuantityException : Exception
{
    public InvalidQuantityException() {}
    public InvalidQuantityException(string message) : base(message) {}
    public InvalidQuantityException(string message, Exception innerException)
        : base(message, innerException) { }
}

```

Adding the `ShoppingCartRecords Repository`

Add a new class in the `Repos` directory named `ShoppingCartRepo.cs` and update the code to the following:

```

using System;
using System.Collections.Generic;
using System.Data;
using System.Data.SqlClient;
using System.Linq;

```



```

using Microsoft.EntityFrameworkCore;
using SpyStore.DAL.EF;
using SpyStore.DAL.Exceptions;
using SpyStore.DAL.Repos.Base;
using SpyStore.DAL.Repos.Interfaces;
using SpyStore.Models.Entities;
using SpyStore.Models.ViewModels;

namespace SpyStore.DAL.Repos
{
    public class ShoppingCartRepo : RepoBase<ShoppingCartRecord>, IShoppingCartRepo
    {
        private readonly IProductRepo _productRepo;
        public ShoppingCartRepo(DbContextOptions<StoreContext> options,
            IProductRepo productRepo) : base(options)
        {
            _productRepo = productRepo;
        }
        public ShoppingCartRepo(IProductRepo productRepo) : base()
        {
            _productRepo = productRepo;
        }
        public override IEnumerable<ShoppingCartRecord> GetAll()
            => Table.OrderByDescending(x => x.DateCreated);
        public override IEnumerable<ShoppingCartRecord> GetRange(int skip, int take)
            => GetRange(Table.OrderByDescending(x => x.DateCreated), skip, take);
        public ShoppingCartRecord Find(int customerId, int productId)
        {
            return Table.FirstOrDefault(
                x => x.CustomerId == customerId && x.ProductId == productId);
        }
        public override int Update(ShoppingCartRecord entity, bool persist = true)
        {
            return Update(entity, _productRepo.Find(entity.ProductId)?.UnitsInStock, persist);
        }
        public int Update(ShoppingCartRecord entity, int? quantityInStock, bool persist = true)
        {
            if (entity.Quantity <= 0)
            {
                return Delete(entity, persist);
            }
            if (entity.Quantity > quantityInStock)
            {
                throw new InvalidQuantityException("Can't add more product than available in stock");
            }
            return base.Update(entity, persist);
        }
        public override int Add(ShoppingCartRecord entity, bool persist = true)
        {
            return Add(entity, _productRepo.Find(entity.ProductId)?.UnitsInStock, persist);
        }
    }
}

```

```

public int Add(ShoppingCartRecord entity, int? quantityInStock, bool persist = true)
{
    var item = Find(entity.CustomerId, entity.ProductId);
    if (item == null)
    {
        if (quantityInStock != null && entity.Quantity > quantityInStock.Value)
        {
            throw new InvalidQuantityException("Can't add more product than available in stock");
        }
        return base.Add(entity, persist);
    }
    item.Quantity += entity.Quantity;
    return item.Quantity <= 0 ? Delete(item,persist) : Update(item,quantityInStock,persist);
}

internal CartRecordWithProductInfo GetRecord(
    int customerId, ShoppingCartRecord scr, Product p, Category c)
=> new CartRecordWithProductInfo
    {
        Id = scr.Id,
        DateCreated = scr.DateCreated,
        CustomerId = customerId,
        Quantity = scr.Quantity,
        ProductId = scr.ProductId,
        Description = p.Description,
        ModelName = p.ModelName,
        ModelNumber = p.ModelNumber,
        ProductImage = p.ProductImage,
        ProductImageLarge = p.ProductImageLarge,
        ProductImageThumb = p.ProductImageThumb,
        CurrentPrice = p.CurrentPrice,
        UnitsInStock = p.UnitsInStock,
        CategoryName = c.CategoryName,
        LineItemTotal = scr.Quantity * p.CurrentPrice,
        TimeStamp = scr.TimeStamp
    };

public CartRecordWithProductInfo GetShoppingCartRecord(int customerId, int productId)
=> Table
    .Where(x => x.CustomerId == customerId && x.ProductId == productId)
    .Include(x => x.Product)
    .ThenInclude(p => p.Category)
    .Select(x => GetRecord(customerId, x, x.Product, x.Product.Category))
    .FirstOrDefault();

public IEnumerable<CartRecordWithProductInfo> GetShoppingCartRecords(int customerId)
=> Table
    .Where(x => x.CustomerId == customerId)
    .Include(x => x.Product)
    .ThenInclude(p => p.Category)
    .Select(x => GetRecord(customerId, x, x.Product, x.Product.Category))
    .OrderBy(x => x.ModelName);

public int Purchase(int customerId)

```

```

{
    var customerIdParam = new SqlParameter("@customerId", SqlDbType.Int)
    {
        Direction = ParameterDirection.Input,
        Value = customerId
    };
    var orderIdParam = new SqlParameter("@orderId", SqlDbType.Int)
    {
        Direction = ParameterDirection.Output
    };
    try
    {
        Context.Database.ExecuteNonQuery(
            "EXEC [Store].[PurchaseItemsInCart] @customerId, @orderid out",
            customerIdParam, orderIdParam);
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex);
        return -1;
    }
    return (int)orderIdParam.Value;
}
}
}

```

The additional Find method uses the Id of the Customer and the Id of a Product to find a specific record. The Add method checks to see if there are any of the same products already in the cart. If there are, the quantity is updated (instead of adding a new record). If there aren't any, the product is added to the cart. When the cart is updated, if the new quantity is less than or equal to zero, the item is deleted from the cart. Otherwise, the quantity is simply updated. Both the Add and Update methods check the available inventory, and if the user is attempting to add more records into the cart than are available, an *InvalidQuantityException* is thrown.

The GetRecord method simplifies the projections in the GetShoppingCartRecord and GetShoppingCartRecords methods.

The Purchase() method executes the stored procedure built earlier in the chapter.

■ **Note** The data access layer for the SpyStore data access layer is now complete. The remainder of the chapter covers seeding the database with data.

Initializing the Database with Data

A powerful feature of EF 6.x was the ability to initialize the database with data using the built-in *DropCreatedatabaseIfModelChanges<TContext>* and *DropCreateDatabaseAlways<TContext>* classes. These classes are not replicated in EF Core, so the process to initialize data needs to be built by hand.

■ **Note** The code shown in this section is a small subset of the full code available in the downloadable chapter files. The examples have been shortened to save space in this chapter.

Two files will be created, one to create sample data, and the other to handle loading and cleaning the database. Start by creating a folder named `Initializers` in the `SpyStore.DAL` project.

Creating Sample Data

Add a new file named `StoreSampleData.cs` to the `Initializers` folder. Add or update the using statements to the following:

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.EntityFrameworkCore;
using SpyStore.Models.Entities;
using SpyStore.DAL.EF;
```

The file consists of a series of static functions that create sample data.

```
namespace SpyStore.DAL.Initializers
{
    public static class StoreSampleData
    {
        public static IEnumerable<Category> GetCategories() => new List<Category>
        {
            new Category {CategoryName = "Communications"},
            new Category {CategoryName = "Deception"},
            new Category {CategoryName = "Travel"},
        };
        public static IList<Product> GetProducts(IList<Category> categories)
        {
            var products = new List<Product>();
            foreach (var category in categories)
            {
                switch (category.CategoryName)
                {
                    case "Communications":
                        products.AddRange(new List<Product>
                        {
                            new Product
                            {
                                Category = category,
                                CategoryId = category.Id,
                                ProductImage = "product-image.png",
                                ProductImageLarge = "product-image-lg.png",
                                ProductImageThumb = "product-thumb.png",
                                ModelNumber = "RED1",
                                ModelName = "Communications Device",
                            }
                        });
                }
            }
        }
    }
}
```

```

        UnitCost = 49.99M,
        CurrentPrice = 49.99M,
        Description = "Lorem ipsum",
        UnitsInStock = 2,
        IsFeatured = true
    },
    });
    break;
default:
    break;
}
}
return products;
}
public static IEnumerable<Customer> GetAllCustomerRecords(StoreContext context)
=> new List<Customer>
{
    new Customer() { EmailAddress = "spy@secrets.com", Password = "Foo",
        FullName = "Super Spy",
    }
};
public static List<Order> GetOrders(Customer customer, StoreContext context)
=> new List<Order>
{
    new Order()
    {
        Customer = customer,
        OrderDate = DateTime.Now.Subtract(new TimeSpan(20, 0, 0, 0)),
        ShipDate = DateTime.Now.Subtract(new TimeSpan(5, 0, 0, 0)),
        OrderDetails = GetOrderDetails(context)
    }
};
public static IList<OrderDetail> GetOrderDetails(
    Order order, StoreContext context)
{
    var prod1 = context.Categories.Include(c => c.Products).FirstOrDefault()?
        .Products.FirstOrDefault();
    var prod2 = context.Categories.Skip(1).Include(c => c.Products).FirstOrDefault()?
        .Products.FirstOrDefault();
    var prod3 = context.Categories.Skip(2).Include(c => c.Products).FirstOrDefault()?
        .Products.FirstOrDefault();
    return new List<OrderDetail>
    {
        new OrderDetail() { Order = order, Product = prod1, Quantity = 3,
            UnitCost = prod1.CurrentPrice},
        new OrderDetail() { Order = order, Product = prod2, Quantity = 2,
            UnitCost = prod2.CurrentPrice},
        new OrderDetail() { Order = order, Product = prod3, Quantity = 5,
            UnitCost = prod3.CurrentPrice},
    };
}
}

```

```

public static IList<ShoppingCartRecord> GetCart(Customer customer, StoreContext context)
{
    var prod1 = context.Categories.Skip(1)
        .Include(c => c.Products).FirstOrDefault()?
        .Products. FirstOrDefault();
    return new List<ShoppingCartRecord>
    {
        new ShoppingCartRecord {
            Customer = customer, DateCreated = DateTime.Now,
            Product = prod1, Quantity = 1}
    };
}
}
}

```

There isn't much to discuss in the previous code listing. Each method creates sample records to use for seeding the database.

Using the Sample Data

The next step is to create the methods that add the sample data to the database as well as remove them. Create a new class named `StoreDataInitializer.cs` in the `Initializers` directory. Add or update the using statements to the file:

```

using System;
using System.Linq;
using Microsoft.EntityFrameworkCore;
using Microsoft.Extensions.DependencyInjection;
using SpyStore.DAL.EF;

```

Update the code listing to match the following:

```

namespace SpyStore.DAL.Initializers
{
    public static class SampleData
    {
        public static void InitializeData(IServiceProvider serviceProvider)
        {
            var context = serviceProvider.GetService<StoreContext>();
            InitializeData(context);
        }
        public static void InitializeData(StoreContext context)
        {
            context.Database.Migrate();
            ClearData(context);
            SeedData(context);
        }
    }
}

```

```

public static void ClearData(StoreContext context)
{
    ExecuteDeleteSQL(context, "Categories");
    ExecuteDeleteSQL(context, "Customers");
    ResetIdentity(context);
}
public static void ExecuteDeleteSQL(StoreContext context, string tableName)
{
    context.Database.ExecuteSqlCommand($"Delete from Store.{tableName}");
}
public static void ResetIdentity(StoreContext context)
{
    var tables = new[] {"Categories", "Customers",
        "OrderDetails", "Orders", "Products", "ShoppingCartRecords"};
    foreach (var itm in tables)
    {
        context.Database.ExecuteSqlCommand($"DBCC CHECKIDENT (\\"Store.{itm}\\", RESEED, -1);");
    }
}
public static void SeedData(StoreContext context)
{
    try
    {
        if (!context.Categories.Any())
        {
            context.Categories.AddRange(StoreSampleData.GetCategories());
            context.SaveChanges();
        }
        if (!context.Products.Any())
        {
            context.Products.AddRange(StoreSampleData.GetProducts(context.Categories.ToList()));
            context.SaveChanges();
        }
        if (!context.Customers.Any())
        {
            context.Customers.AddRange(StoreSampleData.GetAllCustomerRecords(context));
            context.SaveChanges();
        }
        var customer = context.Customers.FirstOrDefault();
        if (!context.Orders.Any())
        {
            context.Orders.AddRange(StoreSampleData.GetOrders(customer, context));
            context.SaveChanges();
        }
        if (!context.ShoppingCartRecords.Any())
        {
            context.ShoppingCartRecords.AddRange(
                StoreSampleData.GetCart(customer, context));
            context.SaveChanges();
        }
    }
}

```

```

        catch (Exception ex)
        {
            Console.WriteLine(ex);
        }
    }
}

```

There are two static methods named `InitializeData()`. The first one takes an `IServiceProvider`, which is a component of the ASP.NET Core Dependency Injection framework, and is used in later chapters. The second one takes an instance of a `StoreContext` and is used by the tests in the downloadable code.

The `ClearData` method deletes all of the records in the database, then resets the identity for all of the primary keys. The `SeedData` method calls the methods in the `StoreSampleData` class to populate the database.

Using the Initializer in Tests

Each test needs to start with the database in a predictable and repeatable state. Before each test is run, code needs to ensure that the database is first reset to a clean state, then add the sample data. After each test, the database then needs to be reset back to the initial state. Recall from the “Unit Testing” section in Chapter 1 that the constructor runs before each test, and the `Dispose` method runs after each test.

The following is a sample from the downloadable unit tests. The constructor clears the database first, in case another test failed to execute the cleanup process, then seeds the data. The `Dispose` method cleans up the data before disposing of the repository.

```

public class OrderRepoTests : IDisposable
{
    private readonly OrderRepo _repo;
    public OrderRepoTests()
    {
        _repo = new OrderRepo(new OrderDetailRepo());
        StoreDataInitializer.ClearData(_repo.Context);
        StoreDataInitializer.InitializeData(_repo.Context);
    }
    public void Dispose()
    {
        StoreDataInitializer.ClearData(_repo.Context);
        _repo.Dispose();
    }
}

```

Using the initializer code is straightforward. The following is an example from the `ProductRepoTests.cs` constructor (available in the `SpyStore.DAL.Tests` project from the downloadable code). It first creates a `ProductRepo` class (this also created a new `StoreContext` instance), then calls the `ClearData` and `InitializeData` static methods of the `StoreDataInitializer` class:

```

_repo = new ProductRepo();
StoreDataInitializer.ClearData(_repo.Context);
StoreDataInitializer.InitializeData(_repo.Context);

```

■ **Source Code** The completed `SpyStore.DAL` solution with unit tests can be found in the Chapter 02 subdirectory of the download files.

Creating NuGet Packages for the Data Access Library

One of the best ways to distribute code between teams is by using private NuGet packages due to the ease of creation and management within other solutions. The `dotnet pack .NET Core CLI` command creates NuGet packages from your projects.

Setting the NuGet Properties

If no version is specified in the project file, projects are built using the assembly version as the version prefix. To set the value in the project file, add the `Version` tag to the `PropertyGroup`. The following code block shows this and additional properties that can be set for NuGet packages:

```
<PropertyGroup>
  <TargetFramework>netstandard1.6</TargetFramework>
  <Version>1.0.0</Version>
  <PackageProjectUrl>http://www.apress.com</PackageProjectUrl>
  <Authors>Phil Japikse</Authors>
  <Description>Models for the SpyStore Data Access Library</Description>
  <PackageReleaseNotes>Initial Release</PackageReleaseNotes>
  <Copyright>Copyright 2016 (c) Phil Japikse. All rights reserved.</Copyright>
  <PackageTags>SpyStore Models DAL</PackageTags>
</PropertyGroup>
```

For more information on the properties that can be set for NuGet packages, see the documentation here: <https://docs.nuget.org/ndocs/schema/msbuild-targets#pack-target>.

Creating the NuGet Packages

The packages will be created in the `bin\debug` directory by default. To change this, add the `-o` parameter to the `pack` command. The `-version-suffix` parameter adds values after the first three number groups (1.0.0 in this example). If the values added are letters, the package is considered pre-release by NuGet. The following command will create the packages version identifier of 1.0.0 in the `nugetpackages` directory of the solution. Be sure to execute the command in the same directory as the project files for both the `SpyStore.Models` and `SpyStore.DAL` projects.

```
dotnet pack -o ..\nugetpackages
```

These packages will be used in the next chapter.

Summary

This chapter completed the data access layer for the remaining chapters. The chapter began discussing navigation properties and their role in EF model classes. That was followed with a brief discussion of handling display names, stored procedures, and user defined functions. With a firm understanding of navigation properties, the rest of the model classes were created and fine-tuned using data annotations and the Fluent API. Then the database was updated using an EF migration.

The next section demonstrated adding a stored procedure and user defined function to the database using an EF migration. After the objects were created in the database, a computed column, based on the function, was added to the `Orders` table using a property on the `Order` class and the Fluent API.

The final classes added to the object model were the view models that combine different model classes into one for convenience (these will be used in later chapters). With the entire model and view models in place, the data access layer was completed by updating the `CategoryRepo` class and adding all of the remaining repository classes. The last section showed an example of seeding the data for purposes of testing.

You now know what you need to know to use Entity Framework Core in a vast majority of line of business applications. However, even as long as these two chapters are, they really just scratch the surface of everything that EF Core offers.

CHAPTER 3



Building the RESTful Service with ASP.NET Core MVC Services

Now that you have completed the data access layer for the SpyStore e-commerce site, it's time to create the RESTful service that exposes the data operations. The service will serve as the backend for all of the UI frameworks used in the rest of this book, and is built using ASP.NET Core MVC. Before building the service layer, it's important to briefly introduce the Model View Controller (MVC) pattern.

Introducing the MVC Pattern

The Model-View-Controller (MVC) pattern has been around since the 1970s, originally created as a pattern for use in Smalltalk. The pattern has made a resurgence recently, with implementations in many different and varied languages, including Java (Spring Framework), Ruby (Ruby on Rails), and .NET (ASP.NET MVC). The pattern is very simple, composed of the three pieces that compose the name.

The Model

The *model* is the data of your application. The data is typically represented by Plain Old CLR Objects (POCOs), as used in the data access library created in the previous two chapters. View models are composed of one or more models and shaped specifically for the view that uses them. Think of models and view models as database tables and database views.

Academically, models should be extremely clean and not contain validation or any other business rules. Pragmatically, whether or not models contain validation logic or other business rules entirely depends on the language and frameworks used, as well as specific application needs. For example, EF Core contains many data annotations that double as a mechanism for shaping the database tables and a means for validation in Core MVC. In this book, the examples focus on reducing duplication of code, which places data annotations and validations where they make the most sense.

The View

The *View* is the UI of the application. Views accept commands and render the results of those commands to the user. The view should be as lightweight as possible, and not actually process any of the work, but hand off all work to the controller.

The Controller

The *controller* is the brains of the operation. Controllers take commands/requests from the user (via the View), marshal them to the appropriate service, and then send the application data to the view. Controllers should also be lightweight and leverage other components to maintain Separation of Concerns (SoC).

Introducing ASP.NET Core MVC Web API

Microsoft released ASP.NET MVC in 2007 to great success. Building on that success (and sharing a lot of the MVC framework code), the ASP.NET Web API framework was first released alongside MVC 4 with Visual Studio 2012. The framework started gaining traction almost immediately, and Microsoft released ASP.NET Web API 2 with Visual Studio 2013, and then updated the framework to version 2.2 with Visual Studio 2013 Update 1.

ASP.NET Web API from the beginning was designed to be a service based framework for building RESTful (**R**epresentational **S**tate **T**ransfer) services. It is essentially the MVC framework minus the “V” (view) with optimizations for creating headless services. These services can be called by any technology, not just those under the Microsoft umbrella. Calls to a Web API service are based on the core HTTP verbs (Get, Put, Post, Delete) through a URI (Uniform Resource Identifier) such as the following:

<http://www.mysite.com:33826/api/category>

If this looks like a URL (Uniform Resource Locator), it’s because it is! A URL is simply a URI that points to physical resource on a network.

Calls to Web API use the HTTP (**H**ypertext **T**ransfer **P**rotocol) scheme on a particular host (in this example www.mysite.com), on a specific port (33816), followed by the path (`api/category`) and optional query and fragment (not shown in this example). Note that the port for HTTP calls defaults to 80 if one isn’t specified. Web API calls can also include text in the body of the message, as you will see later in this chapter.

ASP.NET Core MVC (formally called ASP.NET 5) unified the Web API and MVC into one framework, named ASP.NET Core MVC. The framework formerly known as the Web API is now just included in Core MVC and not a separate framework. However, the Visual Studio template for creating RESTful services with Core MVC Services is still named “Web API”. Therefore, in this book I refer to the service capabilities within Core MVC as the either Core MVC Services or just simply Web API.

ASP.NET Core and .NET Core

Just as Entity Framework Core is a complete rewrite of Entity Framework 6, ASP.NET Core is a rewrite of the popular ASP.NET framework. Rewriting ASP.NET was no small task, but necessary in order to remove the dependency on `System.Web`. Removing the dependency on `System.Web` enables ASP.NET applications to run on operating systems other than Windows, and other web servers besides Internet Information Services (IIS). If the ability to run ASP.NET applications on other operating systems isn’t enough motivation to use ASP.NET Core, the rewrite brings many other benefits, including a significant performance boost.

Moving away from `System.Web` and IIS opened the door for ASP.NET Core applications to use a cross-platform, light-weight, and open source web server called Kestrel. Kestrel presents a uniform development experience across all platforms. Windows developers, have no fear! ASP.NET Core applications fully support Windows deployments, and when using Visual Studio to develop ASP.NET Core applications, IIS Express is one of the options for debugging.

Like EF Core, ASP.NET Core is being developed on GitHub as a completely open source project (<https://github.com/aspnet>). It is also designed as a modular system of NuGet packages. Developers only install the features that are needed for a particular application, minimizing the application footprint, reducing the overhead, and decreasing security risks. Additional improvements include a simplified startup, built-in dependency injection, a cleaner configuration system, and pluggable middleware.

■ **Note** One chapter isn't nearly enough space to completely cover ASP.NET Core MVC services. I am only going to cover the features needed to implement the service layer for the SpyStore sample sites.

Dependency Injection

Dependency Injection (DI) is a mechanism to support loose coupling between objects, instead of directly creating dependent objects or passing specific implementations into methods, classes, and methods program to interfaces. That way any implementation of the interface can be passed into methods and classes, dramatically increasing the flexibility of the application.

DI support is one of the main tenets in the rewrite ASP.NET Core. Not only does the `Startup` class (covered later in this chapter) accept all of the configuration and middleware services through dependency injection, your custom classes can (and should) be added to the service container to be injected into other parts of the application.

The ASP.NET Core DI container is typically configured in the `ConfigureServices` method of the `Startup` class using the instance of `IServiceCollection` that itself is injected in. When an item is configured into the DI container, there are four lifetime options, as shown in Table 3-1.

Table 3-1. *Lifetime Options for Services*

Lifetime Option	Functionality Provided
Transient	Created each time they are needed.
Scoped	Created once for each request. Recommended for Entity Framework <code>DbContext</code> objects.
Singleton	Created once on first request and then reused for the lifetime of the object. This is the recommended approach vs. implementing your class as a <code>Singleton</code> .
Instance	Similar to <code>Singleton</code> , but created when <code>Instance</code> is called vs. on first request.

More information on DI and ASP.NET Core can be found in the documentation here:

<https://docs.microsoft.com/en-us/aspnet/core/fundamentals/dependency-injection>

Determining the Runtime Environment

Another improvement in .NET Core is the new environment checking capability. ASP.NET Core looks for an environment variable named `ASPNETCORE_ENVIRONMENT` to determine the current running environment. The environment variable value is usually read from a settings file, the development tooling (if running in debug mode), or through the environment variables of the machine executing the code. The names can be anything, but convention usually defines the states as `Development`, `Staging`, and `Production`.

Once the environment is set, there are many convenience methods available to developers to take advantage of the setting. For example, the following code enables logging for the development environment, but not for the staging or production environments:

```
if (env.IsDevelopment())
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();
}
```

For more information on ASP.NET environments, reference the help at <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/environments>.

Routing

Routing is how Core MVC matches URL requests to controllers and actions in your application, instead of the old Web Forms process of matching URLs to the project file structure. It also provides a mechanism for creating URLs based on Controllers and Actions, instead of hard coding them into your application.

URL Templates

Routes are URL patterns comprised of literals, tokens, and variable placeholders. Tokens are one of [area], [controller], and [action] values. These correspond to Core MVC areas, controller classes, and controller methods. Controllers and actions are covered in the next section. Areas are sub sites within an MVC site, and are beyond the scope of this book.

Variable placeholders are listed in braces, such as {id}. Adding a question mark (such as {id?}) indicates the parameter is optional. Take the example route:

```
api/[controller]/[action]/{id?}
```

In the aforementioned route, api is a literal value. The next two values in the path are looking for a named controller and action. The final parameter is optional, and if it's found in the URL, it's passed into the action method as a parameter of the same name. For example, both of the following URLs map to the ValuesController (the Controller suffix is dropped) and Get action method. The second URL passes the value of 5 to the parameter named id on the action method:

```
http://www.mysite.com/api/values/get
http://www.mysite.com/api/values/get/5
```

Attribute Routing

In Core MVC, attribute routing is the preferred method for defining routes. Prior versions of MVC used a Route Table (and Core MVC UI projects still do for the default route), which could become difficult to troubleshoot as the site grows more complicated. Attribute routing places the route information in the controller and action methods, which enhances the clarity of the site. Attribute routing without a configured route table also increases the amount of configuration that needs to be done, since every controller and action need to have routing information specified.

For example, the default ValuesController has the Route attribute that defines a literal value and a token.

```
[Route("api/[controller]")]
public class ValuesController : Controller
{
    //omitted for brevity
}
```

One could argue that using attribute routing leads to duplication of code, since each of the controllers would need to have the same route defined, instead of placing the core route into the Route Table. While that is true, the benefits of having the routes defined right next to the code that executes them is of value as well. I'll leave it to the reader to decide the best course of action. When you build the Core MVC UI, you will see the Route Table in action.

Named Routes

Routes can also be assigned a name. This creates a shorthand method for redirecting to a particular route just by using the name. For example, the following route attribute has the name of `GetOrderDetails`:

```
[HttpGet("{orderId}", Name = "GetOrderDetails")]
```

To refer to this route in code, the following code can be used:

```
return CreatedAtRoute("GetOrderDetails", routeValues: new { controller = "Orders",
customerId = customerId, orderId = orderId }, value: orderId);
```

Responding to the Type of HTTP Request

Routing is just part of the story regarding which action method to call. Core MVC Web API analyzes the verb used in the request (one of `HttpGet`, `HttpPost`, `HttpPut`, and `HttpDelete`) to determine the final leg of the route. The HTTP verb handled by a particular action method is indicated with a `.NET` attribute, which can also contain route information. For example, the following action responds to a HTTP Get request and adds a parameter (`id`) to the request:

```
[HttpGet("{id}")]
public string Get(int id)
{
    return "value";
}
```

In Core MVC, any action *without* an HTTP attribute (such as `HttpPost`) will be executed as a Get operation.

■ **Note** Previous versions of the Web API included a shortcut for verb assignment. If the prefix of the action method name matched `Put`, `Create`, or `Delete`, the routing would respond to the corresponding HTTP verbs. Fortunately, this convention was removed in the Core MVC, as it was just too much secret sauce. Just like many of the EF conventions discussed in Chapters 1 and 2, I believe it's better to be explicit than to rely on "magic" when writing code.

Creating the Solution and the Core MVC Project

Create a new Visual Studio solution. Select the ASP.NET Core Web Application (.NET Core) template available under `Installed > Templates > Visual C# > .NET Core`. Name the project and solution `Spystore.Service` and click OK, as shown in Figure 3-1.

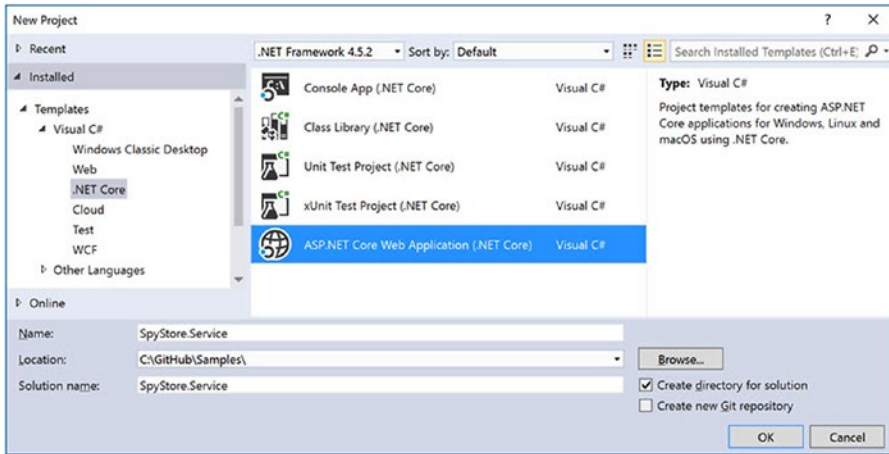


Figure 3-1. Adding a new ASP.NET web application

Note If you have not already installed Visual Studio 2017 and .NET Core, follow the steps outlined in Chapter 1.

The new screen shows the Long Term Support (LTS) templates (ASP.NET Core Templates 1.0) and the current templates (ASP.NET Core Templates 1.1). Select the Web API template under ASP.NET Core Templates 1.1, leave the Enable Container (Docker) Support check box unchecked, and leave the Authentication set to No Authentication, as shown in Figure 3-2.

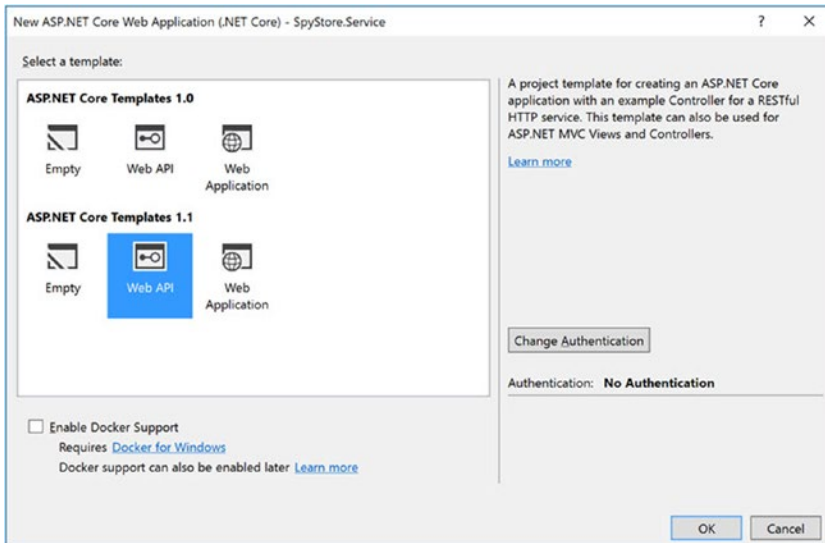


Figure 3-2. Selecting the Web API project template

■ **Note** Docker is a containerization technology that is beyond the scope of this book. More information can be found here: <https://docs.microsoft.com/en-us/azure/vs-azure-tools-docker-hosting-web-apps-in-docker>. Security is a very large topic, and there isn't room in this chapter (or book) to cover it properly. We (the author team) decided to ignore it completely. We feel very strongly about security and didn't want to do an injustice by only partially covering the topic. You need to assess the security needs of your application and implement the appropriate level. Information on ASP.NET Core security is available in the official documentation: <https://docs.microsoft.com/en-us/aspnet/core/>.

Click OK, and the new solution and project are created. Your solution and projects should look like Figure 3-3.

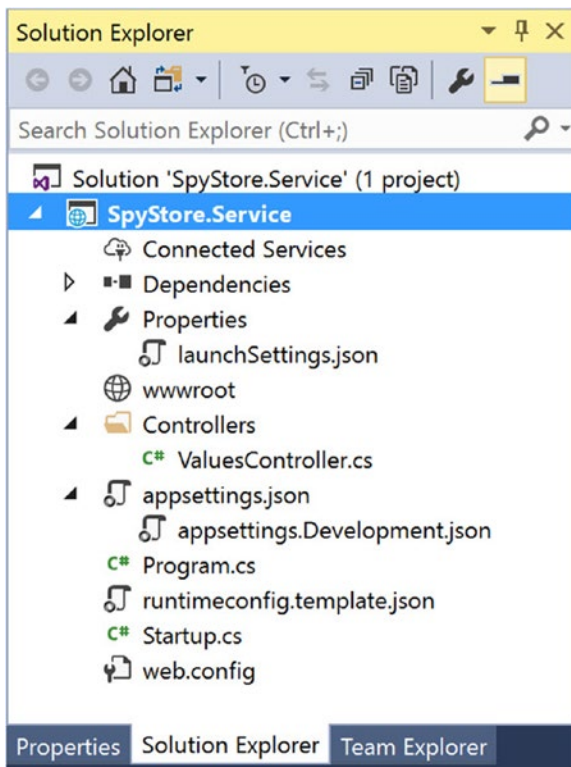


Figure 3-3. The generated files and folders for a Core MVC Web API app

Compared to Web API 2.2 projects, the Core MVC Web API template is very lightweight. ASP.NET Core projects have been simplified to the point that they are now console applications that have different services added in, such as listening for HTTP traffic.

■ **Note** The templates included with VS2017 RC3 do not create the content in the `src` directory like in prior versions. The project structure has been flipping back and forth between using the `src` directory and not using it with practically each VS2017 release. I personally like the organization with the `src` directory, and the code samples are structured using it. The images and text match the templates provided with VS2017 RC3.

Adding the Package Source for the Data Access Layer

The NuGet packages for the `SpyStore.DAL` and `Spystore.Models` projects need to be added as package references. They were created in a local directory and not published to NuGet.org, so the local directory needs to be added as a package source. In Visual Studio, select **Tools** ► **Options**, and then select **NuGet Package Manager** ► **Package Sources** in the left side of the Options dialog. On the right side, click the green plus sign, then add a name and the location of the packages created in Chapter 2, similar to Figure 3-4 (your location will most likely be different). Click **Update** and then **OK** to add the new package source.

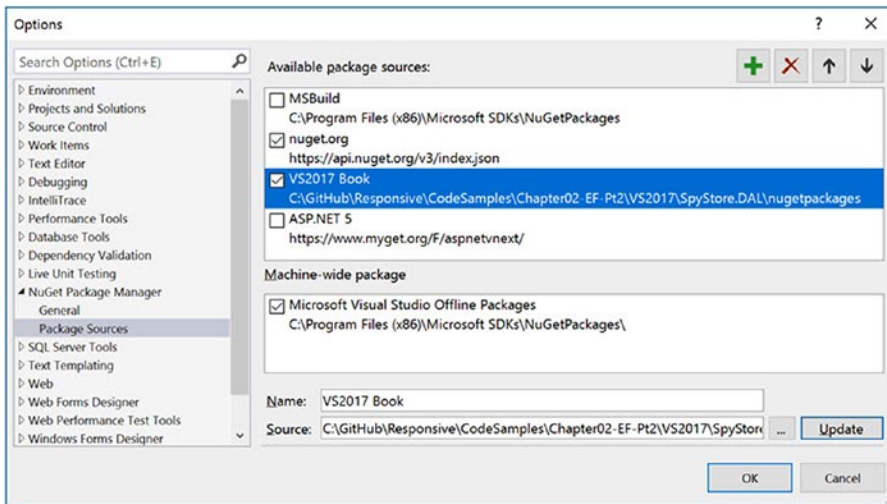


Figure 3-4. Creating a local package source

Updating and Adding NuGet Packages

Right-click on the `SpyStore.Service` project and select **Manage NuGet Packages**. Make sure the **Include Prerelease** check box is checked. Click on **Updates** at the top of the NuGet Package Manager screen, check the **Select All Packages** check box, and click on the **Update** button. Unless you checked the **Don't Show This Message Again** check box, you might get prompted to confirm the installs and accept license agreements. Just click **OK** on the dialogs to install the updated packages.

■ **Note** Depending on when you are reading this and the latest versions of the VS2017 templates, there might not be any packages to update. At the time of this writing, there was only one update available, and that was for `Microsoft.ApplicationInsights.AspNetCore`, which has now been promoted to version 2.0.0 from a beta release.

Next, click on Browse, enter `SpyStore` into the search box, and change the Package Source to All (which includes the local package source just created). Install the `SpyStore.Models` package and then the `SpyStore.DAL` package, in that order. Unfortunately, due to some kinks that still need to be worked out with the integration between the .NET Core CLI and NuGet packaging, installing the `SpyStore.DAL` package first fails because the `SpyStore.Models` project doesn't get pulled in correctly. Hopefully, by the time you are reading this, all of the tooling is fully released and working correctly.

Finally, install the Entity Framework Core packages. Enter `Microsoft.EntityFrameworkCore` in the search box and install the following:

- `Microsoft.EntityFrameworkCore` (v1.1.1)
- `Microsoft.EntityFrameworkCore.SqlServer` (v1.1.1)

The ASP.NET Core “Super” Packages

Readers who have worked with earlier editions of VS 2017 and .NET Core might notice that there are a lot fewer packages referenced by the `SpyStore.Service` project. This is due to the template taking advantage of what I am calling the two Core MVC “super” packages that are included in the Core MVC Service template: `Microsoft.AspNetCore` and `Microsoft.AspNetCore.Mvc`.

The `Microsoft.AspNetCore` super package brings in the following additional packages:

- `Microsoft.AspNetCore.Diagnostics` (v1.1.1)
- `Microsoft.AspNetCore.Hosting` (v1.1.1)
- `Microsoft.AspNetCore.Routing` (v1.1.1)
- `Microsoft.AspNetCore.Server.IISIntegration` (v1.1.1)
- `Microsoft.AspNetCore.Server.Kestrel` (v1.1.1)
- `Microsoft.Extensions.Configuration.EnvironmentVariables` (v1.1.1)
- `Microsoft.Extensions.Configuration.FileExtensions` (v1.1.1)
- `Microsoft.Extensions.Configuration.Json` (v1.1.1)
- `Microsoft.Extensions.Logging` (v1.1.1)
- `Microsoft.Extensions.Logging.Console` (v1.1.1)
- `Microsoft.Extensions.Options.ConfigurationExtensions` (v1.1.1)

The `Microsoft.AspNetCore.Mvc` super package brings in the following additional packages:

- `Microsoft.AspNetCore.Mvc.ApiExplorer` (v1.1.2)
- `Microsoft.AspNetCore.Mvc.Cors` (v1.1.2)
- `Microsoft.AspNetCore.Mvc.DataAnnotations` (v1.1.2)
- `Microsoft.AspNetCore.Mvc.Formatters.Json` (v1.1.2)

- Microsoft.AspNetCore.Mvc.Localization (v1.1.2)
- Microsoft.AspNetCore.Mvc.Razor (v1.1.2)
- Microsoft.AspNetCore.Mvc.TagHelpers (v1.1.2)
- Microsoft.AspNetCore.Mvc.ViewFeatures (v1.1.2)
- Microsoft.Extensions.Caching.Memory (v1.1.1)
- Microsoft.Extensions.DependencyInjection (v1.1.0)

This greatly simplifies the required package list for Core MVC packages.

■ **Note** Super packages is not the official name, but something I made up because it seems more fitting than the official name. The official name used by the project team is *meta-package*, but I think super package is more descriptive, so it's what I'll be using in this book.

MVC Projects and Files

Web API projects in Core MVC have removed a lot of the cruft from previous versions. The trimmed down project basics are explained in the following sections. The new project file format is covered in Chapter 1, so I won't repeat it here.

The Program.cs File

Core MVC applications are simply console applications. The Main method creates a `WebHostBuilder` that sets the application up to listen for (and handle) HTTP requests.

The following is the default Main method created by the Visual Studio Web API template:

```
public class Program
{
    public static void Main(string[] args)
    {
        var host = new WebHostBuilder()
            .UseKestrel()
            .UseContentRoot(Directory.GetCurrentDirectory())
            .UseIISIntegration()
            .UseStartup<Startup>()
            .UseApplicationInsights
            .Build();
        host.Run();
    }
}
```

The preceding code demonstrates the modularity of ASP.NET Core and the opt-in nature of its services. After creating a new instance of the `WebHostBuilder`, the code enables just the features that the application needs. The Kestrel web server is enabled, as is IIS. The root directory for content is set as the project directory. The `UseStartup` method sets the class that configures the HTTP pipeline and the rest of the application to the `Startup` class (covered next). The next line adds in support for `ApplicationInsights`.

ApplicationInsights is a way to instrument and monitor applications. While not covered in this book, you can get the full details here: <https://azure.microsoft.com/en-us/services/application-insights/>. The next line builds all of the features into the WebHostBuilder, and the final line runs the web host.

By default, Kestrel listens on port 5000. To change the target port to 40001 (for example), add the following line to the WebHostBuilder (after the UseKestrel call):

```
.UseUrls("http://*:40001/")
```

This changes the Kestrel configuration for this application to listen on port 40001. Update the code to match this:

```
var host = new WebHostBuilder()
    .UseKestrel()
    .UseUrls("http://*:40001/")
    .UseContentRoot(Directory.GetCurrentDirectory())
    .UseIISIntegration()
    .UseStartup<Startup>()
    .UseApplicationInsights
    .Build();
host.Run();
```

The appsettings.json File(s)

Previous versions of ASP.NET used the web.config file to configure services and applications, and developers accessed the configuration settings through the System.Configuration class. Of course, *all* configuration settings for the site, not just application specific settings, were dumped into the web.config file, making it a (potentially) complicated mess.

ASP.NET Core introduces a greatly simplified configuration system. It's based on simple JSON files that hold configuration settings as name/value pairs. The default file for configuration is the appsettings.json file. The initial version of appsettings.json file created by the VS2017 template simply contains configuration information for the logger (created in the Startup.cs class) and is listed here:

```
{
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Warning"
    }
  }
}
```

The Core MVC Web API template also creates an appsettings.Development.json file. As you will see shortly, the configuration system uses appsettings.{environmentname}.json files for runtime environment specific settings. The default file changes the logging levels if the environment is set to Development. The file contents are shown here:

```
{
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Debug",
```

```

    "System": "Information",
    "Microsoft": "Information"
  }
}
}

```

Application settings like connection strings are typically stored in the `appsettings.json` file. Add your machine specific connection string to the `appsettings.json` file by updating the text to the following (this should match the connection string used in Chapters 1 and 2):

```

{
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Debug",
      "System": "Information",
      "Microsoft": "Information"
    }
  },
  "ConnectionStrings": {
    "SpyStore":
"Server=(localdb)\MSSQLLocalDB;Database=SpyStore;Trusted_Connection=True;MultipleActiveResultSets=true"
  }
}

```

This setting is used later in the chapter.

The `runtimeconfig.template.json` File

If you've been following along with the releases of .NET Core, you've noticed a significant amount of churn. The latest change that came with VS2017 RC3 is the inclusion of the `runtimeconfig.template.json` file into the Core MVC Services template. This file is used for setting runtime properties (as the name suggests). The same settings were in the `*.csproj` file, but have been relocated here to make the MSBuild process smoother. Open the file and you will see just one setting enabled by default:

```

{
  "configProperties": {
    "System.GC.Server": true
  }
}

```

You don't need to make any changes to the file for this project.

The `Startup.cs` File

The `Startup` class configures how the application will handle HTTP requests and response, initiates the configuration system, and sets up the dependency injection container. The class name can be anything, as long as it matches the `UseStartup<T>` line in the configuration of the `WebHostBuilder`, but the convention is to name the class `Startup.cs`.

Available Services for Startup

The startup process needs access to framework and environmental services and values, and these are injected into the Startup class. There are five services available to the Startup class for configuring the application, listed in Table 3-2. The following sections show these services in action.

Table 3-2. Available Services in Startup

Service	Functionality Provided
IApplicationBuilder	Defines a class that provides the mechanisms to configure an application's request pipeline.
IApplicationEnvironment	Provides access to common application information, such as ApplicationName and ApplicationVersion.
IHostingEnvironment	Provides information about the web hosting environment an application is running in.
ILoggerFactory	Used to configure the logging system and create instances of loggers from the registered logging providers.
IServiceCollection	Specifies the contract for a collection of service descriptors. Part of the Dependency Injection framework.

The constructor can take IHostingEnvironment and ILoggerFactory, although typical implementations only use the IHostingEnvironment parameter. The Configure method must take an instance of IApplicationBuilder, but can also take instances of IHostingEnvironment and/or ILoggerFactory. The ConfigureServices method takes an instance of IServiceCollection.

The base Startup class with the services injected in looks like this (parameters in bold are required):

```
public class Startup
{
    public Startup(IHostingEnvironment env, ILoggerFactory loggerFactory)
    {
    }
    public IConfigurationRoot Configuration { get; }
    // This method gets called by the runtime. Use this method to add services to the
    container.
    public void ConfigureServices(IServiceCollection services)
    {
    }
    // This method gets called by the runtime.
    //Use this method to configure the HTTP request pipeline.
    public void Configure(
        IApplicationBuilder app,
        IHostingEnvironment env,
        ILoggerFactory loggerFactory)
    {
    }
}
```

Add a private variable of type `IHostingEnvironment` to the top of the class. This will be set in the constructor so it can be used in the `ConfigureServices` method. More on this later. For now add the following line:

```
private IHostingEnvironment _env;
```

The Constructor

The ASP.NET Core configuration system uses the `ConfigurationBuilder` class in conjunction with JSON formatted files. The accepted pattern (and how the default VS2017 template is configured) for ASP.NET Core MVC is to create a new `ConfigurationBuilder` in the constructor of the `Startup` class and assign the resulting `IConfigurationRoot` to a public property on the class.

To get started, open the `Startup.cs` class and examine the following code in the constructor (I changed the `var` keyword to `IConfigurationBuilder` to the type specific variable declaration):

```
public Startup(IHostingEnvironment env)
{
    IConfigurationBuilder builder = new ConfigurationBuilder()
        .SetBasePath(env.ContentRootPath)
        .AddJsonFile("appsettings.json", optional: true, reloadOnChange: true)
        .AddJsonFile($"appsettings.{env.EnvironmentName}.json", optional: true)
        .AddEnvironmentVariables();
    Configuration = builder.Build();
}
```

The `SetBasePath` method sets the location for the JSON files. This environment property was set in the `Main` method with the following line of code:

```
UseContentRoot(Directory.GetCurrentDirectory())
```

The `AddJsonFile` provider takes one parameter—the location and name of the file that holds the JSON configuration information. In this example, the file used is the `appsettings.json` file located in the project content root directory, so the path is not necessary. If the file isn't found, the method does nothing, and the project watches the file for changes. If changes are detected, Core MVC reloads the settings from the updated file. The second call to `AddJsonFile` looks for a JSON file with the environment value in the name, such as the `appsettings.development.json` included with the default template. If such a file exists, the settings contained in the file are loaded and override any matching properties from the previous file.

The final line adds any environment variables to the configuration. The order of adding providers matters. The last setting configured wins if a setting is found in multiple locations. For example, if the deployment machine has an environment variable named `ASPNETCORE_ENVIRONMENT`, the machine setting overrides the values from the JSON files.

This makes deploying applications up the environment chain much simpler. Simply change environment specific values (such as database connection strings) between `appsettings.development.json`, `appsettings.staging.json`, and `appsettings.production.json` files and the project will be automatically configured. After the providers have been added to the `IConfigurationBuilder`, the next line builds the configuration and sets the `Configuration` property of the class.

Once the configuration is built, settings can be accessed using the traditional Get family of methods (GetSection, GetValue, etc.) and default indexer properties (key/value pairs). There is also a shortcut for getting application connection strings. You will see this in action in the ConfigureServices method covered later in the chapter:

```
Configuration.GetConnectionString("SpyStore")
```

The final line to add to the constructor sets the `_env` variable to the `IHostingEnvironment` that was injected into the constructor, as follows:

```
_env = env;
```

The Configure Method

The Configure method is used to set up the application to respond to HTTP requests. The default template sets up logging and enables MVC services, shown in the following code:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory
loggerFactory)
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();
    app.UseMvc();
}
```

The first two lines set up logging for the app. The enables logging for all environments. To turn off logging for non-development environments, update the code to the following:

```
if (env.IsDevelopment())
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();
}
```

■ **Note** Logging is not covered in this book. The default settings enable logging when running in debug mode and outputs the logged information to the console window. For more information, check out the ASP.NET docs, available here: <https://docs.microsoft.com/en-us/aspnet/core/fundamentals/logging>.

The last line sets up all of the services needed to run an MVC application, including attribute routing.

If the environment is Development, the app should initialize the data in the database using the `StoreDataInitializer` created in the data access library. The next line gets the application's `IServiceProvider` (configured in the `ConfigureServices` method, covered next) to pass into the `InitializeData` override. To do this, start by adding a using statement for the data initializer:

```
using SpyStore.DAL.Initializers;
```

Then add the following code just before the `app.UseMvc` line:

```
if (env.IsDevelopment())
{
    using (var serviceScope =
        app
            .ApplicationServices
            .GetRequiredService<IServiceScopeFactory>()
            .CreateScope())
    {
        StoreDataInitializer.InitializeData(app.ApplicationServices);
    }
}
```

The final code to add into the `Configure` method is Cross-Origin Requests (CORS) support. This is needed by JavaScript frameworks used later in the book. For more information on CORS support, refer to the document article here: <https://docs.microsoft.com/en-us/aspnet/core/security/cors>.

Add the following code before the `app.UseMvc` line:

```
app.UseCors("AllowAll"); // has to go before UseMvc
```

The ConfigureServices Method

The `ConfigureServices` method is used to configure any services needed by the application and insert them into the dependency injection container.

Configuring MVC

The default template only has one line, which adds MVC to the services collection:

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddMvc();
}
```

There is a streamlined version of the MVC services that only adds the services necessary to create a Core MVC Services application. Change the `AddMvc` line to `AddMvcCore`, as follows:

```
public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddMvcCore();
}
```

Configuring JSON Formatting

Prior versions of ASP.NET Web API (and the early versions of Core MVC) returned JSON with initial capital letters (called *Pascal casing*). ASP.NET Core changed the response format to initial small letters (called *camel casing*) when .NET Core and Core MVC went RTM. For example, this is the JSON returned from the RTM version of Core MVC:

```
[{"categoryName":"Communications","products":[],"id":0,"timeStamp":"AAAAAABaBA="}]
```

To revert to Pascal casing, a `JsonFormatter` needs to be configured for the `MvcCore` services. Add the following using statements:

```
using Newtonsoft.Json;
using Newtonsoft.Json.Serialization;
```

Next, replace the `AddMvcCore` code with the following:

```
services.AddMvcCore()
    .AddJsonFormatters(j =>
    {
        j.ContractResolver = new DefaultContractResolver();
        j.Formatting = Formatting.Indented;
    });
```

The first formatter reverts the results into Pascal casing. The second adds indented formatting to the output. The very same service call returns the following result:

```
[
  {
    "CategoryName": "Communications",
    "Products": [],
    "Id": 0,
    "TimeStamp": "AAAAAABaBA="
  }
]
```

Configuring CORS

The next step is to configure CORS. Add the following code after the `AddMvcCore` method call:

```
services.AddCors(options =>
{
    options.AddPolicy("AllowAll", builder =>
    {
        builder.AllowAnyHeader().AllowAnyMethod().AllowAnyOrigin().AllowCredentials();
    });
});
```

Configuring EF Core

EF Core is configured using the `IServiceCollection` as well. Add the following using statements:

```
using Microsoft.EntityFrameworkCore;
using SpyStore.DAL.EF;
```

Next, add the following code after the call to `AddDbContext`:

```
services.AddDbContext<StoreContext>(
    options => options.UseSqlServer(Configuration.GetConnectionString("SpyStore")));
```

The `AddDbContext` call adds the `StoreContext` to the DI Framework, passing an instance of `DbContextOptions` into the constructor along with the connection string retrieved through the configuration framework. Application configuration is covered later in the chapter.

Configuring the Dependency Injection Container

The next block of code will add all of the data access library repositories into the DI container. Add the following using statements to the top of the file:

```
using SpyStore.DAL.Repos;
using SpyStore.DAL.Repos.Interfaces;
```

Add the following services, scoped at the request level.

```
services.AddScoped<ICategoryRepo, CategoryRepo>();
services.AddScoped<IProductRepo, ProductRepo>();
services.AddScoped<ICustomerRepo, CustomerRepo>();
services.AddScoped<IShoppingCartRepo, ShoppingCartRepo>();
services.AddScoped<IOrderRepo, OrderRepo>();
services.AddScoped<IOrderDetailRepo, OrderDetailRepo>();
```

The Controllers Folder

Controllers are classes, and actions are methods contained in controller classes. By convention, controllers are located in the `Controllers` folder, and the default template creates a sample controller named `ValuesController`. Delete the `ValuesController` file.

Controllers will be covered in detail later in this chapter, but for now, just understand that this is where Controller classes are stored, and by convention they are named ending with “Controller” (e.g., `ValuesController`).

The wwwroot Folder

This folder is not used in Core MVC Web API. It plays a vital role in Core MVC UI applications and is covered in Chapter 5.

The web.config File

There is a single `web.config` file, and it is used to add support for debugging in Visual Studio using IIS Express. No changes will be necessary in this file.

The launchsettings.json File

The `launchsettings.json` file (located in the Properties node in Solution Explorer) is the backing store for the project properties UI. Any changes to the UI are written to the file, and any changes written to the file update the UI. These settings are only used when running the project from within Visual Studio. Running from the command line with `dotnet run` ignores these settings.

The `launchsettings.json` file is listed here for reference:

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:50018/",
      "sslPort": 0
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "launchUrl": "api/values",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "SpyStore.Service": {
      "commandName": "Project",
      "launchBrowser": true,
      "launchUrl": "api/values",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      },
      "applicationUrl": "http://localhost:50019"
    }
  }
}
```

The `iisSettings` section defines the settings for running the application on IIS Express. The most important settings to note are `applicationUrl`, which defines the port, and the `environmentVariables` block, which defines the runtime environment. The `profiles` section provides named debugging profiles to Visual Studio and determines if a browser is launched with the debugger, the environment, and the initial URL. The profiles show in the Run command in Visual Studio, as shown in Figure 3-5.

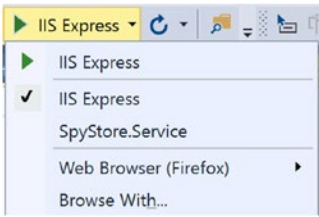


Figure 3-5. The available debugging profiles

In this sample file, pressing F5 with the IIS Express profile opens a browser to the following:

`http://localhost:50018/api/values`

The `SpyStore.Service` profile runs using Kestrel. This profile has its own application URL. When this profile is selected, the Visual Studio debugger launches a browser with this URL:

`http://localhost:50019/api/values`

Controllers and Actions

Controllers and actions are the workhorses of a MVC Web API application.

Controllers

A *controller* is a class that inherits from the abstract `Controller` class or ends in the name `Controller`. Inheriting from the base `Controller` class is the most efficient way to develop a controller due to the helper methods contained in the base class. As mentioned earlier, convention dictates that MVC controllers are located in the `Controllers` directory of the project. When referring to controller in routes, the `Controller` suffix is dropped. For example, the following URL will route to the `ValuesController`:

<http://www.mysite.com/api/values/get>

Actions

Actions are simply methods on controllers. Actions can take parameters, and while they can return anything, they typically return an `IActionResult` (`Task<IActionResult>` for async operations) or a class that implements `IActionResult`. The most commonly used classes that implement `IActionResult` are described in the following sections.

HTTP Status Code Results

The base `Controller` class contains helper methods for returning HTTP Status Codes as action results. Table 3-3 lists some of the most commonly used helper methods that return HTTP Status Code Results.

Table 3-3. Some of the Helper Methods Provided by the Base Controller Class

Controller Helper Method	HTTP Status Code	Action Result	Raw Code Equivalent
NoContent		NoContentResult	204
Ok		OkResult	200
NotFound		NotFoundResult	404
BadRequest		BadRequestResult	400
Created		CreatedResult	201
CreateAtAction		CreatedAtActionResult	
CreatedAtRoute		CreateAtRouteResult	
Accepted		AcceptedResult	202
AcceptedAtAction		AcceptedAtActionResult	
AcceptedAtRoute		AcceptedAtRouteResult	

As an example, the following code returns a 404 HTTP result when the record isn't found:

```
[HttpGet("{id}", Name = "GetCategory")]
public IActionResult GetById(int id)
{
    var item = Repo.Find(id);
    if (item == null)
    {
        return NotFound();
    }
    return Json(item);
}
```

Formatted Response Results

To return an object (or object collection) formatted as JSON, wrap the object(s) in the `Json` object (or an `ObjectResult` object), which creates a `JsonResult`. An example of this is as follows:

```
[HttpGet]
public IActionResult GetAll()
{
    return Json(Repo.GetAll());
}
```

Actions that return a type (or a strongly typed collection) do the same as the previous method, thanks to the Core MVC framework. An alternative way to write the preceding action is as follows:

```
[HttpGet]
public IEnumerable<Category> GetAll()
{
    return Repo.GetAll();
}
```

Formatted response results can be combined with HTTP Status Codes by wrapping the object result in one of the status code action results, as follows:

```
[HttpGet]
public IActionResult GetAll()
{
    return Ok(Repo.GetAll());
}
```

Redirect Results

The `RedirectActionResult` and its siblings (`RedirectToAction`, `RedirectToRoute`) redirects the users to another route. These are not used in the Web API code in this chapter, but will be used when building the Core MVC user interface later in this book.

An Example Controller

If you worked with previous versions of ASP.NET MVC, one of the hallmarks of Visual Studio was the rich scaffolding support in Visual Studio. If you worked with .NET Core in Visual Studio 2015, you were probably disappointed by the lack of scaffolding support. Fortunately, Visual Studio 2017 has brought scaffolding back for .NET Core! One of the available items available for scaffolding is the API controller, This creates a template with basic action methods for all four of the HTTP verbs (Get, Post, Put, and Delete).

Right-click on the Controllers folder, select Add ► New Item, and then select Installed ► Visual C# ► ASP.NET Core on the left side of the screen and Web API Controller Class on the right. Name the new class `CategoryController.cs`, as shown in Figure 3-6.

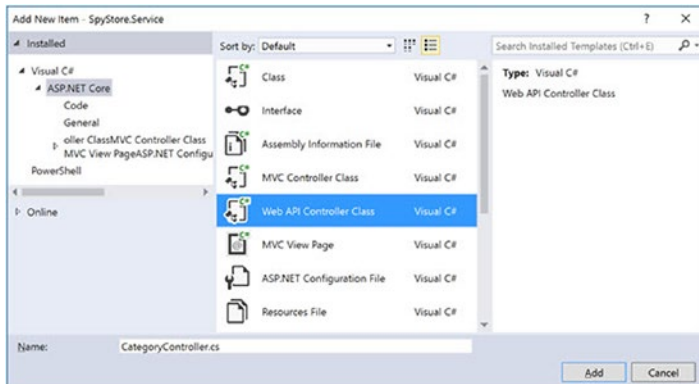


Figure 3-6. Adding the `CategoryController` class

The scaffolded template is shown here:

```
[Route("api/[controller]")]
public class CategoryController : Controller
{
    // GET: api/values
    [HttpGet]
    public IEnumerable<string> Get()
    {
        return new string[] { "value1", "value2" };
    }
}
```



```

// GET api/values/5
[HttpGet("{id}")]
public string Get(int id)
{
    return "value";
}
// POST api/values
[HttpPost]
public void Post([FromBody]string value)
{
}
// PUT api/values/5
[HttpPut("{id}")]
public void Put(int id, [FromBody]string value)
{
}
// DELETE api/values/5
[HttpDelete("{id}")]
public void Delete(int id)
{
}

```

At the top of the class is the route attribute. This defines the route with the literal `api` followed by the controller name. An example URL that would route to this controller is the following (note that the comments in the template incorrectly have the routes defined as `"api/values"`):

<http://www.mysite.com/api/category>

When the preceding URL is called using a HTTP Get command (for example, from a browser), the first Get action executes and returns the following JSON:

```
[
  "value1",
  "value2"
]
```

As mentioned before, the `HttpGet` attribute is optional. The same result would happen if the method was coded without the attribute, as follows:

```
public IEnumerable<string> Get()
{
    return new string[] { "value1", "value2" };
}

```

The second Get method adds the variable place holder `{id}` to the route defined at the controller level. Routes parameters can be added into the HTTP verb attributes, as demonstrated here:

```
[HttpGet("{id}")]
public string Get(int id)
{
    return "value";
}

```

Since HTTP Get is the default, this method could also be written as follows and accomplish the same result:

```
[Route("{id}")]
public string Get(int id)
{
    return "value";
}
```

The final examples to discuss before moving on to writing the real application code are the Post and Put methods. Specifically, the [FromBody] attributes. This attribute indicates that Core MVC should attempt to instantiate an instance of the datatype specified (in these simple examples, the string type) using a process called *model binding*.

Model Binding

Model binding is the process where Core MVC takes the name/value pairs available (in the example above from the request body) and attempts to reconstitute the specified type using reflection. If one or more properties can't be assigned (e.g., due to datatype conversion issues or validation errors), the binding engine sets the ModelState.IsValid = false. If all matched properties are successfully assigned, the binding engine sets ModelState.IsValid = true. The ModelState also contains error information for every property that failed.

Changing the Visual Studio Launch Settings

The launchSettings.json file controls many aspects of the debugging experience when using Visual Studio. As mentioned earlier, these settings can be changed through the project properties screen or by updating the file itself. Open the file and update it to match the following:

```
{
  "iisSettings": {
    "windowsAuthentication": false,
    "anonymousAuthentication": true,
    "iisExpress": {
      "applicationUrl": "http://localhost:8477/",
      "sslPort": 0
    }
  },
  "profiles": {
    "IIS Express": {
      "commandName": "IISExpress",
      "launchBrowser": true,
      "launchUrl": "api/category",
      "environmentVariables": {
        "ASPNETCORE_ENVIRONMENT": "Development"
      }
    },
    "SpyStore.Service": {
      "commandName": "Project",
      "launchBrowser": true,

```

```

    "launchUrl": "api/category",
    "environmentVariables": {
      "ASPNETCORE_ENVIRONMENT": "Development"
    },
    "applicationUrl": "http://localhost:40001"
  }
}
}

```

The lines to change are in bold. The `launchUrl` property under the IIS Express profile must be changed to `api/category`. The `applicationUrl` for the Kestrel profile needs to be changed to port 40001, the same port that was set for Kestrel in the application's `Main` method.

Running the Application

Now that the `launchSettings.json` file has been updated, the application can be run by pressing F5 in Visual Studio, just as any other application. If the selector (as shown in Figure 3-4) is set to IIS Express, the application is started with IIS Express, and VS 2017 opens a browser to `http://localhost:8477/api/category`. If the option is set to `SpyStore.Service`, the application is started with Kestrel, and VS 2017 opens a browser to `http://localhost:40001/api/category`.

.NET Core also provides full command line support through the .NET CLI (Command Line Interface). Open a command prompt and navigate to the `SpyStore.Service` project (at the same level of the `SpyStore.Service.csproj` file). Enter the command:

```
dotnet run
```

This launches the application under Kestrel, using the values from the `Main` method of the `Program.cs` file. The output in the command window should resemble this:

```

WebAPI\Begin\SpyStore.Service\src\SpyStore.Service>dotnet run
Hosting environment: Production
Content root path: C:\GitHub\SpyStoreService\src\SpyStore.Service
Now listening on: http://*:40001
Application started. Press Ctrl+C to shut down.

```

Notice that the environment is set to production, so there isn't a browser window launched for the app. This shows that the `launchSettings.json` file is only used when running the application with Visual Studio 2017.

To run it under a different environment, set the value in the command window. This will be scoped to your current command window and does not change your machine settings. Press Ctrl+C to stop the Kestrel service and your application. Set the environment to development (case doesn't matter) and then run the application by entering the following:

```

C:\GitHub\SpyStoreService\src\SpyStore.Service>set aspnetcore_environment=development
C:\GitHub\SpyStoreService\src\SpyStore.Service>dotnet run

```

The data initializer executes since the environment is development and shows in the console window because of the logging configuration. The final lines of the output confirm the environment:

```
Hosting environment: development
Content root path: C:\GitHub\SpyStoreService\src\SpyStore.Service
Now listening on: http://*:40001
Application started. Press Ctrl+C to shut down.
```

Press Ctrl+C to shut down the application. For convenience, create a new file in the same directory as the `SpyStore.Service.csproj` file named `run_development.cmd` and enter the following lines:

```
set aspnetcore_environment=development
dotnet run
```

Now, to run your application in Kestrel as development outside of visual studio, just double-click the command file in File Explorer.

Exception Filters

When an exception occurs in a Web API application, there isn't an error page that gets displayed to the user. In fact, the user is very seldom a person, but another application (such as an MVC application). Instead of just returning an HTTP 500 (server error code), it's helpful to send additional information in the JSON that's returned, so the calling application can react appropriately. While it would be fairly trivial to wrap all of the action methods in try-catch blocks, doing so would lead to a lot of repeated code.

Fortunately, there is a better way. Core MVC allows the creation of filters that can run before or after a particular stage of the pipeline, or in the event of an unhandled exception. Filters can be applied globally, at the controller level, or at the action level. For this application, you are going to build an exception filter to send formatted JSON back (along with the HTTP 500) and include a stack trace if the site is running in debug mode.

■ **Note** Filters are an extremely powerful feature of Core MVC. In this chapter, we are only examining exception filters, but there are many more that can be created that can save significant time when building Core MVC applications. For the full information on filters, refer to the documentation here: <https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/filters>.

Creating the SpyStoreExceptionFilter

Create a new directory in the `SpyStore.Service` project named `Filters` by clicking the project name and selecting `Add ► New Folder`. Add a new class named `SpyStoreExceptionFilter.cs` in this folder by right-clicking the folder and selecting `Add ► Class`.

Add the following using statements to the top of the class:

```
using SpyStore.DAL.Exceptions;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;
using Microsoft.EntityFrameworkCore;
```

Next, change the class to public and implement the `IExceptionHandler` interface, like so:

```
public class SpyStoreExceptionHandler : IExceptionHandler
{
    public void OnException(ExceptionContext context)
    {
        throw new NotImplementedException();
    }
}
```

Unlike most filters in Core MVC that have a before and after event handler, exception filters only have one handler—`OnException` (or `OnExceptionAsync`). If the environment is `Development`, the response should also include the stack trace. Add a class level variable to hold the development indicator and add a constructor as follows:

```
private readonly bool _isDevelopment;

public SpyStoreExceptionHandler(bool isDevelopment)
{
    _isDevelopment = isDevelopment;
}
```

When this filter is instantiated, you will use the `IHostingEnvironment` to determine if this is running in a development environment, and if so, include the stack trace in the error message like this:

```
string stackTrace = (_isDevelopment)?context.Exception.StackTrace:string.Empty;
```

Finally, add the code for the `OnException` event handler. The flow is very straightforward: Determine the type of exception that occurred, build a dynamic object to contain the values to be sent to the calling request, and return an appropriate `ActionResult`. Each of the `BadRequestObjectResult` and `ObjectResult` convert the anonymous objects into JSON as part of the HTTP response. The code to do this is shown here:

```
public void OnException(ExceptionContext context)
{
    var ex = context.Exception;
    string stackTrace = (_isDevelopment)?context.Exception.StackTrace:string.Empty;
    string message = ex.Message;
    string error = string.Empty;
    IActionResult actionResult;
    if (ex is InvalidQuantityException)
    {
        //Returns a 400
        error = "Invalid quantity request.";
        actionResult = new BadRequestObjectResult(
            new { Error = error, Message = message, StackTrace = stackTrace});
    }
    else if (ex is DbUpdateConcurrencyException)
    {
        //Returns a 400
        error = "Concurrency Issue.";
    }
}
```

```

    actionResult = new BadRequestObjectResult(
        new { Error = error, Message = message, StackTrace = stackTrace });
}
else
{
    error = "General Error.";
    actionResult = new ObjectResult(
        new { Error = error, Message = message, StackTrace = stackTrace })
    {
        StatusCode = 500
    };
}
context.Result = actionResult;
}

```

If you want the exception filter to swallow the exception and set the response to a 200, add the following line before setting the Result:

```
context.ExceptionHandled = true;
```

Adding the Exception Filter for All Actions

Filters can be set globally, at the class level, or on individual actions. Exception filters are usually set globally, and that is what you are going to do next. Open `Startup.cs` and add the following using statement to the top of the file:

```
using SpyStore.Service.Filters;
```

Next, navigate to the `ConfigureServices` method. Change the `AddMvcCore` line to the following:

```

services.AddMvcCore(config =>
    config.Filters.Add(new SpyStoreExceptionFilter(_env.IsDevelopment())))
    .AddJsonFormatters(j =>
    {
        j.ContractResolver = new DefaultContractResolver();
        j.Formatting = Formatting.Indented;
    });

```

This sets the `SpyStoreExceptionFilter` for all actions in the application.

Building the Controllers

Now that the groundwork has been laid, it's time to build the controllers that compose the service. The majority of the controllers in the `SpyStore` service provide read-only data through HTTP Get commands. The `ShoppingCartController` contains examples of read-write actions, in addition to providing read-only data about the shopping cart.

The Category Controller

The first controller to program is the `CategoryController`, which you added previously. Delete the `Post`, `Put`, and `Delete` methods, since they are not used in this controller. Add the following `using` statements to the top of the file:

```
using SpyStore.DAL.Repos.Interfaces;
using SpyStore.Models.ViewModels.Base;
```

The `CategoryController` needs instances of the `CategoryRepo` and `ProductRepo` classes. These were added to the DI container in the `ConfigureServices` method in the `Startup` class. All the controller needs is parameters of type `ICategoryRepo` and `IProductRepo`. The Core MVC DI container does all of the work. Add the following properties at the top of the class and the following constructor code:

```
private ICategoryRepo Repo { get; set; }
private IProductRepo ProductRepo { get; set; }

public CategoryController(ICategoryRepo repo, IProductRepo productRepo)
{
    Repo = repo;
    ProductRepo = productRepo;
}
```

While not necessary in the RTM version of Core MVC, the `[FromServices]` attribute can be applied to the parameters to indicate that the parameters get populated by the DI container.

The `Get` methods use the techniques already discussed in this chapter to return the list of `Category` records, or a specific `Category` by `Id`. Update the `Get` methods to the following code:

```
[HttpGet]
public IActionResult Get()
{
    return Ok(Repo.GetAll());
}

[HttpGet("{id}")]
public IActionResult Get(int id)
{
    var item = Repo.Find(id);
    if (item == null)
    {
        return NotFound();
    }
    return Json(item);
}
```

The final method retrieves all products for a specific category. Add the following code to the class:

```
[HttpGet("{categoryId}/products")]
public IEnumerable<ProductAndCategoryBase> GetProductsForCategory(int categoryId)
=> ProductRepo.GetProductsForCategory(categoryId).ToList();
```

The route adds a variable and a literal to the controller route and returns all of the products with category information for a specific `Category Id`.

The Customer Controller

The `CustomerController` provides two API methods, one that returns all customer records, and the other that gets a specific `Customer` by `Id`. Add a new controller named `CustomerController.cs` and delete the `Post`, `Put`, and `Delete` methods. Add the following `using` statements to the top of the file:

```
using SpyStore.Models.Entities;
using SpyStore.DAL.Repos.Interfaces;
```

Leave the `Route` attribute as it is, create a constructor that takes an instance of the `ICustomerRepo` interface, and add a private variable to hold the instance, like this:

```
[Route("api/[controller]")]
public class CustomerController : Controller
{
    private ICustomerRepo Repo { get; set; }
    public CustomerController(ICustomerRepo repo)
    {
        Repo = repo;
    }
    //omitted for brevity
}
```

Update the `Get` methods to the following:

```
[HttpGet]
public IEnumerable<Customer> Get() => Repo.GetAll();

[HttpGet("{id}")]
public IActionResult Get(int id)
{
    var item = Repo.Find(id);
    if (item == null)
    {
        return NotFound();
    }
    return new ObjectResult(item);
}
```

The `Get` methods use a different format than the `Get` methods from the `CategoryController`, demonstrating another way of writing the code to achieve the same results.

The Search Controller

The SearchController has one action method that uses the IProductRepo to search for products. Instead of creating a new controller class, you just add a new class named SearchController.cs. Delete all of the contents of the new class and add the following code:

```
using Microsoft.AspNetCore.Mvc;
using SpyStore.Models.ViewModels.Base;
using SpyStore.DAL.Repos.Interfaces;
using System.Collections.Generic;

namespace SpyStore.Service.Controllers
{
    [Route("api/[controller]")]
    public class SearchController : Controller
    {
        private IProductRepo Repo { get; set; }
        public SearchController(IProductRepo repo)
        {
            Repo = repo;
        }

        [HttpGet("{searchString}", Name = "SearchProducts")]
        public IEnumerable<ProductAndCategoryBase> Search(string searchString)
            => Repo.Search(searchString);
    }
}
```

The Search uses one of the view models created in the data access method, and it is invoked from the route /api/search/{searchString}, like this example search:

<http://localhost:8477/api/search/persuade%20anyone>

The Orders Controller

The OrdersController only contains two methods, one to get the entire order history for a customer, and the other to get an individual order for a Customer. Create a new class named OrdersController.cs, clear out the contents, and add the following code:

```
using Microsoft.AspNetCore.Mvc;
using SpyStore.DAL.Repos.Interfaces;

namespace SpyStore.Service.Controllers
{
    [Route("api/[controller]/{customerId}")]
    public class OrdersController : Controller
    {
        private IOrderRepo Repo { get; set; }
        public OrdersController(IOrderRepo repo)
        {
            Repo = repo;
        }
    }
}
```

```

public IActionResult GetOrderHistory(int customerId)
{
    var orderWithTotals = Repo.GetOrderHistory(customerId);
    return orderWithTotals == null ? (ActionResult)NotFound()
        : new ObjectResult(orderWithTotals);
}

[HttpGet("{orderId}", Name = "GetOrderDetails")]
public IActionResult GetOrderForCustomer(int customerId, int orderId)
{
    var orderWithDetails = Repo.GetOneWithDetails(customerId, orderId);
    return orderWithDetails == null ? (ActionResult)NotFound()
        : new ObjectResult(orderWithDetails);
}
}
}

```

The base route adds the `customerId` variable placeholder to the standard `api/[controller]` route used in the other controllers. The controller takes an instance of the `IOrderRepo`, a pattern that should be familiar by now. The two HTTP Get methods return a `NotFoundResult` if there aren't any records that match the query.

The `GetOrderHistory` method doesn't have a routing attribute because it responds to a HTTP Get without any modifications to the route. The `GetOrderForCustomer` action method returns an `IEnumerable<OrderWithDetailsAndProductInfo>`, another one of the data access library view models. It also includes a named route, which will come into play with the `ShoppingCartController`, which will be covered soon.

The Product Controller

The `ProductController` has the (now) standard Get action methods and another action method to get the featured products with their category information. All three methods use the `ProductAndCategoryBase` view model from the data access layer. Create a new class named `ProductController.cs`, clear out the template code, and add the following code:

```

using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
using SpyStore.Models.ViewModels.Base;
using SpyStore.DAL.Repos.Interfaces;
using System.Linq;

namespace SpyStore.Service.Controllers
{
    [Route("api/[controller]")]
    public class ProductController : Controller
    {
        private IProductRepo Repo { get; set; }
        public ProductController(IProductRepo repo)
        {
            Repo = repo;
        }
        [HttpGet]

```

```

public IEnumerable<ProductAndCategoryBase> Get()
    => Repo.GetAllWithCategoryName().ToList();
[HttpGet("{id}")]
public IActionResult Get(int id)
{
    var item = Repo.GetOneWithCategoryName(id);
    if (item == null)
    {
        return NotFound();
    }
    return new ObjectResult(item);
}
[HttpGet("featured")]
public IEnumerable<ProductAndCategoryBase> GetFeatured()
    => Repo.GetFeaturedWithCategoryName().ToList();
}
}

```

The `GetFeatured` action method adds a literal to the default controller route.

The Shopping Cart Controller

The final controller to add is the workhorse of the sample application. In addition to returning one or all of the shopping cart records for a customer, the rest of the CRUD operations are implemented as well. The final method executes the stored procedure to execute a purchase.

Create a new class in the `Controllers` directory named `ShoppingCartController.cs`. Clear out the existing code and add the following using statements, namespace, class definition, and constructor:

```

using System;
using System.Collections.Generic;
using Microsoft.AspNetCore.Mvc;
using Newtonsoft.Json;
using SpyStore.DAL.Repos.Interfaces;
using SpyStore.Models.Entities;
using SpyStore.Models.ViewModels;

namespace SpyStore.Service.Controllers
{
    [Route("api/[controller]/{customerId}")]
    public class ShoppingCartController : Controller
    {
        private IShoppingCartRepo Repo { get; set; }
        public ShoppingCartController(IShoppingCartRepo repo)
        {
            Repo = repo;
        }
    }
}

```

The preceding code should be familiar by now. The route extends the standard route with a `customerId` variable. An instance of `IShoppingCartRepo` is injected into the constructor and assigned to a class level variable.

The `GetShoppingCartRecord` action method returns a single record from the cart for a `CustomerId` and `ProductId`. The `GetShoppingCart` action method gets all records for a `CustomerId`, and also has a route name assigned to it that will be used later in this controller. Add the following code:

```
[HttpGet("{productId}")]
public CartRecordWithProductInfo GetShoppingCartRecord(int customerId, int productId)
    => Repo.GetShoppingCartRecord(customerId, productId);
[HttpGet(Name = "GetShoppingCart")]
public IEnumerable<CartRecordWithProductInfo> GetShoppingCart(int customerId)
    => Repo.GetShoppingCartRecords(customerId);
```

The next action method to add is to create new records in the shopping cart, which in HTTP parlance is a *post*. Add the following code to the `ShoppingCartController`:

```
[HttpPost]
public IActionResult Create(int customerId, [FromBody] ShoppingCartRecord item)
{
    if (item == null || !ModelState.IsValid)
    {
        return BadRequest();
    }
    item.DateCreated = DateTime.Now;
    item.CustomerId = customerId;
    Repo.Add(item);
    return CreatedAtRoute("GetShoppingCart",
        new { controller = "ShoppingCart", customerId = customerId });
}
```

The customer ID is passed in through the route, and the new cart record is sent in through the body of the request as JSON. The `[FromBody]` attribute instructs Core MVC to use model binding to create an instance of the `ShoppingCartRecord` class from the JSON contained in the request body. Unlike the `FromServices` attribute for objects injected into constructors and methods, the `FromBody` attribute is required if the JSON is in the body of the request. If nothing is passed (`item == null`) or if the JSON contains invalid values (`ModelState.IsValid`), a new `BadRequestResult` is returned.

If a new `ShoppingCartRecord` is successfully created from the JSON in the request message, the `DateCreated` is set to the current date, the `CustomerId` is set to the `customerId` from the route values, and the item is added to the database.

The final line creates the URL for the route named `GetShoppingCart` using the supplied values, adds the URL to the header of the HTTP response, and returns an HTTP 201 (Created). The created header value will resemble this:

Location: <http://localhost:8477/api/ShoppingCart/0>

The `Update` action method responds to HTTP Put requests, and then takes the `ShoppingCartRecord` Id as the final of the route value and the `ShoppingCartRecord` from the JSON contained in the body of the request message. Add the following code to the controller:

```
[HttpPut("{shoppingCartRecordId}")]
public IActionResult Update(int customerId, int shoppingCartRecordId,
    [FromBody] ShoppingCartRecord item)
```

```

{
    if (item == null || item.Id != shoppingCartRecordId || !ModelState.IsValid)
    {
        return BadRequest();
    }
    item.DateCreated = DateTime.Now;
    Repo.Update(item);
    //Location: http://localhost:8477/api/ShoppingCart/0 (201)
    return CreatedAtRoute("GetShoppingCart", new { customerId = customerId });
}

```

After checking to make sure there is a record and the model binding is valid, the `DateCreated` property is set to the current date and then passed into the `Update` method on the repo. Once complete, a HTTP 201 is returned, and the URL of the shopping cart is added to the header of the response.

The final CRUD operation is the `Delete` action method. The method adds the `shoppingCartRecordId` and the string version of the `TimeStamp` property to the route, and only responds to HTTP Delete requests. Add the following code to the controller:

```

[HttpDelete("{shoppingCartRecordId}/{timeStamp}")]
public IActionResult Delete(int customerId, int shoppingCartRecordId, string timeStamp)
{
    if (!timeStamp.StartsWith("\""))
    {
        timeStamp = $"\"{timeStamp}\"";
    }
    var ts = JsonConvert.DeserializeObject<byte[]>(timeStamp);
    Repo.Delete(shoppingCartRecordId, ts);
    return NoContent();
}

```

The first block of code makes sure that the `timeStamp` value contains quotes, as they are needed by the `JsonConvert.DeserializeObject` method. Then the string is converted to a `byte[]` and is passed into the `Delete` method with the ID. If all is successful, a `NoContentResult` is returned, which equates to an HTTP 204.

The final action method is for purchasing the items in the cart. The `Purchase` action method takes the literal `buy` at the end of the controller route and responds to HTTP Post requests. As a precaution, the customer making the purchase is sent into the method as JSON in the request body. Add the following code to the controller:

```

[HttpPost("buy")]
public IActionResult Purchase(int customerId, [FromBody] Customer customer)
{
    if (customer == null || customer.Id != customerId || !ModelState.IsValid)
    {
        return BadRequest();
    }
    int orderId;
    orderId = Repo.Purchase(customerId);
    //Location: http://localhost:8477/api/Orders/0/1
    return CreatedAtRoute("GetOrderDetails",
        routeValues: new { customerId = customerId, orderId = orderId },
        value: orderId);
}
}

```

If all the values check out, the `Purchase` method on the `repo` is called. This method returns the new `OrderId`, which is then passed into the `CreatedAtRoute` method. This version calls a route in a different controller (specifically in the `OrdersController`), passing in the `CustomerId` and the `OrderId`.

Using the Combined Solution

The source code for this chapter includes a folder named `SpyStore.Service.Combined`. It includes the projects from Chapter 2 instead of using the NuGet packages.

The Unit Test Solution

Unit testing with `xUnit` was introduced in Chapter 1. Due to space considerations, testing the service couldn't be covered in the text, but rest assured, there is a full set of unit/integration tests available.

Included in the downloadable source code is another solution named `SpyStore.Service.Tests`, which is a test project for the `SpyStore.Service` project. It is included as a separate project for debugging purposes. If the tests and the service are all in the solution, running the tests and the service gets a little bit tricky. It's much easier to have two separate solutions.

To run the unit tests, make sure the `SpyStore.Service` is running (either through the .NET Core CLI or Visual Studio), and then execute the tests in the test solution the same way they were executed when testing the data access layer. Just make sure the `ServiceAddress` in the `BaseTestClass` is correct.

Open the `BaseTestClass.cs` file (located in `TestClasses\Base` of the `SpyStore.Service.Tests` project), and check that the following line contains the correct URL for the running service:

```
protected string ServiceAddress = "http://localhost:40001/";
```

■ **Source Code** The completed `SpyStore.Service` and `SpyStore.Service.Tests` solutions can be found in the `Chapter 03` subdirectory of the download files.

Summary

This chapter covered the RESTful service that is the backend for the UI frameworks covered in the rest of the book. The chapter began by examining the Model View Controller pattern and ASP.NET Core MVC.

The next section explored the key pieces of Core MVC, creating the projects in Core MVC, adding the NuGet packages, and examining and updating each of the files in the project for the `SpyStore`-specific requirements.

The next section went into routing, and how requests are directed to your controllers and actions. That led into controller and actions, creating an example controller class with actions, and then running under IIS and Kestrel.

Next, you created an `ExceptionHandler` to handle errors in your action methods and wired the filter into all action methods in the application. The final section built all of the controllers for the `SpyStore` service.

CHAPTER 4



Introducing ASP.NET Core MVC Web Applications

Now that the database, data access layer, and the RESTful service are built, it's time to create a user interface. The next two chapters stay within the ASP.NET .NET Core technology stack, using Core MVC to create the SpyStore web application. This chapter builds the foundation for the web application, and then the UI is completed in the next chapter.

■ **Note** The two chapters covering Core MVC Web Applications are laid out in a serial, bottom-up manner, which is probably different than most developers' workflow when building web applications. The site infrastructure is built first, and then the next chapter builds the Views for UI. I typically build applications in a more parallel, top-down manner, working on the UI and building the supporting items as they are needed. The first draft of these chapter were documented in that manner, and it was very hard to write (and read) due to the necessary bouncing between topics.

Introducing the “V” in ASP.NET Core MVC

The previous chapter covered building the RESTful service layer using Core MVC. When building services, only the “M” (Models) and the “C” (Controllers) of Core MVC are used. This chapter (and the next) builds on your knowledge by bringing the “V” (Views) into the development process. The merger between ASP.NET Web API and ASP.NET Core MVC means all of the tools and code that you used to build Core MVC Services apply to Core MVC. The new features in Core MVC explored in Chapter 3—including (but not limited to) the streamlined HTTP pipeline configuration, improved configuration system, and native dependency injection—are also supported in Core MVC Web Applications.

Core MVC supports everything you are accustomed to in prior versions of MVC, including (but not limited to) routing, model binding, validation, view and editor templates, strongly typed views, the Razor View Engine, and areas. Two new features in Core MVC—Tag Helpers and View Components—are introduced in this chapter. A *Tag Helper* is server side code, placed in an HTML tag, that helps shape the rendering of that tag. *View Components* are a combination of child actions and partial views. All of these features will be used to build the *SpyStore.MVC* web application.

Creating the Solution and the Core MVC Project

Create a new solution by clicking on File ► New Project and then selecting the ASP.NET Core Web Application (.NET Core) template available under Installed ► Templates ► Visual C# ► Web. Name the project and solution `SpyStore.MVC`, as shown in Figure 4-1.

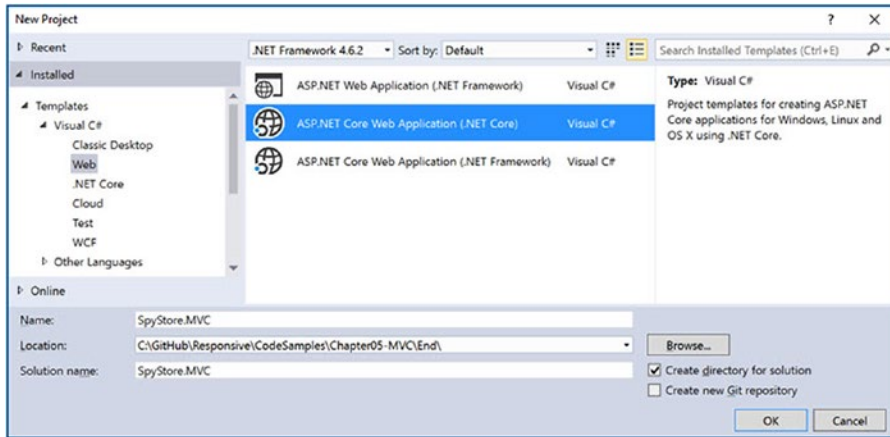


Figure 4-1. Adding a new ASP.NET web application

■ **Note** If you have not already installed Visual Studio 2017 and .NET Core, follow the steps outlined in Chapter 1 to do so.

On the New ASP.NET Core Web Application (.NET Core) screen, select the Web Application template under the ASP.NET Core Templates 1.1 section, leave the Enable Docker Support check box unchecked, and leave the Authentication set to No Authentication, as shown in Figure 4-2.

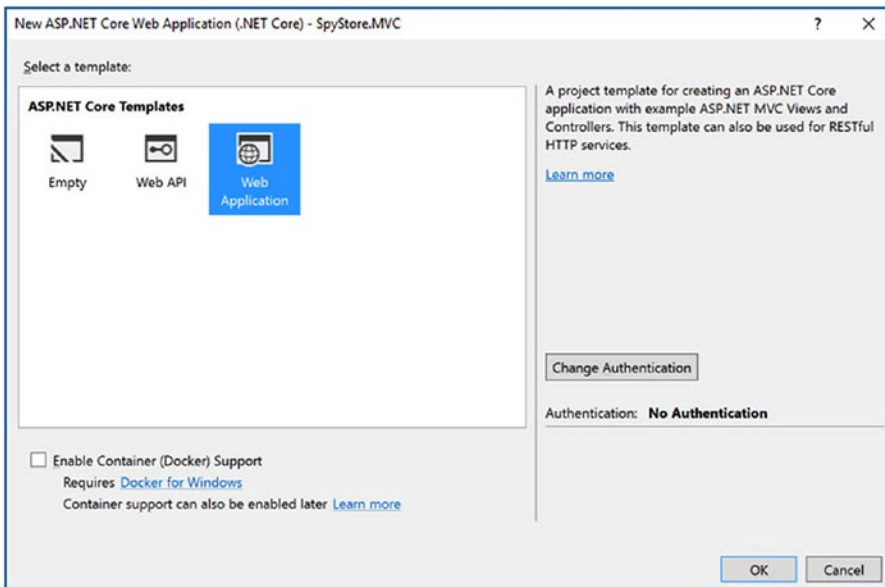


Figure 4-2. *Selecting the Web Application project template*

Click OK, and the new solution and project are created. Your solution and projects should look like Figure 4-3.

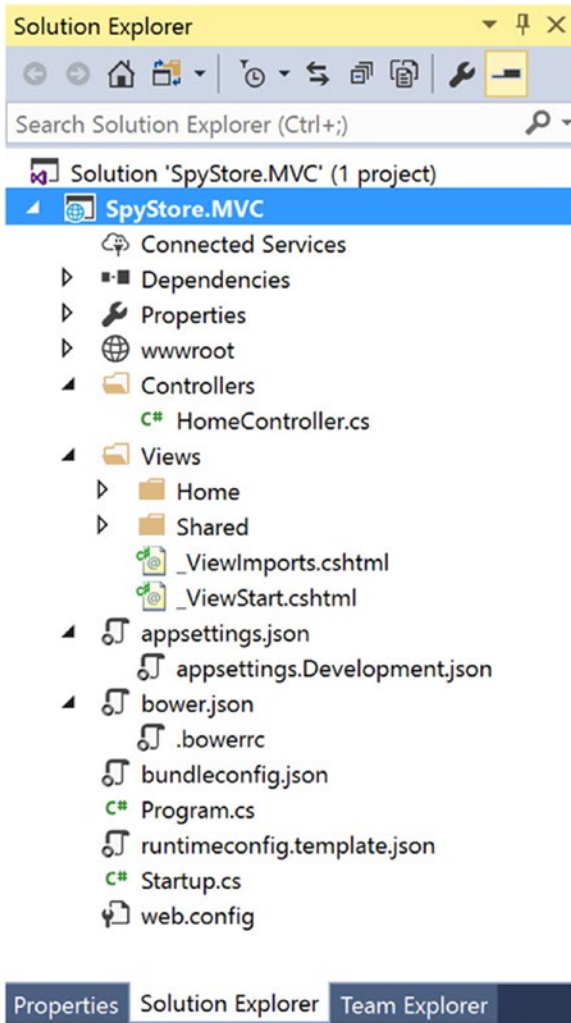


Figure 4-3. The generated files and folders for a Core MVC Web API app

The project structure should look familiar if you've used previous versions of ASP.NET MVC (and from the previous chapter). For Core MVC Service applications, the `wwwroot` directory is empty and there isn't a Views folder. Both of these are covered soon.

Updating and Adding NuGet Packages

Right-click on the `SpyStore.MVC` project and select `Manage NuGet Packages`. Make sure the `Include Prerelease` check box is checked. Click on `Updates` at the top of the `NuGet Package Manager` screen, check the `Select All Packages` check box, and click on the `Update` button. Depending on your NuGet settings, you might get prompted to confirm the installs and accept license agreements. Just click `OK` on the dialogs to install the updated packages.

Next, click on `Browse`, enter `SpyStore` into the search box, and change the `Package Source` to `All` (which includes the local package source just created). Install the `SpyStore.Models` package.

Finally, install the AutoMapper package by clicking on Browse at the top of the screen and entering AutoMapper in the Search box. When the package is located, select it in the left half of the dialog and click on the Install button on the right side. AutoMapper is a utility built by Jimmy Bogard that is designed to flatten complex models for better JSON serialization (among other uses). It is used later in the application.

■ **Note** The version of ASP.NET Core and Visual Studio 2017 used to write this book is having issues with BrowserLink. Since BrowserLink isn't discussed in this book, you can remove the BrowserLink package from the project (either by deleting the line from the `SpyStre.MVC.csproj` file or by using NuGet Package Manager) and comment out the `UserBrowserLink` line in the `Startup.cs` file (covered later in this chapter).

Routing Revisited

Routing with attribute routing was covered extensively in Chapter 3. Traditional routing (using the `MapRoute` method to add routes to the route table), used in earlier versions of MVC, is available in Core MVC in addition to attribute routing.

■ **Note** All Core MVC projects support both attribute routing and the “traditional” method of defining routes in the route table, as well as a combination of the two. Core MVC Web API projects tend to use attribute routing, and Core MVC Web Applications tend to use a combination approach, but that isn't a rule. Decide the approach that works best for you.

The Route Table

MVC Web Applications use a centralized route table that holds the routing information for the application and is handled by the `RouterMiddleware`. The route table values are used by the routing middleware to decompose incoming URL paths into the correct Controller and Action method as well as any parameters that get passed into the Action. Outgoing routes can be created from the route table, eliminating hard coding actual URL strings in the application. The `MvcRouteHandler` (created by the routing middleware) will pick the first matching route in the route table, even if it might not be the most “correct” route, so care must be used when defining the routes.

URL Templates and Default Values

The same tokens used in attribute routing are valid, although they are delineated with braces (`{}`) instead of brackets (`[]`). Tokens can also have default values assigned to them when added through the `MapRoute` command. For example, the following route is the standard “default” route in MVC:

```
{controller=Home}/{action=Index}/{id?}
```

This is the same format of many of the routes used in Chapter 3 with the addition of the default values, which are invoked from left to right. For example, if a request is made to the site without a controller specified, this route will map to `\Home\Index`. If a controller (e.g., `ProductsController`) is specified, but an action is not, the request is routed to `\Products\Index`. As with attribute routing, the Controller part of the name is dropped when building routes.

MVC Web Applications Projects and Files

This section covers the files and folders that are not used in Core MVC Services or have changed from how they are used when creating Core MVC Services.

The Program.cs File

The template for web applications creates the same `program.cs` file and `Main` method as the Web API template does. The only change that is needed is to update the Kestrel configuration for this application to use port 40002, as follows:

```
var host = new WebHostBuilder()
    .UseKestrel()
    .UseUrls("http://*:40002/")
    .UseContentRoot(Directory.GetCurrentDirectory())
    .UseIISIntegration()
    .UseStartup<Startup>()
    .Build();
host.Run();
```

Specifying the port for Kestrel is optional, since the web application is called from another application, but viewed through a browser. I find it a good practice to change the default port to prevent different Core MVC applications from colliding on your development machine.

The appsettings.json File

The root URL for the `SpyStore.Service MVC Web API` application needs to be added to the `appsettings.json` file. Add the base URL and port for your service into the `appsettings.json` file by updating the text to the following:

```
{
  "Logging": {
    "IncludeScopes": false,
    "LogLevel": {
      "Default": "Warning"
    }
  },
  "WebServiceLocator": {
    "ServiceAddress": "http://localhost:40001/"
  }
}
```

No changes need to be made to the `appsettings.development.json` file.

The Startup.cs File

Most of the `Startup.cs` file is the same between MVC Web APIs and MVC Web Applications. The differences are covered in the next sections.

The ConfigureServices Method

To the ConfigureServices method, add the following line:

```
services.AddSingleton(_ => Configuration);
```

This adds the Configuration property to the DI container for use in the rest of the application.

The Configure Method

The MVC Web Application template creates the following code in the Configure method:

```
public void Configure(
    IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory loggerFactory)
{
    loggerFactory.AddConsole(Configuration.GetSection("Logging"));
    loggerFactory.AddDebug();

    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
        app.UseBrowserLink();
    }
    else
    {
        app.UseExceptionHandler("/Home/Error");
    }

    app.UseStaticFiles();

    app.UseMvc(routes =>
    {
        routes.MapRoute(
            name: "default",
            template: "{controller=Home}/{action=Index}/{id?}");
    });
}
```

The first two lines set up logging for the app. The if block checks the environment, and if it's development, adds two middleware packages. The first (UseDeveloperExceptionPage) is a diagnostic package that will display an enhanced error page geared for debugging. The second adds BrowserLink support. Comment out this line if you removed the BrowserLink package from your project. If the environment is not development, the error route "/Home/Error" will be launched. The SpyStore.MVC app uses the ProductsController (coming later) as the base controller, not the HomeController created with the template. Update the UseExceptionHandler method to use the ProductsController, as follows:

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

The next line adds support for static files located in the default wwwroot directory. If this middleware isn't added, the site will not serve up any style sheets or other static content. When UseStaticFiles is called without a parameter, the directory from the call to UseContentRoot in the Main method is used. This directory can also be changed by passing a new location into the call to UseStaticFiles.

The final line enables full MVC support. The lambda in the constructor adds a route to the route table. The route supplied is the default route, and it is so commonly used that there is a convenience method that replaces the `UseMvc` method and adds the default route:

```
app.UseMvcWithDefaultRoute();
```

The only change needed for this app is to change the default route to use the `ProductsController` instead of the `HomeController`, as follows:

```
app.UseMvc(routes =>
{
    routes.MapRoute(
        name: "default",
        template: "{controller=Products}/{action=Index}/{id?}");
});
```

The Controllers Folder

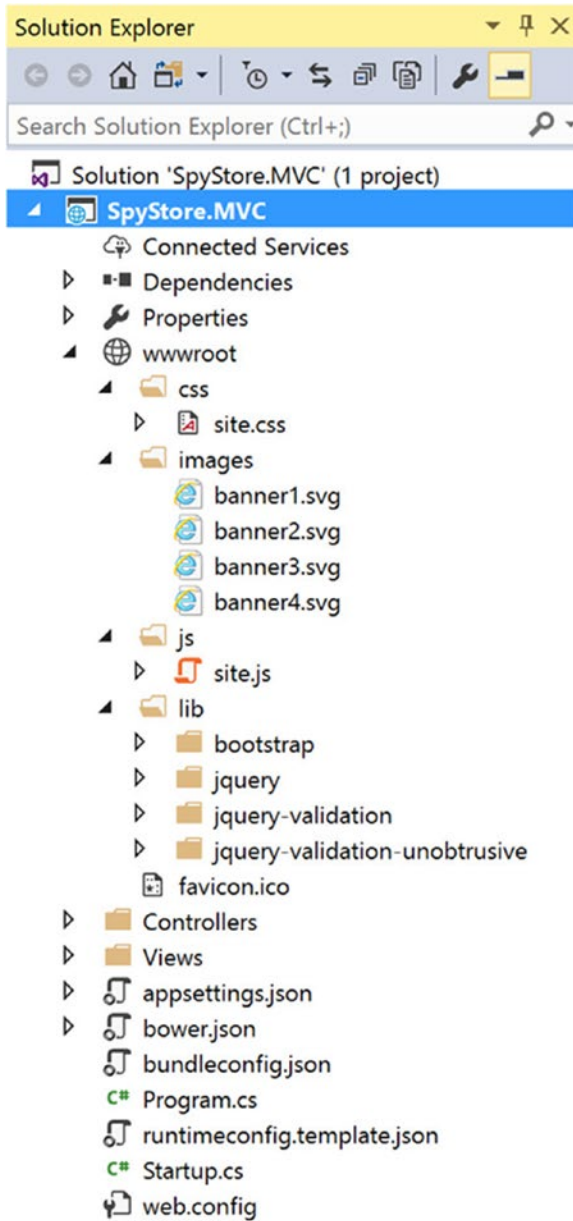
Controllers in Web API and Web Applications are responsible for the same work load. MVC conventions dictate that all controllers are located in this directory. Controllers and Actions are covered later in the chapter. For now, delete the `HomeController.cs` file.

The Views Folder

The Views are the user interface component of MVC applications. MVC conventions dictate that all views are located in this directory. Views are covered in-depth later in the chapter. For now, delete the `Home` folder under `Views`.

The wwwroot Folder

This folder contains all of the client side assets (such as style sheets, JavaScript files, and images) that are served up by the `UseStaticFiles` middleware. The Core MVC Web Application template preloads key client side files (images, style sheets, JavaScript files, etc.), as shown in [Figure 4-4](#).



Properties Solution Explorer Team Explorer

Figure 4-4. The generated files and folders under `wwwroot` for a Core MVC Web API app

The next sections cover the directories and files contained in the `wwwroot` directory.

The Site CSS, JavaScript, and Images

There are four initial folders under the `wwwroot` directory. Three of them are designed to contain application specific code and content: one for style sheets files, one for JavaScript files, and one for image files. This is the template's default organizational structure and can easily be changed if application needs dictate a change from the default structure. Moving content outside of `wwwroot` is possible as well, as long as the `UseStaticFiles` is updated to specify the new location. This breaks convention and is not covered in this book.

Adding the Site CSS and Image Files

This application's style sheets and images are contained in the downloadable code for this chapter under the `WebArtifacts` directory. Delete the CSS and images files from the `SpyStore.MVC` project and add the following files from the downloads to the `css` directory:

- `site.css`
- `spystore-bootstrap.css`
- `spystore-bootstrap.min.css`
- `spystore-bootstrap.min.css.map`

Next, add a folder named `fonts` under the `css` directory. Add the following files from the downloads to the `fonts` folder:

- `glyphicons-halflings-regular.eot`
- `glyphicons-halflings-regular.svg`
- `glyphicons-halflings-regular.ttf`
- `glyphicons-halflings-regular.woff`
- `glyphicons-halflings-regular.woff2`

Finally, add the following files from the downloads to the `images` folder:

- `bg.jpg`
- `jumbo.png`
- `product-image.png`
- `product-image-lg.png`
- `product-thumb.png`
- `store-logo.png`

Adding the JavaScript Files

Create a new directory named `validations` under the `js` directory. In this directory, create two empty JavaScript files named `validators.js` and `errorFormatting.js` by right-clicking on the `validations` directory and selecting `Add ► New Item`. Select `ASP.NET Core` in the left side of the dialog and `JavaScript File` in the right side. These files are used later in the chapter.

The lib Folder

This folder is the holding location for Bower packages (covered soon) installed into the application. Typically, there is never a need to change this folder or the contents, as they are managed by Bower.

The favicon.ico File

This is the standard image file that all web sites should have. You can add a custom image here or just leave the one provided with the template.

Controllers, Actions, and Views

Controller and actions not only work the same way in Core MVC Web Applications as in Core MVC Services, they are now exactly the same—no more deciding between Controller and ApiController. The biggest difference is in the behavior of the action methods. In Core MVC Services, action methods typically accept and return JSON formatted values.

In web applications, where users (instead of services) are interacting with the site, the action methods are designed to accept data gathered from the user, use that data to interact with the model(s), and then return a `ViewResult` back to the user.

ViewResults

A `ViewResult` is a type of `ActionResult` that encapsulates a Razor View. A `PartialViewResult` is a type of `ViewResult` that is designed to be rendered inside another view. Views and partial views are covered in detail shortly.

Controller Helper Methods for Returning ViewResults

Just as the base Controller class has helper methods for returning HTTP result codes and formatted object results, it also contains helper methods for creating `ViewResult` and `PartialViewResult` instances (named `View` and `PartialView`, respectively) from `*.cshtml` files. If the method is executed without any parameters, the view of the same name as the action method will be rendered without any model data. For example, the following code will render the `MyAction.cshtml` view:

```
public ActionResult MyAction()
{
    return View();
}
```

To change the specific `*.cshtml` file that is rendered, pass in the name of the file (without a extension). The following code will render the `CustomViewName.cshtml` view.

```
public ActionResult MyAction()
{
    return View("CustomViewName");
}
```

The final two overloads provide for passing in a data object that becomes the model for the view. The following examples show this. The first example uses the default view name and the second example specifies a different view name.

```
public ActionResult MyAction()
{
    var myModel = new MyActionViewModel();
    return View(myModel);
}
public ActionResult MyAction()
{
    var myModel = new MyActionViewModel();
    return View("CustomViewName",myModel);
}
```

Views

Views encapsulate the presentation layer in Core MVC Web Applications. They are written using a combination of HTML markup, CSS, JavaScript, and Razor syntax. If you are new to ASP.NET MVC, Razor is used to embed server side code into the client side markup of a view.

The Views Folder

The Views folder is where views are stored in an MVC project, as the name suggests. In the root of the Views folder there are two files, `_ViewStart.cshtml` and `_ViewImports.cshtml`. The `_ViewStart.cshtml` file specifies the default layout to use if one is not specifically assigned for a view. This is discussed in greater detail in the Layouts section. The `_ViewImports.cshtml` file defines the default using statements to be applied to all for all views. These files are executed before a view located in any lower level directory is executed.

■ **Note** Why the leading underscore for `_ViewStart.html`, `_ViewImports.cshtml`, and `_Layout.cshtml`? The Razor View Engine was originally created for WebMatrix, which would allow any file that did *not* start with an underscore to be rendered, so core files (like layout and configuration) all have names that began with an underscore. This is not a convention that MVC cares about, since MVC doesn't have the same issue as WebMatrix, but the underscore legacy lives on anyway.

Each controller gets its own directory under the Views folder where its specific views are stored. The names match the name of the controller (minus the word "Controller"). For example, the `Views\Account` directory holds all of the views for the `AccountController`.

The views are named after the action methods by default, although the names can be changed, as shown earlier. For example, the `Index` method of the `AccountController` would load `Views\Account\Index.cshtml` when a `ViewResult` is requested.

The Shared Folder

There is special directory under Views named `Shared`. This directory holds views that are available to all controllers and actions. If a requested `*.cshtml` file can't be found in the controller-specific directory, the shared folder is searched.

The DisplayTemplates Folder

The `DisplayTemplates` folder holds custom templates that are rendered using the `@Html.Display`, `@Html.DisplayFor`, or `@Html.DisplayForModel` helpers. A template name can be specified in each of those calls, otherwise Razor will search for a file named the same as the type being rendered (e.g., `Customer.cshtml`). The search path starts in the `Views\{CurrentControllerName}\DisplayTemplates` folder, and if it's not found, then looks in the `Views\Shared\DisplayTemplates` folder.

To demonstrate this, create a new folder named `DisplayTemplates` under the `Views\Shared` folder. Add a new view named `Boolean.cshtml` by selecting `Add > New Item` and selecting `MVC View Page` from the `Add New Item` dialog, as shown in [Figure 4-5](#).

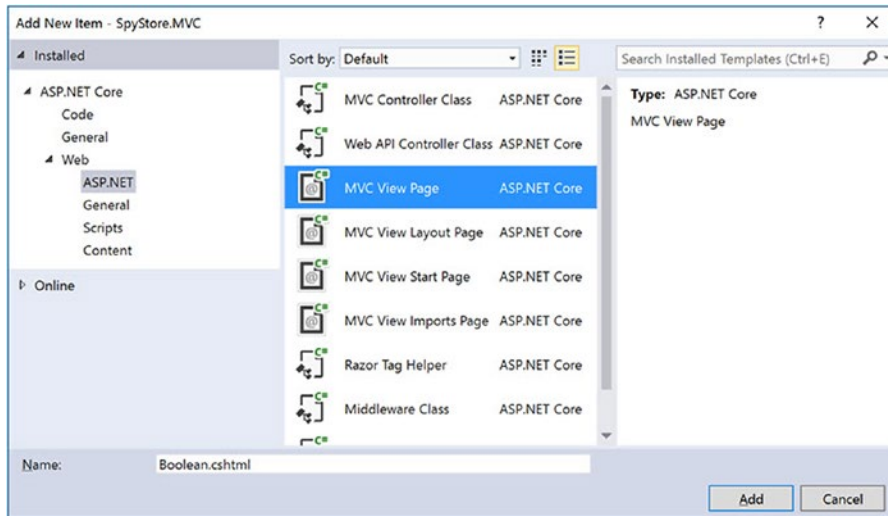


Figure 4-5. Adding a new MVC Razor view page

Clear out any existing content and add the following Razor code:

```
@model bool?
@{
    if (!Model.HasValue)
    {
        @:Unknown
    }
    else if (Model.Value)
    {
        @:Yes
    }
    else
    {
        @:No
    }
}
```

The template is very simple and works with standard bool properties as well as nullable bool properties. If the property is nullable and does not have a value, the template displays the word “Unknown”. Otherwise, it displays “Yes” or “No,” as appropriate. The display template will be automatically used for Boolean (and nullable Boolean) properties.

The EditorTemplates Folder

The EditorTemplates folder works the same as the DisplayTemplates folder, except the templates are used by the @Html.Editor and @Html.EditorFor Razor Helpers and Input Tag Helpers (covered in the next chapter).

Create a new folder named EditorTemplates in the Views\Shared folder. Add a new MVC View Page named Boolean.cshtml to that folder. Clear out the templated contents and add the following code:

```
@model Boolean?
@{
    var value = false;
    var selectionMade = false;
    var list = new List<SelectListItem>();
    if (ViewData.ModelMetadata.IsNullableValueType)
    {
        list.Add(new SelectListItem
        {
            Text = "",
            Value = "",
        });
        if (!Model.HasValue)
        {
            list[0].Selected = true;
            selectionMade = true;
        }
        else
        {
            value = Model.Value;
        }
    }
    else
    {
        value = (bool)Model;
    }
    list.Add(new SelectListItem
    {
        Text = "Yes",
        Value = "true",
        Selected = (!selectionMade && value == true)
    });
    list.Add(new SelectListItem
    {
        Text = "No",
        Value = "false",
        Selected = (!selectionMade && value == false)
    });
}
@Html.DropDownList(string.Empty, list, ViewData["htmlAttributes"])
```

The template creates a drop-down for choosing “Yes” or “No” instead of the standard check box control. If the type is a nullable bool, there is an additional select option for undecided.

The Razor View Engine

The Razor View Engine was designed as an improvement over the Web Forms View Engine and uses Razor as the core language. Razor is server side code that is embedded into a view, is based on C#, and has corrected many annoyances with the Web Forms View Engine. The incorporation of Razor into HTML and CSS results in code that is much cleaner and easier to read, and therefore easier to support and maintain.

Razor Syntax

The first difference between the Web Forms View Engine and the Razor View Engine is that you add Razor code with the @ symbol. There is also intelligence built into Razor that removes a significant number of the Web Forms “nuggets” (<% %>). For example, Razor is smart enough to handle the @ sign in e-mail addresses correctly.

Statement blocks open with an @ and are enclosed in braces, like this (notice how @ is not used as a statement terminator):

```
@foreach (var item in Model)
{
}
```

Code blocks can intermix markup and code. Lines that begin with markup are interpreted as HTML, while lines that begin with code are interpreted as code, like this:

```
@foreach (var item in Model)
{
    int x = 0;
    <tr></tr>
}
```

Lines can also intermix markup and code, like this:

```
<h1>Hello, @username</h1>
```

The @ sign in front of a variable is equivalent to `Response.Write` and, by default, HTML encodes all values. If you want to output un-encoded data (i.e. potentially unsafe data), you have to use the `@Html.Raw(username)` syntax.

The Final Word on Razor

There just isn’t enough space in this book to detail everything that you can do with Razor. You will see more examples of Razor as you work through the rest of this chapter.

Layouts

Similar to Web Forms Master Pages, MVC supports layouts. MVC views can be (and typically are) based on a single base layout to give the site a universal look and feel. Navigate to the `Views\Shared` folder and open the `_Layout.cshtml` file. It looks like a full-fledged HTML file, complete with `<head>` and `<body>` tags.

This file is the foundation that other views are rendered into. This creates a consistent look and feel to the site. Additionally, since most of the page (such as navigation and any header and/or footer markup) is handled by the layout page, Views Pages are kept small and simple. Scroll down in the file until you see the following line of Razor code:

```
@RenderBody()
```

That line instructs the Layout page to render the content of the view. Now scroll down to the line just before the closing `</body>` tag. The following line creates a new section for the layout and makes it optional.

```
@RenderSection("scripts", required: false)
```

Sections can also be marked as required by passing in `true` as the second parameter, like this:

```
@RenderSection("Header", true)
```

The `@section` Razor block lets the layout page know that the content in the block should be rendered in a particular section. For example, to load the `jquery.validate.js` file in a view, add the following markup:

```
@section Scripts {
    @{
        <script src="~/lib/jquery-validation/dist/jquery.validate.js"></script>
    }
}
```

Specifying the Default Layout for Views

The default layout page is defined in the `_ViewStart.cshtml` file. Open this file and examine the contents, shown here:

```
@{
    Layout = "~/Views/Shared/_Layout.cshtml";
}
```

This file just has one Razor code block that sets the layout to a specific file. If a view does not specify its own layout file, the file specified here is used.

Partial Views

Partial views are conceptually similar to a user control in Web Forms. Partial views are useful for encapsulating UI, which reduces (or eliminates) repeating code. A partial view (injected into another view with the `@Html.Partial` helper or returned from a controller with the `PartialView` method) does not use a layout, unless one is specified.

Sending Data to Views

MVC views are strongly typed, based on a model (or `ViewModel`). The model data is passed into the view in the action method by passing the object into the base controller's `View` method.

Strongly Type Views and View Models

When a model or `ViewModel` is passed into the view method, the value gets assigned to the `@model` property of a strongly typed view, as shown here:

```
@model IEnumerable<Order>
```

The `@model` sets the type for the view and can then be accessed by using the `@Model` Razor command, like this:

```
@foreach (var item in Model)
{
    //Do something interesting here
}
```

ViewBag, ViewData, and TempData

The `ViewBag`, `ViewData`, and `TempData` objects are mechanisms for sending small amounts of data into a View. Table 4-1 lists the three mechanisms to pass data from a controller to a view (besides the `Model` property), or from a view to a view.

Table 4-1. Ways to Send Data to a View

Data Transport Object	Description of Use
<code>TempData</code>	This is a short-lived object that works during the current request and next request only. Typically used when redirecting to another view.
<code>ViewData</code>	A dictionary that allows storing values in name/value pairs (e.g., <code>ViewData["Title"] = "Foo"</code>).
<code>ViewBag</code>	Dynamic wrapper for the <code>ViewData</code> dictionary (e.g., <code>ViewBag.Title = "Foo"</code>).

Package Management with Bower

Bower (<https://bower.io/>) is a powerful client side library manager for web sites, containing thousands of packages. Its library is much more extensive than NuGet, and it's managed by the web community at large. Bower is not a replacement for NuGet, but should be considered a supplement. Visual Studio 2017 and .NET Core use Bower to manage client side packages.

The `.bowerrc` file is the configuration file for Bower. It is used to set the relative path for the root location where packages are to be installed into the application, and it defaults to the `lib` directory under `wwwroot`, as follows:

```
{
  "directory": "wwwroot/lib"
}
```

Bower packages included in a Core MVC application are managed through the `bower.json` file. You can edit this file directly to add/remove packages, or use VS2017's Bower Package Manager GUI. The Bower Package Manager works just like NuGet Package Manager and is accessed by right-clicking on the project name in Solution Explorer and selecting Manage Bower Packages.

The `bower.json` files lists all of the packages under the `dependencies` node. The VS2017 template installs four packages, as shown here:

```
{
  "name": "asp.net",
  "private": true,
  "dependencies": {
    "bootstrap": "3.3.7",
    "jquery": "2.2.0",
    "jquery-validation": "1.14.0",
    "jquery-validation-unobtrusive": "3.2.6"
  }
}
```

Updating and Adding Bower Packages

Open the Bower Package Manager GUI by right-clicking on the `bower.json` (or the project name) in Solution Explorer and clicking Update Available near the top. You will see a screen similar to the one shown in Figure 4-6.

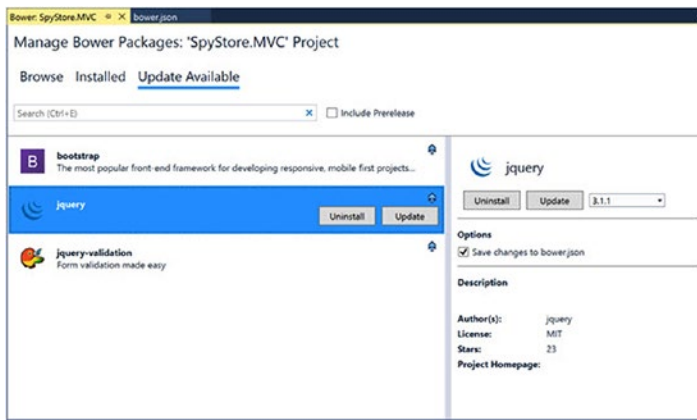


Figure 4-6. Updating the Bower packages

Select `jquery` in the left section and then version 2.2.4 in the right side of the GUI. Click Update. Select `jquery.validation` and click Update in the right side of the GUI. There isn't a need to update Bootstrap because of the custom version that Ben (one of my co-authors) created for the web sites. jQuery can't be updated beyond 2.2.4 because of the custom Bootstrap implementation and the version of Bootstrap used.

Next, click Browse near the top and enter `jquery-unobtrusive-ajax` in the search box. Select the package on the left side of the GUI and install the latest version (3.2.4 at the time of this writing). This package enables `data-ajax` attributes, which is used in the next chapter.

Bower Execution

Visual Studio 2017 watches for the `bower.json` file to change, and when a change is detected, the packages are added or removed (based on the listed packages) into individual subdirectories under `wwwroot\lib`. You can force Visual Studio to update the packages by right-clicking the `bower.json` file and selecting Restore

Packages. The process can also be executed manually from the Package Manager Console. Open Package Manager Console, navigate to the directory where the `bower.json` file is located (e.g., `src/SpyStore.MVC`), and enter the following command:

```
bower install
```

Bundling and Minification

Bundling and minification are important tools for increasing your application's performance. Splitting custom JavaScript and CSS files into smaller, focused files increases maintainability, but doing this can cause performance issues. Browsers have built-in limits controlling how many files can be simultaneously downloaded from the same URL, so too many files (even if they are very small) can lead to slower page loads. Bundling is the process of combining smaller files into larger files. This gets around the browser file download limitation, but must be done wisely, since gigantic files can cause issues as well.

Minification is the process of reducing a file's size by removing unnecessary whitespace, and for JavaScript files, essentially obfuscating the contents. Variable and function names are renamed to values that are much shorter, among size other optimizations. It's not a secure obfuscation scheme, but it does make the file extremely awkward to read.

The BundlerMinifier Project

Bundling and minification for .NET Core projects seemed to be in constant flux during the early versions. The team has settled on the `BundlerMinifier` project created by Mads Kristensen. It runs as both a Visual Studio extension and as a NuGet package for the .NET Core CLI. More information can be found here: <https://github.com/madskristensen/BundlerMinifier>.

■ **Note** Gulp, Grunt, WebPack, and many other build tools are still supported by .NET Core and VS2017. Microsoft backed off of picking a specific framework as the default due to the longevity of support for templates (three years for LTS versions) and the rapid mood swings of the JavaScript community. Chapter 6 covers several client side build tools, including Gulp, Grunt, and WebPack.

Configuring Bundling and Minification

The `bundleconfig.json` file configures the targets and the output files. The default configuration is listed here:

```
// Configure bundling and minification for the project.
// More info at https://go.microsoft.com/fwlink/?LinkId=808241
[
  {
    "outputFileName": "wwwroot/css/site.min.css",
    // An array of relative input file paths. Globbing patterns supported
    "inputFiles": [
      "wwwroot/css/site.css"
    ]
  }
],
```

```

{
  "outputFileName": "wwwroot/js/site.min.js",
  "inputFiles": [
    "wwwroot/js/site.js"
  ],
  // Optionally specify minification options
  "minify": {
    "enabled": true,
    "renameLocals": true
  },
  // Optionally generate .map file
  "sourceMap": false
}
]

```

Each block of JSON has (at a minimum) definitions for input files (globbing is available) and the name and location of the output bundled and/or minified file. For example, the first block in this file listing takes the `site.css` file and creates the minified version named `site.min.css`. If there was more than one file in the `inputFiles` array (or if wildcards are used), then all of the files listed are bundled into the output file. The second block does the same thing for the `site.js` file and shows the optional `minify` and `sourceMap` parameters. Both of these parameters are optional. They are shown set to their respective default values for illustration purposes. The `sourceMap` option is for creating source map files from the input JavaScript files.

Adding Files for Bundling and Minification

The Bootstrap files supplied with the downloadable code are already minified, so those don't need to be added to the `bundleconfig.json` file. The two JavaScript files created earlier (but still empty) do. Add the following to the `bundleconfig.json` file:

```

{
  "outputFileName": "wwwroot/js/validations/validations.min.js",
  "inputFiles": [
    "wwwroot/js/validations/*.js"
  ],
  "minify": {
    "enabled": true,
    "renameLocals": true
  },
  "sourceMap": false
}

```

This block configures `BundlerMinifier` to minify any `.js` files found in the `wwwroot/js/validations` directory and then bundle them into `validations.min.js`. This file won't be created yet, because empty (or missing) files are ignored. When the source files contain content, they will be bundled and minified into the output file.

Visual Studio Integration

Check for the version of the BundlerMinifier extension installed in Visual Studio by selecting Tools ► Extensions and Updates, then selecting Installed in the left side of the dialog. If the extension is not installed, select Online ► Visual Studio Gallery. It should show up in the first page of the Most Popular extensions, but if not, search for it by entering Bundler in the search box, as shown in Figure 4-7. If there is an updated version from what is installed on your machine, it will appear in the Updates section. Install or update the extension to the latest version, which at the time of this writing is 2.2.307. This might require a restart of Visual Studio.

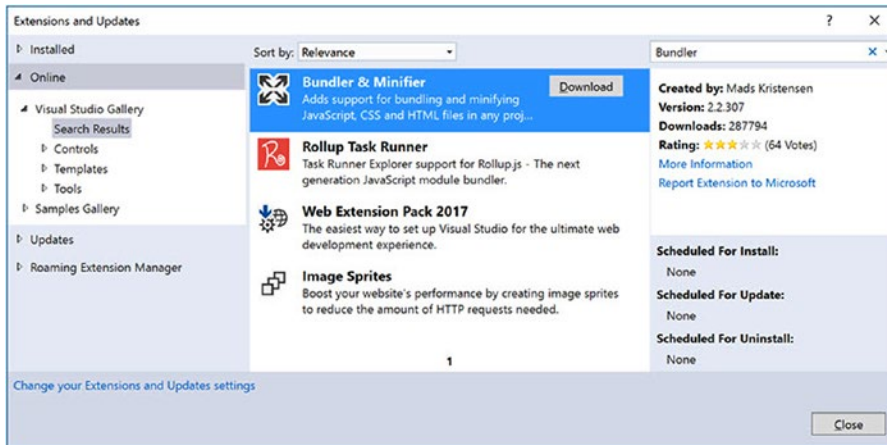


Figure 4-7. Adding the BundlerMinifier extension

Bundling on Change

To configure bundling and minification to watch for changes, right-click on the project name in Solution Explorer and select Bundler & Minifier. Make sure that Produce Output Files is checked, as shown in Figure 4-8.

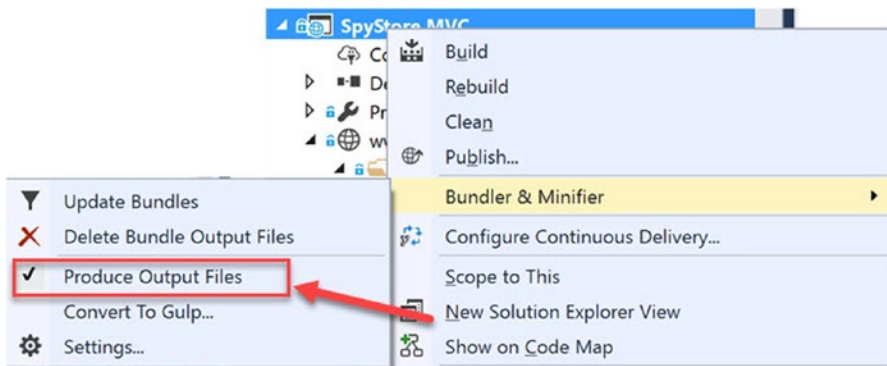


Figure 4-8. Adding the BundlerMinifier extension

Checking this option adds a watcher to the files referenced in `bundleconfig.json`. To test this, navigate to the `css` directory under `wwwroot` and delete the `site.min.css` file. Open the `site.css` file, make an inconsequential change to the file (such as adding a space, or typing a character then backspacing over it), and save it. The `site.min.css` file is recreated.

Bundling on Build

There are two ways to enable bundling when VS2017 builds your project. The first is to right-click on `bundleconfig.json` file, select `Bundler & Minifier`, then check the `Enable Bundle on Build...` option, as shown in Figure 4-9.

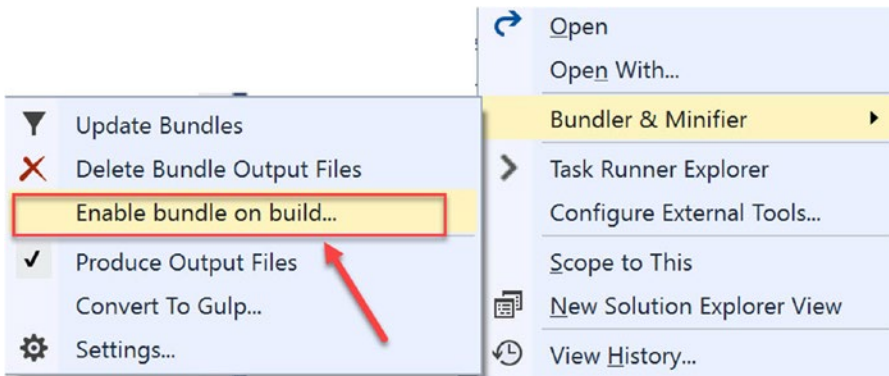


Figure 4-9. Enabling bundling on build

This downloads another NuGet package that hooks the `BundlerMinifier` to the `MSBuild` process. You will have to confirm the install, as shown in Figure 4-10.

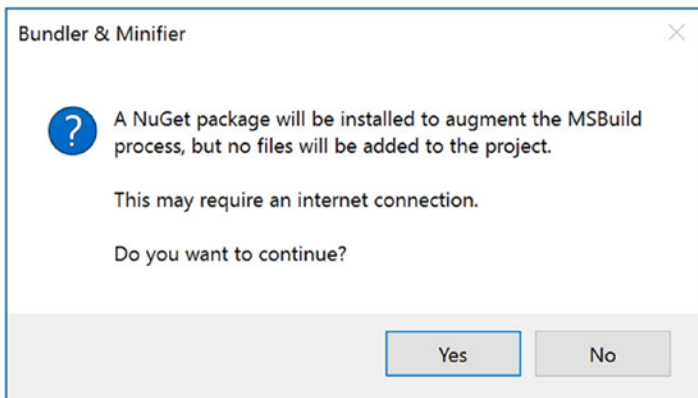


Figure 4-10. Adding support for MSBuild integration

Using the Task Runner Explorer

The Task Runner Explorer can also be used to bind the bundling tasks into the build process. The Task Runner Explorer is another Visual Studio extension created by Mads Kristensen, but is now part of Visual Studio. It enables binding command line calls into the MSBuild process. Open it by selecting Tools ► Task Runner Explorer. The left side of the explorer shows the available commands, and the right side shows the assigned bindings (currently there aren't any), as shown in Figure 4-11.

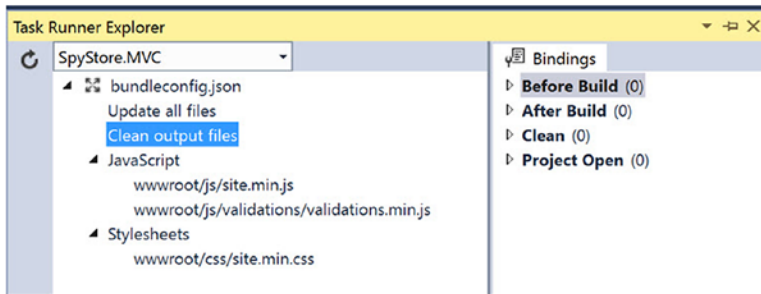


Figure 4-11. Enabling bundling on build

To assign the binding, right-click on the Update All Files command and select Bindings ► After Build, as shown in Figure 4-12.

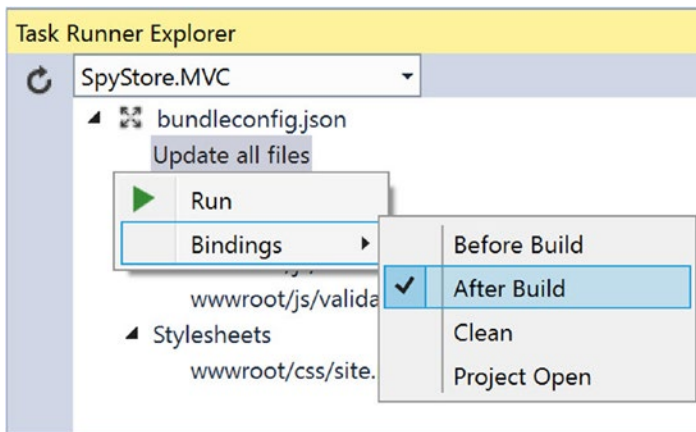


Figure 4-12. Updating all files after each build

For clean output files, assign the Before Build and Clean bindings. This will remove the minified and bundled files before a build starts or if a clean command is executed.

.NET Core CLI Integration

In addition to integrating with Visual Studio, the BundlerMinifier also works with the .NET Core CLI by using the BundlerMinifier.Core NuGet package. Confirm the version available by opening the NuGet Package Manager and searching for BundlerMinifier.Core. At the time of this writing, the current version is

2.2.301, slightly behind the VS extension. Open the `SpyStoreMVC.csproj` file by right-clicking on the project name in Solution Explorer and then selecting `Edit SpyStoreMVC.csproj`. Add the following markup:

```
<ItemGroup>
  <DotNetCliToolReference Include="BundlerMinifier.Core" Version="2.2.301" />
</ItemGroup>
```

The Clean and Update processes can also be run from the Package Manager Console. Open the Package Manager Console, navigate to the directory containing the project file, and execute the following command:

```
dotnet bundle
```

To remove all generated files, enter the clean command, like this:

```
dotnet bundle clean
```

Creating the Web Service Locator

Earlier, the URL and the port for the RESTful service were added to the `appsettings.json` file. Now it's time to leverage the configuration system to create a call that will expose the value to the rest of the application. The class will be hooked into the DI framework so it can be injected into any class that needs the URL and port information.

Creating the IWebServiceLocator Interface

Create a new directory named `Configuration` in the `SpyStore.MVC` project. In this directory, create an interface name `IWebServiceLocator`, clear out the templated code, and add the following:

```
namespace SpyStore.MVC.Configuration
{
  public interface IWebServiceLocator
  {
    string ServiceAddress { get; }
  }
}
```

Creating the WebServiceLocator Class

In the `Configuration` directory, create another class named `WebServiceLocator.cs`. Clear out the code in the class and add the following (including the using statement):

```
using Microsoft.Extensions.Configuration;

namespace SpyStore.MVC.Configuration
{
  public class WebServiceLocator : IWebServiceLocator
  {
```

```

public WebServiceLocator(IConfigurationRoot config)
{
    var customSection = config.GetSection(nameof(WebServiceLocator));
    ServiceAddress = customSection?.GetSection(nameof(ServiceAddress))?.Value;
}

public string ServiceAddress { get; }
}
}

```

The constructor takes an instance of `IConfigurationRoot` from the DI container and uses it to get the value of the web service's URL from the `appsettings.json` file.

Adding the `WebServiceLocator` Class to the DI Container

The final step is to add the `WebServiceLocator` to the DI container. Open `Startup.cs` and add the following using statements to the top of the file:

```
using SpyStore.MVC.Configuration;
```

Next, navigate to the `ConfigureServices` method and add the following line before the call to `AddMvc`:

```
services.AddSingleton<IWebServiceLocator, WebServiceLocator>();
```

When code in the application requests a parameter of type `IWebServiceLocator`, a new instance of the `WebServiceLocator` class is supplied.

Creating the `WebAPICalls` Class to Call the Web API Service

The `WebAPICalls` class will encapsulate all of the API calls to the Core MVC Service built in the previous chapter. Create a new directory named `WebServiceAccess` in the `SpyStore.MVC` project, and in that directory, add a subdirectory named `Base`.

■ **Note** If you used .NET to call Web API services before, the rest of the code in this section is pretty standard and not .NET Core or MVC specific. If you feel comfortable calling RESTful services, the completed files are located in the `WebServiceAccess` directory in the downloadable code for this chapter. Copy them into your project and skip ahead to “Adding the `WebAPICalls` Class to the DI Container” section.

Creating the `IWebAPICalls` Interface

Add a new interface named `IWebAPICalls.cs` into the `Base` directory just created. Open the `IWebAPICalls.cs` interface file and update the code to the following:

```

using System.Collections.Generic;
using System.Threading.Tasks;
using SpyStore.Models.Entities;
using SpyStore.Models.ViewModels;
using SpyStore.Models.ViewModels.Base;

```

```

namespace SpyStore.MVC.WebServiceAccess.Base
{
    public interface IWebApiCalls
    {
        Task<IList<Category>> GetCategoriesAsync();
        Task<Category> GetCategoryAsync(int id);
        Task<IList<ProductAndCategoryBase>> GetProductsForACategoryAsync(int categoryId);
        Task<IList<Customer>> GetCustomersAsync();
        Task<Customer> GetCustomerAsync(int id);
        Task<IList<Order>> GetOrdersAsync(int customerId);
        Task<OrderWithDetailsAndProductInfo> GetOrderDetailsAsync(int customerId, int orderId);
        Task<ProductAndCategoryBase> GetOneProductAsync(int productId);
        Task<IList<ProductAndCategoryBase>> GetFeaturedProductsAsync();
        Task<IList<ProductAndCategoryBase>> SearchAsync(string searchTerm);
        Task<IList<CartRecordWithProductInfo>> GetCartAsync(int customerId);
        Task<CartRecordWithProductInfo> GetCartRecordAsync(int customerId, int productId);
        Task<string> AddToCartAsync(int customerId, int productId, int quantity);
        Task<string> UpdateCartItemAsync(ShoppingCartRecord item);
        Task RemoveCartItemAsync(int customerId, int shoppingCartRecordId, byte[] timeStamp);
        Task<int> PurchaseCartAsync(Customer customer);
    }
}

```

Creating the Base Class Code

In the Base directory, add a class named `WebApiCallsBase.cs`. This class will hold all of the core code to execute HTTP Get, Put, Post, and Delete commands. Update the namespaces to match the following:

```

using System;
using System.Collections.Generic;
using System.Net.Http;
using System.Text;
using System.Threading.Tasks;
using Newtonsoft.Json;
using SpyStore.MVC.Configuration;

```

Make the class public and abstract, then add class level variables to hold the URI for the web service and the core URIs for the service controllers. The code should look like this:

```

namespace SpyStore.MVC.WebServiceAccess.Base
{
    public abstract class WebApiCallsBase
    {
        protected readonly string ServiceAddress;
        protected readonly string CartBaseUri;
        protected readonly string CategoryBaseUri;
        protected readonly string CustomerBaseUri;
        protected readonly string ProductBaseUri;
        protected readonly string OrdersBaseUri;
    }
}

```


The `WebServiceLocator` is used to get the base URL of the web service, and the best way for this class to get an instance of the `WebServiceLocator` is through dependency injection. Create a protected constructor that accepts an instance of `IWebServiceLocator` and uses it to get the base URI. Next, the constructor builds the controller URIs, as follows:

```
protected WebApiCallsBase(IWebServiceLocator settings)
{
    ServiceAddress = settings.ServiceAddress;
    CartBaseUri = $"{ServiceAddress}api/ShoppingCart/";
    CategoryBaseUri = $"{ServiceAddress}api/category/";
    CustomerBaseUri = $"{ServiceAddress}api/customer/";
    ProductBaseUri = $"{ServiceAddress}api/product/";
    OrdersBaseUri = $"{ServiceAddress}api/orders/";
}
```

Implementing the Base HTTP Get Calls

The first set of methods are for Get calls. The `GetJsonFromGetResponseAsync` method handles executing the call to the service and returning the contents of the response message as JSON. The `GetItemAsync` and `GetItemListAsync` call the `GetJsonFromGetResponseAsync` method to return a single item or a list of items, respectively. Both methods use the `GetAsync` helper method from the `System.Net.HTTP` namespace. This helper method and its three siblings (`PostAsync`, `PutAsync`, and `DeleteAsync`) are shortcuts to creating an `HttpRequestMessage` from scratch and executing a `Send` on the `HttpClient`. Here are all three methods to be added to the base class:

```
internal async Task<string> GetJsonFromGetResponseAsync(string uri)
{
    try
    {
        using (var client = new HttpClient())
        {
            var response = await client.GetAsync(uri);
            if (!response.IsSuccessStatusCode)
            {
                throw new Exception($"The Call to {uri} failed. Status code: {response.
                StatusCode}");
            }
            return await response.Content.ReadAsStringAsync();
        }
    }
    catch (Exception ex)
    {
        //Do something intelligent here
        Console.WriteLine(ex);
        throw;
    }
}
```

```

internal async Task<T> GetItemAsync<T>(string uri) where T : class, new()
{
    try
    {
        var json = await GetJsonFromGetResponseAsync(uri);
        return JsonConvert.DeserializeObject<T>(json);
    }
    catch (Exception ex)
    {
        //Do something intelligent here
        Console.WriteLine(ex);
        throw;
    }
}

internal async Task<IList<T>> GetItemListAsync<T>(string uri) where T : class, new()
{
    try
    {
        return JsonConvert.DeserializeObject<IList<T>>(await GetJsonFromGetResponseAsync(uri));
    }
    catch (Exception ex)
    {
        //Do something intelligent here
        Console.WriteLine(ex);
        throw;
    }
}

```

Implementing the Base HTTP Post and Put Calls

The next method to create is used to handle response messages from Post and Put calls. In both cases, a request is executed, the response is checked for success, and then the content of the message body is returned. The `ExecuteRequestAndProcessResponse` code is listed here:

```

protected static async Task<string> ExecuteRequestAndProcessResponse(
    string uri, Task<HttpResponseMessage> task)
{
    try
    {
        var response = await task;
        if (!response.IsSuccessStatusCode)
        {
            throw new Exception($"The Call to {uri} failed. Status code: {response.StatusCode}");
        }
        return await response.Content.ReadAsStringAsync();
    }
    catch (Exception ex)
    {

```

```

    //Do something intelligent here
    Console.WriteLine(ex);
    throw;
}
}

```

The next helper method creates an instance of the `StringContent` class that is used to format the payload for request messages:

```

protected StringContent CreateStringContent(string json)
{
    return new StringContent(json, Encoding.UTF8, "application/json");
}

```

`SubmitPostRequestAsync` and `SubmitPutRequestAsync` use the `PostAsync` and `PutAsync` helpers (along with the `CreateStringContent` helper) to create their respective tasks. Those tasks are then sent into the `ExecuteRequestAndProcessResponse` method for processing. Here are the methods:

```

protected async Task<string> SubmitPostRequestAsync(string uri, string json)
{
    using (var client = new HttpClient())
    {
        var task = client.PostAsync(uri, CreateStringContent(json));
        return await ExecuteRequestAndProcessResponse (uri, task);
    }
}

```

```

protected async Task<string> SubmitPutRequestAsync(string uri, string json)
{
    using (var client = new HttpClient())
    {
        Task<HttpResponseMessage> task = client.PutAsync(uri, CreateStringContent(json));
        return await ExecuteRequestAndProcessResponse (uri, task);
    }
}

```

Implementing the Base HTTP Delete Call

The final base method to add is for executing an HTTP Delete. This method is pretty straightforward as a Delete request doesn't have a body and doesn't return any content. Add the following code to the base class:

```

protected async Task SubmitDeleteRequestAsync(string uri)
{
    try
    {
        using (var client = new HttpClient())
        {
            Task<HttpResponseMessage> deleteAsync = client.DeleteAsync(uri);
            var response = await deleteAsync;
        }
    }
}

```

```

        if (!response.IsSuccessStatusCode)
        {
            throw new Exception(response.StatusCode.ToString());
        }
    }
}
catch (Exception ex)
{
    //Do something intelligent here
    Console.WriteLine(ex);
    throw;
}
}
}

```

Creating the WebApiCalls Class

Next, create a class named `WebApiCalls.cs` in the `WebServiceAccess` directory. Add the following using statements to the top of the file:

```

using System.Collections.Generic;
using System.Threading.Tasks;
using Newtonsoft.Json;
using SpyStore.MVC.Configuration;
using SpyStore.Models.Entities;
using SpyStore.Models.ViewModels;
using SpyStore.Models.ViewModels.Base;
using SpyStore.MVC.WebServiceAccess.Base;

```

Inherit from `WebApiCallsBase` and implement the `IWebApiCalls` interface. Have the public constructor accept an `IWebServiceLocator` instance and pass it to the base class constructor. The code should look like this:

```

public class WebApiCalls : WebApiCallsBase, IWebApiCalls
{
    public WebApiCalls(IWebServiceLocator settings) : base(settings)
    {
    }
}

```

The methods in this class call the base methods using specific values passed from the controllers. Each method in the following listing starts with a comment describing the actual call to the Web API service, and then the actual code to make the call. Enter the following code (or copy it from the downloadable files):

```

public async Task<IList<CartRecordWithProductInfo>> GetCartAsync(int customerId)
{
    // http://localhost:40001/api/ShoppingCart/0
    return await GetItemListAsync<CartRecordWithProductInfo>($"{CartBaseUri}{customerId}");
}
public async Task<CartRecordWithProductInfo> GetCartRecordAsync(int customerId, int
productId)

```

```

{
    // http://localhost:40001/api/ShoppingCart/0/0
    return await GetItemAsync<CartRecordWithProductInfo>(
        $"{CartBaseUri}{customerId}/{productId}");
}
public async Task<string> AddToCartAsync(int customerId, int productId, int quantity)
{
    //http://localhost:40001/api/shoppingcart/{customerId} HTTPPost
    //Note: ProductId and Quantity in the body
    //http://localhost:40001/api/shoppingcart/0 {"ProductId":22,"Quantity":2}
    //      Content-Type:application/json
    string uri = $"{CartBaseUri}{customerId}";
    string json = $"{{\"ProductId\":{productId},\"Quantity\":{quantity}}}\";
    return await SubmitPostRequestAsync(uri, json);
}
public async Task<int> PurchaseCartAsync(Customer customer)
{
    //Purchase: http://localhost:40001/api/shoppingcart/{customerId}/buy HTTPPost
    //Note: Customer in the body
    //{"Id":1,"FullName":"Super Spy","EmailAddress":"spy@secrets.com"}
    // http://localhost:40001/api/shoppingcart/0/buy
    var json = JsonConvert.SerializeObject(customer);
    var uri = $"{CartBaseUri}{customer.Id}/buy";
    return int.Parse(await SubmitPostRequestAsync(uri, json));
}
public async Task<string> UpdateCartItemAsync(ShoppingCartRecord item)
{
    // Change Cart Item(Qty): http://localhost:40001/api/shoppingcart/{customerId}/{id}
    HTTPPut
    // Note: Id, CustomerId, ProductId, TimeStamp, DateCreated, and Quantity in the body
    //{"Id":0,"CustomerId":0,"ProductId":32,"Quantity":2,
    // "TimeStamp":"AAAAAAAA86s=", "DateCreated":"1/20/2016"}
    //http://localhost:40001/api/shoppingcart/0/AAAAAAAA86s=
    string uri = $"{CartBaseUri}{item.CustomerId}/{item.Id}";
    var json = JsonConvert.SerializeObject(item);
    return await SubmitPutRequestAsync(uri, json);
}
public async Task RemoveCartItemAsync(
    int customerId, int shoppingCartRecordId, byte[] timeStamp)
{
    //Remove Cart Item:
    // http://localhost:40001/api/shoppingcart/{customerId}/{id}/{TimeStamp} HTTPDelete
    // http://localhost:40001/api/shoppingcart/0/0/AAAAAAAA1Uc=
    var timeStampString = JsonConvert.SerializeObject(timeStamp);
    var uri = $"{CartBaseUri}{customerId}/{shoppingCartRecordId}/{timeStampString}";
    await SubmitDeleteRequestAsync(uri);
}
public async Task<IList<Category>> GetCategoriesAsync()
{
    //http://localhost:40001/api/category
    return await GetItemListAsync<Category>(CategoryBaseUri);
}

```

```

public async Task<Category> GetCategoryAsync(int id)
{
    //http://localhost:40001/api/category/{id}
    return await GetItemAsync<Category>($"{{CategoryBaseUri}}{id}");
}
public async Task<IList<ProductAndCategoryBase>> GetProductsForACategoryAsync(int
categoryId)
{
    // http://localhost:40001/api/category/{categoryId}/products
    var uri = $"{{CategoryBaseUri}}{categoryId}/products";
    return await GetItemListAsync<ProductAndCategoryBase>(uri);
}
public async Task<IList<Customer>> GetCustomersAsync()
{
    //Get All Customers: http://localhost:40001/api/customer
    return await GetItemListAsync<Customer>($"{{CustomerBaseUri}}");
}
public async Task<Customer> GetCustomerAsync(int id)
{
    //Get One customer: http://localhost:40001/api/customer/{id}
    //http://localhost:40001/api/customer/1
    return await GetItemAsync<Customer>($"{{CustomerBaseUri}}{id}");
}
public async Task<IList<ProductAndCategoryBase>> GetFeaturedProductsAsync()
{
    // http://localhost:40001/api/product/featured
    return await GetItemListAsync<ProductAndCategoryBase>($"{{ProductBaseUri}}featured");
}
public async Task<ProductAndCategoryBase> GetOneProductAsync(int productId)
{
    // http://localhost:40001/api/product/{id}
    return await GetItemAsync<ProductAndCategoryBase>($"{{ProductBaseUri}}{productId}");
}
public async Task<IList<Order>> GetOrdersAsync(int customerId)
{
    //Get Order History: http://localhost:40001/api/orders/{customerId}
    return await GetItemListAsync<Order>($"{{OrdersBaseUri}}{customerId}");
}
public async Task<OrderWithDetailsAndProductInfo> GetOrderDetailsAsync(
int customerId, int orderId)
{
    //Get Order Details: http://localhost:40001/api/orders/{customerId}/{orderId}
    var url = $"{{OrdersBaseUri}}{customerId}/{orderId}";
    return await GetItemAsync<OrderWithDetailsAndProductInfo>(url);
}
public async Task<IList<ProductAndCategoryBase>> SearchAsync(string searchTerm)
{
    var uri = $"{{ServiceAddress}}api/search/{searchTerm}";
    return await GetItemListAsync<ProductAndCategoryBase>(uri);
}

```

Adding WebApiCalls Class to the DI Container

The final step is to add the `WebApiCalls` to the DI container. Open `Startup.cs` and add the following using statements to the top of the file:

```
using SpyStore.MVC.WebServiceAccess;
using SpyStore.MVC.WebServiceAccess.Base;
```

Next, navigate to the `ConfigureServices` method and add the following line (after the line adding the `WebServiceLocator` class):

```
services.AddSingleton<IWebApiCalls, WebApiCalls>();
```

Adding the Fake Authentication

The `SpyStore.MVC` application uses a fake security mechanism that automatically “logs in” the single user contained in the sample data in the database. The fake authentication system is very simplistic—a class to call the web service to get customer information, and an action filter to set `ViewBag` properties for the customer ID and name.

■ **Note** If you are new to ASP.NET MVC, the `ViewBag` is a dynamic wrapper around the `ViewData` object that allows for passing data from a controller to a view. It is a one-way trip for the data, so the footprint is very small and short lived.

Building the Authentication Helper

Create a new directory named `Authentication` in the `SpyStore.MVC` project, and in this directory add an interface named `IAuthHelper.cs`. Clear out the existing code and replace it with the following:

```
using Microsoft.AspNetCore.Http;
using SpyStore.Models.Entities;

namespace SpyStore.MVC.Authentication
{
    public interface IAuthHelper
    {
        Customer GetCustomerInfo();
    }
}
```

Next, add a class named `AuthHelper.cs` to the directory. Clear out the existing code and replace it with this:

```
using System;
using System.Linq;
using Microsoft.AspNetCore.Http;
using SpyStore.MVC.WebServiceAccess.Base;
using SpyStore.Models.Entities;
```

```

namespace SpyStore.MVC.Authentication
{
    public class AuthHelper : IAuthHelper
    {
        private readonly IWebApiCalls _webApiCalls;
        public AuthHelper(IWebApiCalls webApiCalls)
        {
            _webApiCalls = webApiCalls;
        }
        public Customer GetCustomerInfo()
        {
            return _webApiCalls.GetCustomersAsync().Result.FirstOrDefault();
        }
    }
}

```

The class implements the `IAuthHelper` interface, which enables it to be loaded from the DI container. The constructor takes an instance of the `IWebApiCalls` interface (which is also loaded from the DI container), and then assigns it to a class level variable. The `GetCustomerInfo` returns the first customer record it finds.

Adding the AuthHelper Class to the DI Container

Add the `AuthHelper` class to the DI container by opening the `Startup.cs` file and adding the following using statement to the top of the file:

```
using SpyStore.MVC.Authentication;
```

Next, navigate to the `ConfigureServices` method and add the following line (after the line adding the `WebApiCalls` class):

```
services.AddSingleton<IAuthHelper, AuthHelper>();
```

The entire method should look like this:

```

public void ConfigureServices(IServiceCollection services)
{
    // Add framework services.
    services.AddSingleton(_ => Configuration);
    services.AddSingleton<IWebServiceLocator, WebServiceLocator>();
    services.AddSingleton<IWebApiCalls, WebApiCalls>();
    services.AddSingleton<IAuthHelper, AuthHelper>();
    services.AddMvc();
}

```

Creating the Action Filter for the Fake Authentication

In Chapter 3, you built an exception filter to catch any unhandled exceptions in action methods. Action filters are very similar. Instead of catching exceptions, they have two event handlers—one fires just before an action method executes and the other fires right after it executes. Like exception filters, action filters can be applied to a single action, controller, or across an entire application.

Create a new directory named `Filters` in the `SpyStore.MVC` project and add a new class named `AuthActionFilter.cs`. Add the following using statements to the top of the class:

```
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.Filters;
using SpyStore.MVC.Authentication;
```

Next, change the class to public and implement the `IActionFilter` interface, like this:

```
public class AuthActionFilter : IActionFilter
{
    public void OnActionExecuting(ActionExecutingContext context)
    {
        // do something before the action executes
    }

    public void OnActionExecuted(ActionExecutedContext context)
    {
        // do something after the action executes
    }
}
```

The filter needs an instance of the `IAuthHelper` injected in. Create a constructor that takes an `IAuthHelper` as the parameter. In the constructor, assign the injected instance to a class level variable, like this:

```
private IAuthHelper _authHelper;
public AuthActionFilter(IAuthHelper authHelper)
{
    _authHelper = authHelper;
}
```

The `ActionExecutingContext` grants access to the current `HttpContext`, which then grants access to the current `HttpRequest` (if needed) and the MVC `ViewBag`. The goal of the method is to get the `Id` and `FullName` values from the `Customer` record and place them into the `ViewBag`. The entire method is shown here:

```
public void OnActionExecuting(ActionExecutingContext context)
{
    var viewBag = ((Controller)context.Controller).ViewBag;
    var customer = _authHelper.GetCustomerInfo();
    viewBag.CustomerId = customer.Id;
    viewBag.CustomerName = customer.FullName;
}
```

An alternative to injecting the service into the filter is to use the `RequestServices` method, which provides access to all of the services in the DI container, as shown in the following line:

```
(IAuthHelper)context.HttpContext.RequestServices.GetService(typeof(IAuthHelper));
```

Adding the Action Filter for All Actions

Open `Startup.cs` and add the following `using` statement to the top of the file:

```
using SpyStore.MVC.Filters;
```

Next, navigate to the `ConfigureServices` method. Change the `services.AddMvc` line to the following:

```
services.AddMvc(config => {
    config.Filters.Add(
        new AuthActionFilter(services.BuildServiceProvider().GetService<IAuthHelper>()));
});
```

This sets the `AuthActionFilter` for all actions in the application.

Adding the View Models

As discussed when creating the data access layer, view models combine different base models into one entity class as a convenience when passing data between layers. In addition to the view models already created, a few more are needed for the UI.

Create a new directory named `ViewModels` in the root of the `SpyStore.MVC` project. Create a subdirectory named `Base` in the `ViewModels` directory and add a new class named `CartItemViewModelBase.cs`. Add the following `using` statements to the top of the class:

```
using System.ComponentModel.DataAnnotations;
using Newtonsoft.Json;
using SpyStore.Models.ViewModels.Base;
```

Make this class public and inherit from `ProductAndCategoryBase`. Add the following properties and attributes so the class looks like this:

```
public class CartItemViewModelBase : ProductAndCategoryBase
{
    public int? CustomerId { get; set; }
    [DataType(DataType.Currency), Display(Name = "Total")]
    public decimal LineItemTotal { get; set; }
    public string TimestampString =>
        Timestamp != null ? JsonConvert.SerializeObject(Timestamp).Replace("\"", "") : string.
        Empty;
}
```

The `TimestampString` property serializes the `Timestamp` byte array into a string value to make it easier to work with in views.

Next, in the `ViewModels` directory, add a new class named `AddToCartItemViewModel.cs`. Add the following `using` statement:

```
using SpyStore.MVC.ViewModels.Base;
```

Make the class public, inherit from `CartItemViewModelBase`, and add a single property:

```
public class AddToCartItemViewModel :CartItemViewModelBase
{
    public int Quantity { get; set; }
}
```

Add another class named `CartItemRecordViewModel.cs` and update the code as follows:

```
using SpyStore.MVC.ViewModels.Base;

namespace SpyStore.MVC.ViewModels
{
    public class CartItemRecordViewModel : CartItemViewModelBase
    {
        public int Quantity { get; set; }
    }
}
```

You might be wondering why there are two classes with the exact same properties but different names. The reason is that the business rules for the uses of the classes are slightly different. This will all become clear in the next chapter when validation is added to the application.

Add one more class, named `CartItemViewModel`, and update the using statements and properties to match the following:

```
using SpyStore.Models.Entities;
using System.Collections.Generic;

namespace SpyStore.MVC.ViewModels
{
    public class CartItemViewModel
    {
        public Customer Customer { get; set; }
        public IList<CartItemRecordViewModel> CartRecords { get; set; }
    }
}
```

■ **Source Code** This chapter's portion of the `SpyStore.MVC` solution can be found in the `Chapter_04` subdirectory of the download files.

Summary

This chapter introduced Core MVC Web Applications and covered some of the differences between the project structure for Core MVC RESTful services and Core MVC Web Applications. The project structure was detailed and updated as appropriate for the SpyStore MVC application. Routing was revisited, discussing the differences between attribute routing and adding routes directly to the routing table. Controllers and actions were revisited to cover the additional features of MVC Web Applications, and views were introduced, as were Display Templates and Editor Templates.

The next two sections covered the infrastructure items of using Bower to manage client side packages and BundlerMinifier to handle bundling and minification of JavaScript and CSS file.

The configuration service was utilized to build the Web Service Locator. All of the API calls to the RESTful service were created in the `WebAPICalls` class and base class. Next, these were leveraged to create the fake authentication handler and the action filter. Finally, the applications view models were created.

In the next chapter, all of this groundwork will be used to create the SpyStore MVC Web Application.

CHAPTER 5



Building the SpyStore Application with ASP.NET Core MVC

The previous chapter built the Core MVC Web Application project and the infrastructure necessary for the SpyStore MVC application and covered many of the Core MVC concepts. This chapter is almost entirely code based, but also covers the two newest features in Core MVC—Tag Helpers and View Components.

■ **Note** This chapter picks up where the last chapter ended. There is a folder named `Start` in the downloadable code that has the completed code from Chapter 4.

Tag Helpers

Tag Helpers are a new feature in Core MVC that greatly improve the development experience and readability of MVC Views. Unlike HTML Helpers, which are invoked as Razor methods, Tag Helpers are attributes added to HTML elements. Tag Helpers encapsulate server side code that shapes the attached element. If you are developing with Visual Studio, then there is an added benefit of IntelliSense for the built-in Tag Helpers.

For example, the following HTML Helper creates a label for the customer's `FullName`:

```
@Html.Label("FullName", "Full Name:", new { @class = "customer" })
```

This generates the following HTML:

```
<label class="customer" for="FullName">Full Name:</label>
```

For the C# developer who has been working with ASP.NET MVC and Razor, the HTML helper syntax is understood. But it's not intuitive. Especially for someone who works in HTML/CSS/JavaScript, and not C#.

The corresponding Tag Helper would be written into the view like this:

```
<label class="customer" asp-for="FullName">Full Name:</label>
```

This produces the same output. And, if the `FullName` property contained the `Display` attribute (which it does), then the following markup will use the `Name` property on the attribute to produce the same output:

```
<label class="customer" asp-for="FullName"/>
```

There are many built-in Tag Helpers, and they are designed to be used instead of their respective HTML Helpers. However, not all HTML Helpers have an associated Tag Helper. Table 5-1 lists the available Tag Helpers, their corresponding HTML Helper, and the available attributes. Each is explained in the next sections, except for the Label Tag Helper, which was already discussed.

Table 5-1. Built-in Tag Helpers

Tag Helper	HTML Helper	Available Attributes
Form	Html.BeginForm Html.BeginRouteForm Html.AntiForgeryToken	asp-route - for named routes (can't be used with controller or action attributes) asp-antiforgery - if the antiforgery should be added asp-area - the name of the area asp-controller - defines the controller asp-action - defines the action asp-route- <code><ParameterName></code> - adds the parameter to the route, e.g., <code>asp-route-id="1"</code> asp-all-route-data- dictionary for additional route values Automatically adds the antiforgery token
Anchor	Html.ActionLink	asp-route - for named routes (can't be used with controller or action attributes) asp-area - the name of the area asp-controller - defines the controller asp-action - defines the action asp-protocol - http or https asp-fragment - URL fragment asp-host - the host name asp-route- <code><ParameterName></code> - adds the parameter to the route, e.g., <code>asp-route-id="1"</code> asp-all-route-data- dictionary for additional route values
Input	Html.TextBox/TextBoxFor Html.Editor/EditorFor	asp-for - a model property. Can navigate the model (Customer.Address.AddressLine1) and use expressions (asp-for="@localVariable") The id and name attributes are auto-generated Any HTML5 data-val attributes are auto-generated
TextArea	Html.TextAreaFor	asp-for - a model property. Can navigate the model (Customer.Address.Description) and use expressions (asp-for="@localVariable") The id and name attributes are auto-generated Any HTML5 data-val attributes are auto-generated
Label	Html.LabelFor	asp-for - a model property. Can navigate the model (Customer.Address.AddressLine1) and use expressions (asp-for="@localVariable")
Select	Html.DropDownListFor Html.ListBoxFor	asp-for - a model property. Can navigate the model (Customer.Address.AddressLine1) and use expressions (asp-for="@localVariable") asp-items - specifies the options elements Auto-generates the selected="selected" attribute The id and name attributes are auto-generated Any HTML5 data-val attributes are auto-generated

(continued)

Table 5-1. (continued)

Tag Helper	HTML Helper	Available Attributes
Validation Message (Span)	Html.ValidationMessageFor	asp-validation-for – a model property. Can navigate the model (Customer.Address.AddressLine1) and use expressions (asp-for="@localVariable") Adds the data-valmsg-for attribute to the span
Validation Summary (Div)	Html.ValidationSummaryFor	asp-validation-summary – select one of All, ModelOnly, or None Adds the data-valmsg-summary attribute to the div
Link	N/A	asp-append-version – appends version to filename asp-fallback-href – fallback file to use if primary is not available; usually used with CDN sources asp-fallback-href-include – globbed file list of files to include on fallback asp-fallback-href-exclude – globbed file list of files to exclude on fallback asp-fallback-test-* – properties to use on fallback test. Included are class, property, and value asp-href-include – globbed file pattern of files to include asp-href-exclude – globbed file pattern of files to exclude
Script	N/A	asp-append-version – appends version to file name asp-fallback-src – fallback file to use if primary is not available; usually used with CDN sources asp-fallback-src-include – globbed file list of files to include on fallback asp-fallback-src-exclude – globbed file list of files to exclude on fallback asp-fallback-test – the script method to use in fallback test. asp-src-include – globbed file pattern of files to include asp-src-exclude – globbed file pattern of files to exclude
Image	N/A	asp-append-version – appends version to filename
Environment	N/A	name – one of Development, Staging, or Production

In addition to the pre-supplied tag helpers, custom tag helpers can be created as well.

Enabling Tag Helpers

Tag Helpers, like any other code, must be made visible to any code that wants to use them. The `_ViewImports.html` file contains the following line:

```
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
```

This makes all Tag Helpers in the `Microsoft.AspNetCore.Mvc.TagHelpers` assembly (which contains all of the built-in Tag Helpers) available to all of the Views.

The Form Tag Helper

The Form Tag Helper replaces the `Html.BeginForm` and `Html.BeginRouteForm` HTML Helpers. For example, to create a form that submits to the HTTP Post version of the `AddToCart` action on the `CartController` with two parameters, `CustomerId` and `ProductId`, use the following code:

```
<form method="post"
  asp-controller="Cart"
  asp-action="AddToCart"
  asp-route-customerId="@Model.CustomerId"
  asp-route-productId="@Model.Id">
<!-- Omitted for brevity -->
</form>
```

If the Form tag was rendered in a view by the HTTP Get version of the `AddToCart` action of the `CartController`, the `asp-controller` and `asp-action` attributes are optional. As mentioned in Table 5-1, the antiforgery token is by default automatically rendered by the Form Tag Helper.

The Anchor Tag Helper

The Anchor Tag Helper replaces the `Html.ActionLink` HTML Helper. For example, to create a link for the site menu, use the following code:

```
<a asp-controller="Products" asp-action="ProductList" asp-route-id="@item.Id">
  @item.CategoryName
</a>
```

The Tag Helper creates the URL from the route table, using the `ProductsController`, `ProductList` Action method, and the current value for the `Id`. The `CategoryName` property becomes the anchor tag text. The resulting URL looks like this:

```
<a href="/Products/ProductList/0">Communications</a>
```

The Input Tag Helper

The Input Tag Helper is the most versatile of the Tag Helpers. In addition to auto-generating the HTML `id` and `name` attributes, as well as any HTML5 `data-val` validation attributes, the Tag Helper builds the appropriate HTML markup based on the datatype of the target property. Table 5-2 lists the HTML type that is created based on the .NET type of the property.

Table 5-2. HTML Types Generated from .NET Types Using the Input Tag Helper

.NET Type	Generated HTML Type
Bool	type="checkbox"
String	type="text"
DateTime	type="datetime"
Byte, Int, Single, Double	type="number"

Additionally, the Input Tag Helper will add HTML5 type attributes based on data annotations. Table 5-3 lists some of the most common annotations and the generated HTML5 type attributes.

Table 5-3. *HTML5 Types Attributes Generated from .NET Data Annotations*

.NET Data Annotation	Generated HTML5 Type Attribute
EmailAddress	type="email"
Url	type="url"
HiddenInput	type="hidden"
Phone	type="tel"
DataType(DataType.Password)	type="password"
DataType(DataType.Date)	type="date"
DataType(DataType.Time)	type="time"

For example, to display a text box for editing the Quantity value on a model, enter the following:

```
<input asp-for="Quantity" class="cart-quantity" />
```

Presuming the value of the Quantity field is 3, the following HTML gets generated:

```
<input name="Quantity" class="cart-quantity" id="Quantity" type="number" value="3" data-val-required="The Quantity field is required." data-val="true" data-val-notgreaterthan-prefix=" " data-val-notgreaterthan-otherpropertyname="UnitsInStock" data-val-notgreaterthan="Quantity must not be greater than In Stock">
```

The id, name, value, and validation attributes are all added automatically.

The TextArea Tag Helper

The TextArea Tag Helper adds the id and name attributes automatically and any HTML5 validation tags defined for the property. For example, the following line creates a textarea tag for the Description property:

```
<textarea asp-for="Description"></textarea>
```

The resulting markup is this:

```
<textarea name="Description" id="Description" data-val="true" data-val-maxlength-max="3800" data-val-maxlength="The field Description must be a string or array type with a maximum length of '3800'.">Disguised as typewriter correction fluid (rest omitted for brevity)</textarea>
```

The Select Tag Helper

The Select Tag Helper builds input select tags from a model property and a collection. As with the other input Tag Helpers, the id and name are added to the markup, as well as any HTML5 data-val attributes. If the model property value matches one of the select list item's values, that option gets the selected attribute added to the markup.

For example, take a model that has a property named `Country` and a `SelectList` named `Countries`, with the list defined as follows:

```
public List<SelectListItem> Countries { get; } = new List<SelectListItem>
{
    new SelectListItem { Value = "MX", Text = "Mexico" },
    new SelectListItem { Value = "CA", Text = "Canada" },
    new SelectListItem { Value = "US", Text = "USA" },
};
```

The following markup will render the select tag with the appropriate options:

```
<select asp-for="Country" asp-items="Model.Countries"></select>
```

If the value of the `Country` property is set to `CA`, the following full markup will be output to the view:

```
<select id="Country" name="Country">
  <option value="MX">Mexico</option>
  <option selected="selected" value="CA">Canada</option>
  <option value="US">USA</option>
</select>
```

This just scratches the surface of the Select Tag Helper. For more information, refer to the documentation located here: <https://docs.microsoft.com/en-us/aspnet/core/mvc/views/working-with-forms#the-select-tag-helper>.

The Validation Tag Helpers

The Validation Message and Validation Summary Tag Helpers closely mirror the `Html.ValidationMessageFor` and `Html.ValidationSummaryFor` HTML Helpers. The first is applied to a `span` HTML tag for a specific property on the model, and the latter is applied to a `div` tag, and represents the entire model. The Validation Summary has the option of `All` errors, `ModelOnly` (excluding errors on model properties), or `None`.

The following code adds a Validation Summary for model-level errors and a Validation Message for the `Quantity` property of the model:

```
<div asp-validation-summary="ModelOnly" class="text-danger"></div>
<span asp-validation-for="Quantity" class="text-danger"></span>
```

If there was a model-level error and an error on the `Quantity` property, the output would resemble this:

```
<div class="text-danger validation-summary-errors">
  <ul>
    <li>There was an error adding the item to the cart.</li>
  </ul>
</div>
<span class="text-danger field-validation-error" data-valmsg-replace="true" data-valmsg-for="Quantity">Quantity must not be greater than In Stock</span>
```

Validation and model binding are covered in detail later in the chapter.

The Link and Script Tag Helpers

The most commonly used attributes of the Link and Script Tag Helpers are the `asp-append-version` and `asp-fallback-*` attributes. The `asp-append-version` attribute adds the hash of the file to the name so the cache is invalidated when the file changes and the file is reloaded. The following is an example of the `asp-append-version` attribute:

```
<link rel="stylesheet" href="~/css/site.css" asp-append-version="true"/>
```

The rendered code is as follows:

```
<link href="/css/site.css?v=v9cmzjNgxPHiyLIrNom5fw3tZj3TNT2QD7a0hBrSa4U" rel="stylesheet">
```

The `asp-fallback-*` attributes are typically used with CDN file sources. For example, the following code attempts to load jquery from the Microsoft CDN. If it fails, it loads the local version:

```
<script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-3.1.1.min.js"
  asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
  asp-fallback-test="window.jQuery">
```

The Image Tag Helper

The Image Tag Helper provides the `asp-append-version` attribute, which works the same as described in the Link and Script Tag Helpers.

The Environment Tag Helper

The Environment Tag Helper is typically used to conditionally load files based on current environment that the site is running under. The following code will load the un-minified files if the site is running in Development, and the minified versions if the site is running in Staging or Production:

```
<environment names="Development">
  <link rel="stylesheet" href="~/css/site.css" asp-append-version="true"/>
  <link rel="stylesheet" href="~/css/spystore-bootstrap.css" asp-append-version="true"/>
</environment>
<environment names="Staging,Production">
  <link rel="stylesheet" href="~/css/site.min.css" />
  <link rel="stylesheet" href="~/css/spystore-bootstrap.min.css" asp-append-version="true" />
</environment>
```

Custom Tag Helpers

Custom Tag Helpers can also be created. Custom Tag Helpers can help eliminate repeated code, such as creating `mailto:` links. Create a new folder named `TagHelpers` in the root of the `SpyStore.MVC` project. In this folder, create a new class named `EmailTagHelper.cs` and inherit from `TagHelper`, as follows:

```
public class EmailTagHelper : TagHelper
{
}
```

Add the following using statement to the top of the file:

```
using Microsoft.AspNetCore.Razor.TagHelpers;
```

Add two properties to hold the EmailName and the EmailDomain, as follows:

```
public string EmailName { get; set; }
public string EmailDomain { get; set; }
```

Public properties are accessed as attributes of the tag helper, converted to lower-kebab-case. For the EmailTagHelper, they have values passed in like this:

```
<email email-name="blog" email-domain="skimedic.com"></email>
```

When a Tag Helper is invoked, the Process method is called. The Process method takes two parameters, a TagHelperContext and a TagHelperOutput. The TagHelperContext is used to get any other attributes on the tag and a dictionary of objects used to communicate with other tag helpers targeting child elements. The TagHelperOutput is used to create the rendered output.

Add the following method to the EmailTagHelper class:

```
public override void Process(TagHelperContext context, TagHelperOutput output)
{
    output.TagName = "a";    // Replaces <email> with <a> tag
    var address = EmailName + "@" + EmailDomain;
    output.Attributes.SetAttribute("href", "mailto:" + address);
    output.Content.SetContent(address);
}
```

Making Custom Tag Helpers Visible

To make custom Tag Helpers visible, the @addTagHelper command must be executed for any views that use the Tag Helpers. To apply this to all views, add the command to the _ViewImports.cshtml file. Open the _ViewImports.cshtml file in the root of the Views folder and add the following line to make the EmailTagHelper visible to the views in the application:

```
@addTagHelper *, SpyStore.MVC
```

Building the Controllers

The application consists of three controllers, one each for Products, Orders, and the ShoppingCart. The Products and Orders controllers are both very straight forward, as they are used to build display only pages. The ShoppingCartController is the workhorse of this sample application and is covered last.

Delete the HomeController file if you haven't already, as it's not used by this application.

The Products Controller

Create a new file named `ProductsController.cs` in the `Controllers` directory. Update the using statements to match the following:

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using SpyStore.Models.ViewModels.Base;
using SpyStore.MVC.Models;
using SpyStore.MVC.WebServiceAccess.Base;
```

Change the class to public and inherit from `Controller`, as follows:

```
public class ProductsController : Controller
```

The Constructor

The `ProductsController` needs an instance of the `IWebApiCalls` class to call into the service to get product data. Add the following property to the top of the class and the following constructor code:

```
private readonly IWebApiCalls _webApiCalls;
public ProductsController(IWebApiCalls webApiCalls)
{
    _webApiCalls = webApiCalls;
}
```

The Error Action

The `Error` action is invoked when there is an unhandled error in the application and the application is not running in `Development`. This was configured in the `Startup.cs` class `Configure` method, as shown here:

```
if (env.IsDevelopment())
{
    app.UseDeveloperExceptionPage();
    app.UseBrowserLink();
}
else
{
    app.UseExceptionHandler("/Products/Error");
}
```

The action simply returns a `ViewResult` that renders the `Error.cshtml` view:

```
[HttpGet]
public ActionResult Error()
{
    return View();
}
```

The Details Action

The Details action redirects to the AddToCart action of the CartController using the RedirectToAction method. The RedirectToAction method has several overloads, and in this instance, all are in use. The first parameter is the action method name and the second is the controller name, both passed in as strings. The C# nameof method is used to get the string names from the C# objects to help prevent typos and runtime errors in case of a name change. MVC strips off the Controller suffix of a controller name when creating routes, so the Replace function is used to remove the string “Controller” from the result of the nameof function. The final parameter is an anonymous object that contains route values. The first two values (customerId and productId) are used to create the route. Additional values can be passed in to an action method as simple parameters, and a bool is used to indicate that the source of the request is the ProductController Details action method.

```
public ActionResult Details(int id)
{
    return RedirectToAction(
        nameof(CartController.AddToCart),
        nameof(CartController).Replace("Controller", ""),
        new { customerId = ViewBag.CustomerId, productId = id, cameFromProducts = true });
}
```

■ **Note** This code won't compile until the CartController is added, which further illustrates the benefit to using the nameof function.

The GetListOfProducts Helper Method

The home page of the SpyStore site shows a list of featured Products. The user has the option to select one of the Category values to show all of the products in that Category as well as searching for products that contain the search phrase. All of them require a List of ProductAndCategoryBase records and pass that list to the ProductList.cshtml view. Create an internal method that uses the WebApiCalls utility to get the appropriate list (based on the parameters passed in from the calling method) and return the ProductList view with the list as the view's model. If no products are found, a NotFound IActionResult is returned.

```
internal async Task<IActionResult> GetListOfProducts(
    int id = -1, bool featured = false, string searchString = "")
{
    IList<ProductAndCategoryBase> prods;
    if (featured)
    {
        prods = await _webApiCalls.GetFeaturedProductsAsync();
    }
    else if (!string.IsNullOrEmpty(searchString))
    {
        prods = await _webApiCalls.SearchAsync(searchString);
    }
    else
    {
        prods = await _webApiCalls.GetProductsForACategoryAsync(id);
    }
}
```

```

    if (prods == null)
    {
        return NotFound();
    }
    return View("ProductList", prods);
}

```

The Index, Featured, and Search Actions

The previous helper method simplifies the Featured, ProductList, and Search action methods.

The Index and Featured Actions

For the Index and Featured actions, all that needs to be done is to set the correct ViewBag properties and then call the GetListOfProducts helper method. The Featured and ProductList methods are shown here:

```

[HttpGet]
public async Task<IActionResult> Featured()
{
    ViewBag.Title = "Featured Products";
    ViewBag.Header = "Featured Products";
    ViewBag.ShowCategory = true;
    ViewBag.Featured = true;
    return await GetListOfProducts(featured:true);
}

```

```

[HttpGet]
public async Task<IActionResult> ProductList(int id)
{
    var cat = await _webApiCalls.GetCategoryAsync(id);
    ViewBag.Title = cat?.CategoryName;
    ViewBag.Header = cat?.CategoryName;
    ViewBag.ShowCategory = false;
    ViewBag.Featured = false;
    return await GetListOfProducts(id);
}

```

The Search Action

The Search action method code is very similar to the previous two methods. There are two main differences. The first is adding the search string to the route, and the second is setting the method up to respond to HTTP Post requests instead of HTTP Get requests.

As you saw in Chapter 3, route values can be placed in the constructor of `HttpPost` and `HttpGet` attributes. The caveat is that blending `RouteTable` and attribute routing doesn't work. So a `Route` attribute must be used to define the base route. Add the following code to the `ProductsController`:

```

[Route("[controller]/[action]")]
[HttpPost("{searchString}")]
public async Task<IActionResult> Search(string searchString)
{

```

```

ViewBag.Title = "Search Results";
ViewBag.Header = "Search Results";
ViewBag.ShowCategory = true;
ViewBag.Featured = false;
return await GetListOfProducts(searchString:searchString);
}

```

The {searchString} route value could have also been placed in the Route attribute itself. The HttpPost attribute would then be blank, like this:

```

[Route("[controller]/[action]/{searchString}")]
[HttpPost]

```

Which way to code the route is personal preference. I prefer to have my final route controlled by the HttpGet or HttpPost attribute.

The Orders Controller

Create a new file named `OrdersController.cs` in the `Controllers` directory. Update the using statements to match the following:

```

using System.Collections.Generic;
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using SpyStore.Models.Entities;
using SpyStore.Models.ViewModels;
using SpyStore.MVC.WebServiceAccess.Base;

```

Change the class to public and inherit from `Controller`, as follows:

```

public class OrdersController : Controller

```

The Route

The `OrdersController` route is a little different than the default route. Instead of having an optional integer value named `id`, the `OrdersController` needs an integer value named `customerId` as a required value. Add the following Route attribute to the top of the class, like this:

```

[Route("[controller]/[action]/{customerId}")]
public class OrdersController : Controller
{
    //omitted for brevity
}

```


The Constructor

The `OrdersController` needs an instance of the `IWebApiCalls` class to call into the service to get product data (see the pattern that is forming?). Add the following property to the top of the class and the following constructor code:

```
private readonly IWebApiCalls _webApiCalls;
public OrdersController(IWebApiCalls webApiCalls)
{
    _webApiCalls = webApiCalls;
}
```

The Index Action

The `Index` action lists all of the orders for a particular customer. Add the following code for the `Index` action method:

```
[HttpGet]
public async Task<IActionResult> Index(int customerId)
{
    ViewBag.Title = "Order History";
    ViewBag.Header = "Order History";
    IList<Order> orders = await _webApiCalls.GetOrdersAsync(customerId);
    if (orders == null) return NotFound();
    return View(orders);
}
```

The Details Action

The `Details` action method returns all of the details for a single order. The `orderId` is added to the route using the `HttpGet` attribute. Add the following code to finish off the `OrdersController`:

```
[HttpGet("{orderId}")]
public async Task<IActionResult> Details(int customerId, int orderId)
{
    ViewBag.Title = "Order Details";
    ViewBag.Header = "Order Details";
    OrderWithDetailsAndProductInfo orderDetails =
        await _webApiCalls.GetOrderDetailsAsync(customerId, orderId);
    if (orderDetails == null) return NotFound();
    return View(orderDetails);
}
```

The Shopping Cart Controller

Create a new file named `CartController.cs` in the `Controllers` directory. Update the using statements to match the following:

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;
using AutoMapper;
using Microsoft.AspNetCore.Mvc;
using SpyStore.Models.Entities;
using SpyStore.Models.ViewModels;
using SpyStore.MVC.WebServiceAccess.Base;
using SpyStore.MVC.ViewModels;
using SpyStore.Models.ViewModels.Base;
using Newtonsoft.Json;
```

Change the class to public and inherit from `Controller`, as follows:

```
public class CartController : Controller
```

The Route

The final controller to add is the workhorse of the sample application. In addition to returning one or all of the shopping cart records for a customer, the `Add`, `Update`, and `Delete` methods are implemented as well. The final method executes the stored procedure to execute a purchase. The `CartController` route is the same as the `OrdersController` route. Update the class declaration and route as follows:

```
[Route("[controller]/[action]/{customerId}")]
public class CartController : Controller
{
    //omitted for brevity
}
```

The Constructor

The `ProductsController` needs an instance of the `IWebApiCalls` class to call into the service to get product data. Add the following property and the constructor code:

```
private readonly IWebApiCalls _webApiCalls;
public CartController(IWebApiCalls webApiCalls)
{
    _webApiCalls = webApiCalls;
}
```

Creating the AutoMapper Configuration

AutoMapper (created and maintained by Jimmy Bogard) quickly and easily converts one object to another using reflection. Before types can be converted, the type mappings must be configured. Create a class-level variable to hold the configuration and then update the constructor to the following:

```
readonly MapperConfiguration _config;
public CartController(IWebApiCalls webApiCalls)
{
    _webApiCalls = webApiCalls;
    _config = new MapperConfiguration(cfg =>
    {
        cfg.CreateMap<CartRecordViewModel, ShoppingCartRecord>();
        cfg.CreateMap<CartRecordWithProductInfo, CartRecordViewModel>();
        cfg.CreateMap<ProductAndCategoryBase, AddToCartViewModel>();
    });
}
```

The syntax is simple—the CreateMap command creates a mapping from the first type to the second in the call. For example, this command allows conversion of a CartRecordViewModel to a ShoppingCartRecord:

```
cfg.CreateMap<CartRecordViewModel, ShoppingCartRecord>();
```

The Index Action

The two Index action methods return the list of items in the cart. It needs to get the Cart and Customer information and convert the CartRecordWithProductInfo records into CartRecordViewModel records. The Customer and CartRecordViewModel records are then added to a CartViewModel. It finally returns the Index.cshtml view, using the CartRecordViewModel as the data source for the view. Add the following code:

```
[HttpGet]
public async Task<IActionResult> Index(int customerId)
{
    ViewBag.Title = "Cart";
    ViewBag.Header = "Cart";
    var cartItems = await _webApiCalls.GetCartAsync(customerId);
    var customer = await _webApiCalls.GetCustomerAsync(customerId);
    var mapper = _config.CreateMapper();
    var viewModel = new CartViewModel
    {
        Customer = customer,
        CartRecords = mapper.Map<IList<CartRecordViewModel>>(cartItems)
    };
    return View(viewModel);
}
```

Model Binding

Before moving on to the AddToCart, Update, and Delete methods, model binding needs to be explained further. Model binding was briefly mentioned in Chapter 3. As a refresher, model binding is the process of creating an instance of a specific type from submitted name/value. A new instance of the target type is

created, and then the list of properties on the target type (acquired through reflection) is compared to the list of names in the submitted name value pairs. For every match between a property name and a submitted name, the submitted value is assigned to the property on the new instance. Errors in type conversion are tracked through the `ModelState` object, discussed shortly. Any name/value pairs that don't have a match in the target type are just ignored.

There is implicit model binding and explicit model binding. Implicit model binding happens when a datatype is a parameter for an action method, like this:

```
[ActionName("AddToCart"),HttpPost("{productId}"),ValidateAntiForgeryToken]
public async Task<IActionResult> AddToCartPost(
    int customerId, int productId, AddToCartViewModel item)
{
    //omitted for brevity
}
```

Explicit model binding occurs when the developer asks for it through code and not through a method parameter. The following code shows how to accomplish the same results as the previous code block, but uses explicit model binding:

```
[ActionName("AddToCart"),HttpPost("{productId}"),ValidateAntiForgeryToken]
public async Task<IActionResult> AddToCartPost(int customerId, int productId)
    AddToCartViewModel vm = null;
    var success = await TryUpdateModelAsync(vm);
    //omitted for brevity
}
```

Results of binding attempts are stored in a special object called `ModelState`. This object has an `IsValid` property that is set based on the success or failure of the binding. If all matched properties are successfully assigned, `ModelState.IsValid = true`. If any of the values can't be assigned to their respective properties for any reason, the `ModelState.IsValid` property is set to `false`.

Typical reasons for failed value assignments are datatype conversion issues and/or validation errors. Datatype conversion issues can occur because submitted values come across as strings. For example, if a user enters "Sept 31, 2017" for a date value in the user interface, the model binder will not be able to assign that value to a property of type `DateTime`. The binding engine does not throw an exception, but leaves the value unassigned and sets the model property `IsValid=false`.

In addition to the `IsValid` property, the string indexer for `ModelState` is a `ModelStateDictionary`. This dictionary contains any error information for every property that failed as well as any model level error information. Errors for properties are accessed using the property name. For example, the following code gets error(s) for the failure of a property named `Foo`:

```
var errorReason = ModelState[nameof(Foo)];
```

Error messages can manually be added into the `ModelState` like this:

```
ModelState.AddModelError("Name", "Name is required");
```

If you want to add an error for the entire model, use `string.Empty` for the property name, like this:

```
ModelState.AddModelError(string.Empty, $"Unable to create record: {ex.Message}");
```

Server side error handling for `ModelState` is covered in the "Building the Controllers" section (next) and client side display of any errors is covered in the "Views" section.

The AddToCart Actions

Adds, deletes, and updates are typically conducted using two action methods. The first responds to an HTTP Get request and sets up the view for gathering data from the user. The second responds to an HTTP Post request and executes the data change. If the change is successful, the user is redirected to another HTTP Get action. This prevents the double post problem. If the change fails, the view reloads with the user input and any validation messages.

The Details action (from ProductController) and the AddToCart actions render the same view file. There is a slight difference in what is shown in the UI based on how the user arrived at this method. If it's from one of the Product list pages or the ShoppingCart, then there is a Back link added to the screen. If the user clicked on the Add to Cart link on one of the product list pages, the Back link is not rendered.

The HTTP Get Action

The route for this action needs the Product Id added so the correct Product is displayed in the UI and potentially added to the ShoppingCart. The action also needs a parameter for the cameFromProducts bool, which is not part of the route, but sent in the RedirectToRoute ActionResult from the Details action of the ProductController. Here are the route values from the Details action:

```
new { customerId = ViewBag.CustomerId, productId = id, cameFromProducts = true });
```

The method sets up the ViewBag, then gets an instance of the ProductAndCategoryBase from the service. If a Product isn't found using the ProductId passed into the method, the method returns a NotFoundResult. If it is found, the instance is mapped to an AddToCartViewModel, the Quantity defaults to 1, and the AddToCart.cshtml view is returned with the AddToCartViewModel as the data. Add the following code to the class:

```
[HttpGet("{productId}")]
public async Task<IActionResult> AddToCart(
    int customerId, int productId, bool cameFromProducts = false)
{
    ViewBag.CameFromProducts = cameFromProducts;
    ViewBag.Title = "Add to Cart";
    ViewBag.Header = "Add to Cart";
    ViewBag.ShowCategory = true;
    var prod = await _webApiCalls.GetOneProductAsync(productId);
    if (prod == null) return NotFound();
    var mapper = _config.CreateMapper();
    var cartRecord = mapper.Map<AddToCartViewModel>(prod);
    cartRecord.Quantity = 1;
    return View(cartRecord);
}
```

The HTTP Post Action

When the user submits the data from the AddToCart HTTP Get method, the browser submits an HTTP Post request. The method must have the HttpPost attribute with the {productId} route parameter. MVC views also use antiforgery tokens, so the method needs the ValidateAntiForgeryToken attribute. In addition to the customerId and productId parameters, the method takes an AddToCartViewModel parameter that gets

populated with implicit model binding. If the name of the method doesn't exactly match the expected route, the `ActionName` parameter is added. Add the method signature as follows:

```
[HttpPost("{productId}"),ValidateAntiForgeryToken, ActionName("AddToCart")]
public async Task<IActionResult> AddToCartPost(
    int customerId, int productId, AddToCartViewModel item)
{
}
```

The body of the method first checks the `ModelState`. If it's not valid, then the `AddToCart.cshtml` view is returned, with the current values of the `AddToCartViewModel` (including the `ModelState` errors) as the data. If the `ModelState` is valid, the `AddToCartAsync` method is executed to add the product to the cart. If this is successful, the user is redirected back to the Index view. If there is an error adding the item to the cart, a model-level error is added to the `ModelState` and the `AddToCart.cshtml` view is returned so the user can try again. The full method is listed here:

```
[ActionName("AddToCart"),HttpPost("{productId}"),ValidateAntiForgeryToken]
public async Task<IActionResult> AddToCartPost(
    int customerId, int productId, AddToCartViewModel item)
{
    if (!ModelState.IsValid) return View(item);
    try
    {
        await _webApiCalls.AddToCartAsync(customerId, productId, item.Quantity);
    }
    catch (Exception ex)
    {
        ModelState.AddModelError(string.Empty, "There was an error adding the item to the
cart.");
        return View(item);
    }
    return RedirectToAction(nameof(CartController.Index), new { customerId });
}
```

The Update HTTP Post Action

The Update action only has an HTTP Post implementation. This action method is called as an AJAX request from the Cart List (`Index.cshtml` view), which provides the HTTP Get display of the cart records. The Route needs the ID of the `ShoppingCartRecord` in the `HttpPost` attribute and needs the `ValidateAntiForgeryToken` attribute, as do all MVC HTTP Post methods. In addition to the `customerId` and `id` parameters, the method takes a `timeStampString` parameter that will be passed in as a form value and a `CartRecordViewModel` parameter that will be populated using implicit model binding. Add the following signature:

```
[HttpPost("{id}"),ValidateAntiForgeryToken]
public async Task<IActionResult> Update(int customerId, int id,
    string timeStampString, CartRecordViewModel item)
{
}
```

The method starts by de-serializing the `timeStampString` parameter back into a `byte[]` and assigns it to the `CartItemViewModel` instance. This is done to work around an issue with AJAX calls caching values between calls, and while there are other ways to solve this (such as manually invalidating the cache), this was the quickest and easiest. Then the `ModelState` is checked, and if it's not valid, the `Update.cshtml` partial view is returned as the AJAX results.

If the `ModelState` is valid, the view model is converted back into a `ShoppingCartItem` for update. If the update succeeds, the updated item is mapped back into a `CartItemViewModel`, and the `Update.cshtml` partial view is returned. If the update doesn't succeed, a model level `ModelState` error is added and the original item is returned with the `Update.cshtml` partial view so the user can try again. Update the action method to the following code:

```
[HttpPost("{id}"), ValidateAntiForgeryToken]
public async Task<IActionResult> Update(int customerId, int id,
    string timeStampString, CartItemViewModel item)
{
    item.Timestamp = JsonConvert.DeserializeObject<byte[]>($"\"{timeStampString}\"");
    if (!ModelState.IsValid) return PartialView(item);
    var mapper = _config.CreateMapper();
    var newItem = mapper.Map<ShoppingCartItem>(item);
    try
    {
        await _webApiCalls.UpdateCartItemAsync(newItem);
        var updatedItem = await _webApiCalls.GetCartItemAsync(customerId, item.ProductId);
        var newViewModel = mapper.Map<CartItemViewModel>(updatedItem);
        return PartialView(newViewModel);
    }
    catch (Exception ex)
    {
        ModelState.AddModelError(string.Empty,
            "An error occurred updating the cart. Please reload the page and try again.");
        return PartialView(item);
    }
}
```

The Delete HTTP Post Action

Similar to the Update action, the Delete action only needs an HTTP Post implementation. Unlike the Update action, this action isn't called using AJAX, so at the end of the method, the user is redirected to the Index action. Add the following code:

```
[HttpPost("{id}"), ValidateAntiForgeryToken]
public async Task<IActionResult> Delete(int customerId, int id, ShoppingCartRecord item)
{
    await _webApiCalls.RemoveCartItemAsync(customerId, id, item.Timestamp);
    return RedirectToAction(nameof(Index), new { customerId });
}
```

■ **Note** This method could be improved using AJAX as well, but was kept simple to show redirecting the user back to the Index view.

The Buy HTTP Post Action

This method uses (now) familiar techniques to call the `PurchaseCartAsync` method. A production version should add error handling as you have seen in the earlier code samples. Add the following code:

```
[HttpPost, ValidateAntiForgeryToken]
public async Task<IActionResult> Buy(int customerId, Customer customer)
{
    int orderId = await _webApiCalls.PurchaseCartAsync(customer);
    return RedirectToAction(
        nameof(OrdersController.Details),
        nameof(OrdersController).Replace("Controller", ""),
        new { customerId, orderId });
}
```

■ **Note** The examples in this section demonstrate many, but certainly not all, of the features available when developing Core MVC controllers. For more information, refer to the official documentation here: <https://docs.microsoft.com/en-us/aspnet/core/mvc/controllers/actions>.

Validation

Validation in Core MVC applications should be conducted client side and server side. Client side validation is conducted using JavaScript and is immediately responsive to the user. However, if users have JavaScript turned off in their browsers, then the validation doesn't execute, and potentially harmful data (or scripting attacks) can be sent to the server.

Many of the built-in data annotations already used in the data access layer, such as `MaxLength` and `Required`, also participate in validations (both server side and client side). For example, the following `FullName` property is limited to 50 characters in length:

```
[DataType(DataType.Text), MaxLength(50), Display(Name = "Full Name")]
public string FullName { get; set; }
```

If a value longer than 50 characters is submitted, the client side validation will prevent the value from being sent to the server and will display the error message in the Validation Tag Helpers. Client side validation is covered in detail shortly.

Server Side Validation

If client side validation is disabled for any reason, when the data is sent to the server, the model binding engine will fail the `FullName` property, set `ModelState.IsValid = false`, and add the following error to the `ModelState` instance:

The field Full Name must be a string or array type with a maximum length of '50'.

The value `Full Name` comes from the `Display` attribute. If this attribute isn't supplied, then the field name (`FullName` in this example) is used in the message. The errors collection for a field (or the model) in

the `ModelStateDictionary` provides access to the specific `ErrorMessage` and `Exception` (if one exists). For example, the following code gets the error message of the first error for the `FullName` property:

```
var message = ModelState[nameof(Customer.FullName)].Errors[0].ErrorMessage
```

Most attributes can be assigned custom messages in their constructor. Custom messages override the automatic message that gets created by the built-in attributes. To add a custom message for the `MaxLength` validation attribute on the `FullName` property, use the following:

```
[DataType(DataType.Text), MaxLength(50,ErrorMessage="Please make it shorter"), DisplayName = "Full Name"]
```

Custom Server Side Validation

Custom validation attributes can be created by inheriting from `ValidationAttribute`. The `SpyStore.MVC` application uses two custom validation attributes, one to verify if a number is greater than zero, and another to verify that a property is not greater than another property.

Greater Than Zero Attribute

Create a new directory named `Validations` in the `SpyStore.MVC` project. Add a new class named `MustBeGreaterThanZeroAttribute`, add the necessary `using` statements, make the class public, and inherit from `ValidationAttribute`, as shown here:

```
using System.ComponentModel.DataAnnotations;
using Microsoft.AspNetCore.Mvc.ModelBinding.Validation;
public class MustBeGreaterThanZeroAttribute : ValidationAttribute
{
}
```

The base class takes an error message in its constructor. Create two constructors for the class. The first has no parameters and is used to create a default error message for the validation and pass it to the second constructor. The second accepts an error message from the attribute (or the default constructor) and passes it to the base. The constructors are as follows:

```
public MustBeGreaterThanZeroAttribute(): this("{0} must be greater than 0") { }
public MustBeGreaterThanZeroAttribute(string errorMessage): base(errorMessage) { }
```

Override the `FormatErrorMessage` method to format the error message at runtime. This is typically used to add the property name to the message, which is the single parameter of the method. If a `DisplayNameAttribute` is present on the property, that value is passed in as the parameter. Otherwise, the actual property name is used. Override the method and implement it as follows:

```
public override string FormatErrorMessage(string name)
{
    return string.Format(ErrorMessageString, name);
}
```

The final method to override is the `IsValid` method. This method takes the submitted string value and an instance of the `ValidationContext`. The validation context provides access to the property being validated (including its `DisplayName`) as well as the rest of the class instance it belongs to.

This method checks if the value passed in can be converted to an integer and if so, validates that the converted value is greater than zero. If both of these conditions are true, return `ValidationResult.Success`. Otherwise, use the `ValidationResult` helper method to return an instance of the `ValidationResult` class with the formatted error message. When anything other than `ValidationResult.Success` is returned, the `ModelState.IsValid` property is set to `false`, and an entry into the `ModelStateDictionary` for the property is added. The code is listed here:

```
protected override ValidationResult IsValid(object value, ValidationContext
validationContext)
{
    if (!int.TryParse(value.ToString(), out int result))
    {
        return new ValidationResult(FormatErrorMessage(validationContext.DisplayName));
    }
    if (result > 0)
    {
        return ValidationResult.Success;
    }
    return new ValidationResult(FormatErrorMessage(validationContext.DisplayName));
}
```

Must Not Be Greater Than Attribute

Add a new class named `MustNotBeGreaterThanAttribute` into the `Validations` directory. Add the following using statements:

```
using System;
using System.ComponentModel.DataAnnotations;
using System.Linq;
using System.Reflection;
using Microsoft.AspNetCore.Mvc.ModelBinding.Validation;
```

This attribute checks to see if the target property is less than or equal to another property on the class. This check might be necessary against more than one other property, so the attribute must be set up to allow multiple uses on the same property. That is accomplished with the `AttributeUsage`. The class also needs to inherit `ValidationAttribute`. Update the class definition as follows:

```
[AttributeUsage(AttributeTargets.Property, AllowMultiple = true)]
public class MustNotBeGreaterThanAttribute : ValidationAttribute
{
}
```

In addition to the error message, the constructors need the name of the property to be compared against. The attribute can also work in situations where there is a prefix to the property name, such as when displayed in a list. This will need to be passed in as well, defaulting to an empty string. Add the following two constructors and class-level variables (to hold the other property's name and display name), as follows:

```
readonly string _otherPropertyName;
string _otherPropertyDisplayName;
readonly string _prefix;
public MustNotBeGreaterThanAttribute(string otherPropertyName, string prefix = "")
```

```

: this(otherPropertyName, "{0} must not be greater than {1}",
  prefix)
{
}
public MustNotBeGreaterThanAttribute(
  string otherPropertyName, string errorMessage, string prefix) : base(errorMessage)
{
  _otherPropertyName = otherPropertyName;
  _otherPropertyDisplayName = otherPropertyName;
  _prefix = prefix;
}

```

Next, override the `FormatErrorMessage`, as follows:

```

public override string FormatErrorMessage(string name)
{
  return string.Format(ErrorMessageString, name, _otherPropertyDisplayName);
}

```

Add a new internal method named `SetOtherPropertyInfo` that will get the display name of the other property based on the `PropertyInfo`. The code is shown here:

```

internal void SetOtherPropertyInfo(PropertyInfo otherPropertyInfo)
{
  var displayAttribute = otherPropertyInfo
    .GetCustomAttributes<DisplayAttribute>().FirstOrDefault();
  _otherPropertyDisplayName = displayAttribute?.Name ?? _otherPropertyName;
}

```

The `IsValid` method uses the `ValidationContext` to get the `PropertyInfo` and value about the other property. The method attempts to convert the input value to an integer and compares it with the target value (which is assumed to be an integer). If the property's value is less than or equal to the other property's value, the method returns `ValidationResult.Success`.

```

protected override ValidationResult IsValid(object value, ValidationContext
validationContext)
{
  var otherPropertyInfo = validationContext.ObjectType.GetProperty(_otherPropertyName);
  SetOtherPropertyInfo(otherPropertyInfo);
  if (!int.TryParse(value.ToString(), out int toValidate))
  {
    return new ValidationResult($"{validationContext.DisplayName} must be numeric.");
  }
  var otherValue = (int)otherPropertyInfo.GetValue(validationContext.ObjectInstance, null);
  return toValidate > otherValue
    ? new ValidationResult(FormatErrorMessage(validationContext.DisplayName))
    : ValidationResult.Success;
}

```

Client Side Validation

Server side validation is only half of the story. Client side validation needs to be implemented in addition to server side validation to prevent wasted server calls and a degraded user experience. Client side validation is done through a combination of JavaScript and HTML5 attributes. By default, Core MVC uses the `jQuery.Validate` and `jQuery.Validate.Unobtrusive` libraries as the core validation frameworks.

The validation frameworks use the `data-val` HTML5 attributes to indicate that validation is requested on the element. For example, the `Required` attribute adds the following attributes to an element:

```
data-val-required="The Quantity field is required." data-val="true"
```

The `data-val="true"` attribute enables client side validation. This is only added once, even if there are multiple validation attributes assigned. The `data-val-required="The Quantity field is required."` attribute indicates that the JavaScript function used for validation is named `required` and the error message is set to `"The Quantity field is required."`

For the built-in validation attributes, client side validation is already configured. All that needs to be done is to add the proper JavaScript libraries to the page.

Custom Client Side Validation

For custom validations to work client side in addition to server side, the validators must implement `IClientModelValidator` and the `AddValidation` method. Additionally, custom JavaScript must be written and added to the view.

Greater Than Zero Attribute: Attribute Updates

Open the `MustBeGreaterThanZeroAttribute.cs` class and add the `IClientModelValidator` interface and `AddValidation` method, as shown here:

```
public class MustBeGreaterThanZeroAttribute : ValidationAttribute, IClientModelValidator
{
    //omitted for brevity
    public void AddValidation(ClientModelValidationContext context)
    {
    }
}
```

The `AddValidation` method is used to add the required HTML5 attributes to the target element. For this validation, the attribute added will be `data-val-greaterthanzero` and the formatted error message.

The method first uses the `ClientModelValidationContext` to get the display name (if it exists, or the actual property name if it doesn't) for the property being validated. This is used to build the formatted error message and then the formatted HTML5 attribute is added to the element. Update the `AddValidation` method to the following:

```
public void AddValidation(ClientModelValidationContext context)
{
    string propertyDisplayName =
        context.ModelMetadata.DisplayName ?? context.ModelMetadata.PropertyName;
    string errorMessage = FormatErrorMessage(propertyDisplayName);
    context.Attributes.Add("data-val-greaterthanzero", errorMessage);
}
```

Greater Than Zero Attribute: JavaScript

Open the `validators.js` file under `wwwroot\js\validations` and enter the following code:

```
$.validator.addMethod("greaterthanzero", function (value, element, params) {
    return value > 0;
});

$.validator.unobtrusive.adapters.add("greaterthanzero", function (options) {
    options.rules["greaterthanzero"] = true;
    options.messages["greaterthanzero"] = options.message;
});
```

The first code block defines the `greaterthanzero` method, which simply compares the values and returns a boolean. The second code block registers the validation method with the jQuery validation framework. The `options.rules` value gets passed into the validation function as one of the `params`. The `options.rules["greaterthanzero"]` must be set for the adapter to be enabled, even if there aren't any parameters to pass into the function. Finally, the error message is registered with the validation framework. Even though this was set in the `AddValidation` method, it must also be done here.

Must Not Be Greater Than Attribute: Attribute Updates

Open the `MustNotBeGreaterThanAttribute.cs` class and add the `IClientModelValidator` interface and the `AddValidation` method, as shown here:

```
public class MustBeGreaterThanZeroAttribute : ValidationAttribute, IClientModelValidator
{
    //omitted for brevity
    public void AddValidation(ClientModelValidationContext context)
    {
    }
}
```

The `AddValidation` method uses the `ClientModelValidationContext` to get the display name of both the property that is the target of the validation and the property it is being compared to, then adds the HTML5 attributes to the target property. Update the `AddValidation` method to the following:

```
public void AddValidation(ClientModelValidationContext context)
{
    string propertyDisplayName = context.ModelMetadata.GetDisplayName();
    var propertyInfo = context.ModelMetadata.ContainerType.GetProperty(_otherPropertyName);
    SetOtherPropertyName(propertyInfo);
    string errorMessage = FormatErrorMessage(propertyDisplayName);
    context.Attributes.Add("data-val-notgreaterthan", errorMessage);
    context.Attributes.Add("data-val-notgreaterthan-otherpropertyname", _otherPropertyName);
    context.Attributes.Add("data-val-notgreaterthan-prefix", _prefix);
}
```

Must Not Be Greater Than Attribute: JavaScript

Open the `validators.js` file under `wwwroot\js\validations` and enter the following code:

```
$.validator.addMethod("notgreaterthan", function (value, element, params) {
    return +value <= +$(params).val();
});
$.validator.unobtrusive.adapters.add("notgreaterthan", ["otherpropertyname", "prefix"],
function (options) {
    options.rules["notgreaterthan"] =
        "#" + options.params.prefix + options.params.otherpropertyname;
    options.messages["notgreaterthan"] = options.message;
});
```

This is very similar to the previous validator. The first code block defines the `notgreaterthan` method. The second code block registers the validation method with the validation framework. The `options.rules` value combines the prefix to the other property name, which will be passed into the `notgreaterthan` function as a parameter.

Specifying Formatting for Client Side Validation Errors

The final item for client side validation is to change the display default when an error occurs. Open the `errorFormatting.js` file and enter the following code:

```
$.validator.setDefaults({
    highlight: function (element, errorClass, validClass) {
        if (element.type === "radio") {
            this.findByName(element.name).addClass(errorClass).removeClass(validClass);
        } else {
            $(element).addClass(errorClass).removeClass(validClass);
            $(element).closest('.form-group').addClass('has-error');
        }
    },
    unhighlight: function (element, errorClass, validClass) {
        if (element.type === "radio") {
            this.findByName(element.name).removeClass(errorClass).addClass(validClass);
        } else {
            $(element).removeClass(errorClass).addClass(validClass);
            $(element).closest('.form-group').removeClass('has-error');
        }
    }
});
```

The preceding code ties into the `highlight` and `unhighlight` events of the validation framework to set specific styles that are defined in the `spystore-bootstrap.css` style sheet.

Updating the View Models

The final step in the validation chain is to update the `AddToCartViewModel` and the `CartRecordViewModel`. Neither should allow the user to update the `Quantity` to anything greater than the available `Inventory`, and additionally, the `AddToCartViewModel` should now allow the quantity of zero.

Open the `AddToCartViewModel` class, add a `using` for the `SpyStore.MVC.Validation` namespace, and update the `Quantity` property to the following:

```
using SpyStore.MVC.Validation;
public class AddToCartViewModel : CartViewModelBase
{
    [MustNotBeGreaterThan(nameof(UnitsInStock)), MustBeGreaterThanZero]
    public int Quantity { get; set; }
}
```

Next, open the `CartRecordViewModel` class, add a `using` for the `SpyStore.MVC.Validation` namespace, and update the `Quantity` to the following:

```
using SpyStore.MVC.Validation;
public class CartRecordViewModel : CartViewModelBase
{
    [MustNotBeGreaterThan(nameof(UnitsInStock))]
    public int Quantity { get; set; }
}
```

View Components

View Components are another new feature in Core MVC. They combine the benefits of partial views with child actions to render parts of the UI. Like partial views, they are called from another view, but unlike partial views by themselves, View Components also have a server side component. This combination makes them a great fit for functions like creating dynamic menus (as shown shortly), login panels, sidebar content, or anything that needs to run server side code but doesn't qualify to be a standalone view.

Building the Server Side Code

The main page of the site contains a menu across the top that is generated from the Category names in the database. Based on the size of the view port, the menu is either laid out horizontally across the top or hidden behind the responsive “cheeseburger” icon provided by the Bootstrap responsive menu system, as shown in Figure 5-1.

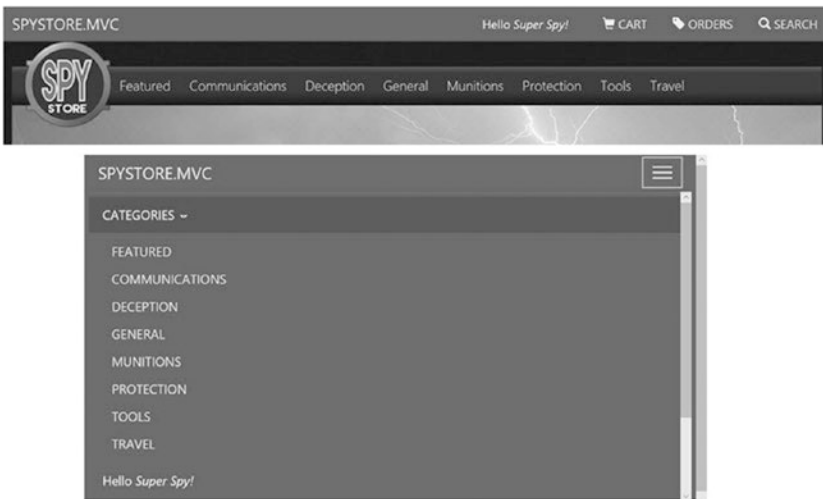


Figure 5-1. The standard menu layout and the responsive version

The same code in more than one place should be encapsulated for reuse, but a partial view won't work here because of the server side code that needs to be executed. This is a perfect user case for View Components.

Create a new folder named `ViewComponents` in the root directory of the `SpyStore.MVC` project. Add a new class file named `Menu.cs` into this folder. Convention dictates that the View Components should be named like controller, i.e. `MenuViewComponent`, instead of just `Menu`. However, I like to keep my View Component names cleaner. I feel this adds readability when they are called from the view. This will be demonstrated shortly. Add the following `using` statements to the top of the file:

```
using System.Threading.Tasks;
using Microsoft.AspNetCore.Mvc;
using Microsoft.AspNetCore.Mvc.ViewComponents;
using SpyStore.MVC.WebServiceAccess.Base;
```

Change the class to `public` and inherit from `ViewComponent`. View Components don't have to inherit from the `ViewComponent` base class, but like the `Controller` base class, there are methods in the base class that are very useful. Update the class to this:

```
public class Menu : ViewComponent
{
}
```

This View Component needs to call the `GetCategoriesAsync` method on the `WebApiCalls` class, and fortunately, View Components can leverage the DI container built into Core MVC. Create a class-level variable to hold an instance of the `IWebApiCalls` interface and create a constructor that receives an instance of the interface, as shown here:

```
public class Menu : ViewComponent
{
    private readonly IWebApiCalls _webApiCalls;
    public Menu(IWebApiCalls webApiCalls)
```



```

{
    _webApiCalls = webApiCalls;
}
}

```

When a View Component is rendered from a View, the public method `InvokeAsync` is invoked. This method returns an `IViewComponentResult`, which is conceptually similar to a `PartialViewResult`, but much more streamlined. In the `InvokeAsync` methods, get the list of `Categories` from the service, and if successful, return a `ViewComponentResult` using the returned list as the view model. If the call to get the `Categories` fails, return a `ContentViewComponentResult` with an error message. Add the following code to the `Menu` class.

```

public async Task<IViewComponentResult> InvokeAsync()
{
    var cats = await _webApiCalls.GetCategoriesAsync();
    if (cats == null)
    {
        return new ContentViewComponentResult("There was an error getting the categories");
    }
    return View("MenuView", cats);
}

```

The View helper method from the base `ViewComponent` class works just like the base `Controller` class helper method of the same name, except for a couple of key differences. The first difference is that the default view file name is `Default.cshtml` instead of the name of the Action method. The second difference is that the location of the view **must** be one of these two directories:

```

Views/<parent_controller_name_calling_view >/Components/<view_component_name>/<view_name>
Views/Shared/Components/<view_component_name>/<view_name>

```

The C# class can live anywhere (even in another assembly), but the `<viewname>.cshtml` must be in one of the directories listed above.

The complete class is listed here:

```

public class Menu : ViewComponent
{
    private readonly IWebApiCalls _webApiCalls;
    public Menu(IWebApiCalls webApiCalls)
    {
        _webApiCalls = webApiCalls;
    }
    public async Task<IViewComponentResult> InvokeAsync()
    {
        var cats = await _webApiCalls.GetCategoriesAsync();
        if (cats == null)
        {
            return new ContentViewComponentResult("There was an error getting the categories");
        }
        return View("MenuView", cats);
    }
}

```

Building the Client Side Code

The View rendered by the Menu View Component will iterate through the Category records, adding each as a list item to be displayed in the Bootstrap menu. The Featured menu item is added first as a hard-coded value.

Create a new folder named Components under the Views\Shared folder. In this new folder, create another new folder named Menu. This folder name must match the name of the View Component class created above. In this folder, create a partial view named MenuView.cshtml by selecting Add ► New Item and selecting the MVC View Page from the Add New Item dialog.

Clear out the existing code and add the following markup:

```
@model IEnumerable<Category>
<li><a asp-controller="Products" asp-action="Featured">Featured</a></li>
@foreach (var item in Model)
{
    <li><a asp-controller="Products" asp-action="ProductList" asp-route-id="@item.Id">
        @item.CategoryName</a>
    </li>
}
```

Invoking View Components

View Components are typically rendered from a view (although they can be rendered from a Controller Action method as well). The syntax is very straightforward: `Component.Invoke(<string name of the view component>)`. Just like with controllers, the `ViewComponent` suffix must be removed when invoking a View Component. This is why I don't use the suffix for my View Components. Leaving it off enables me to write the following code (without using the `Replace` string function as demonstrated in the Controllers section):

```
@await Component.InvokeAsync(nameof(Menu))
```

Invoking View Components as Custom Tag Helpers

New in ASP.NET 1.1, View Components can be called as Tag Helpers. Instead of using `Component.InvokeAsync`, simply call the View Component like this:

```
<vc:menu></vc:menu>
```

In order to use this method of calling View Components, they must be added as Tag Helpers using the `@addTagHelper` command. This was already added to the `_ViewImports.cshtml` file earlier in this chapter to enable the custom Email Tag Helper. As a refresher, here is the line that was added:

```
@addTagHelper *, SpyStore.Mvc
```

Updating and Adding the Views

It's finally time to add (or update) the views for the application. To save you typing (or copying and pasting), all of the views are located in the Views folder in the downloadable code.

■ **Note** As mentioned earlier, the two chapters on Core MVC Web Applications are laid out in a serial, bottom-up manner. I am not suggesting that is how you should develop Core MVC applications, it just turns out to be the clearest way for me to teach the concepts instead of jumping from topic to topic.

The ViewImports File

As a refresher, the `_ViewImports.cshtml` file executes before any view (in the directory structure below the location of the `_ViewStart.cshtml` file) is rendered. The file has already been updated to add the custom Tag Helper and View Component to the file. The final step is to add the common using statements into the file so they don't have to be added into each view. Update the file to the following:

```
@using SpyStore.MVC
@using SpyStore.MVC.WebServiceAccess
@using SpyStore.MVC.WebServiceAccess.Base
@using SpyStore.MVC.ViewModels
@using SpyStore.Models.Entities
@using SpyStore.Models.ViewModels
@using SpyStore.Models.ViewModels.Base
@addTagHelper *, Microsoft.AspNetCore.Mvc.TagHelpers
@addTagHelper *, SpyStore.MVC
```

The Shared Views

The Shared folder contains the views that are accessible to all of the controllers and actions in the application. The `Error.cshtml` that was supplied by the VS2017 template is sufficient for this application, and doesn't need to be updated.

The Login Partial View

The Login Partial View encapsulates the top right of the menu bar and is shown in Figure 5-2 (with the Search box expanded).

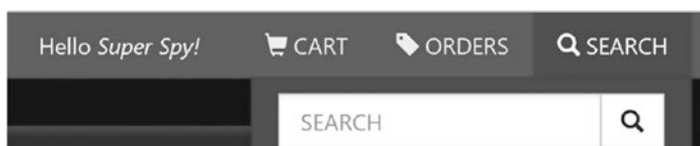


Figure 5-2. The Login View section of the base layout page

Create a new MVC View Page named `LoginView.cshtml` in the `Shared` directory. Clear out the existing content and add the following markup:

```
<li class="navbar-text">Hello <em>@ViewBag.CustomerName!</em></li>
<li><a asp-controller="Cart" asp-action="Index" asp-route-customerId="@ViewBag.CustomerId"
title="Shopping Cart"><span class="glyphicon glyphicon-shopping-cart"></span> Cart</a></li>
<li><a asp-controller="Orders" asp-action="Index" asp-route-customerId="@ViewBag.CustomerId"
title="Order History"><span class="glyphicon glyphicon-tag"></span> Orders</a></li>
<li class="dropdown">
  <a href="#" class="dropdown-toggle" data-toggle="dropdown"><span class="glyphicon
glyphicon-search"></span> SEARCH</a>
  <div class="dropdown-menu search-dropdown-menu">
    <form asp-controller="Products" asp-action="Search" class="navbar-form navbar-left"
role="search">
      <div class="input-group">
        <label class="sr-only" for="searchString">Search</label>
        <input type="text" id="searchString" name="searchString" class="form-control"
placeholder="SEARCH">
        <span class="input-group-btn">
          <button class="btn btn-default" type="submit"><span class="glyphicon glyphicon-
search"></span></button>
        </span>
      </div>
    </form>
  </div>
</li>
```

The view uses Anchor Tag Helpers at the top of the file to create links to the `Index` Action method of the `CartController` and `OrdersController`. The view then uses the Form Tag Helper to build the Search Form and standard HTML controls within the form.

The Layout View

The `_Layout.cshtml` view is the base layout for all of the views in this application. The title of the page is set from the `ViewData`, which is supplied by the contained view. The head uses the Environment Tag Helper to select which version of the CSS files to load, minified or un-minified. The CSS files are loaded using the Link Tag Helper to work around caching issues:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <style>
    body {
      padding-bottom: 60px;
    }
  </style>
```

```

<title>@ViewData["Title"] - SpyStore.MVC</title>
<environment names="Development">
  <link rel="stylesheet" href="~/css/site.css" asp-append-version="true"/>
  <link rel="stylesheet" href="~/css/spystore-bootstrap.css" asp-append-version="true"/>
</environment>
<environment names="Staging,Production">
  <link rel="stylesheet" href="~/css/site.min.css" />
  <link rel="stylesheet" href="~/css/spystore-bootstrap.min.css" asp-append-version="true"/>
</environment>
</head>

```

The body starts with the header, which builds the Bootstrap navigation system. The View Component is used inside the Bootstrap navbar, loaded as a Tag Helper (with the traditional invoke call commented out). This part of the navbar creates the “cheeseburger” in smaller view ports. The header also loads the `LoginView.cshtml` as a partial view.

```

<body>
  <header class="navbar navbar-default navbar-fixed-top">
    <div class="container">
      <div class="navbar-header">
        <button type="button" class="navbar-toggle" data-toggle="collapse" data-target=".
        navbar-collapse">
          <span class="sr-only">Toggle navigation</span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
          <span class="icon-bar"></span>
        </button>
        <a asp-area="" asp-controller="Products" asp-action="Index" class="navbar-
        brand">SpyStore.MVC</a>
      </div>
      <nav class="navbar-collapse collapse header-collapse">
        <ul class="nav navbar-nav navbar-right">
          <li class="dropdown">
            <a href="#" class="dropdown-toggle hidden-md hidden-lg" data-
            toggle="dropdown">CATEGORIES <span class="caret"></span></a>
            <ul class="dropdown-menu">
              <li>
                <code>@*await Component.InvokeAsync(nameof(Menu))*@
              </li>
            </ul>
          </li>
          <li>
            <code>@Html.Partial("LoginView")
          </li>
        </ul>
      </nav>
    </div>
  </header>

```

The next section of the layout creates the page container that all views will be rendered into using Bootstrap styles to manage the layout. The section uses an Anchor Tag Helper around the site logo and the Menu View Component to render the standard menu bar. The `RenderBody` Razor method loads the requested view into the layout at that location.

```

<div class="page container">
  <div class="panel panel-default">
    <nav class="panel-heading hidden-xs">
      <div class="store-logo">
        <a asp-controller="Products" asp-action="Index">
          </a>
        </div>
      <ul class="nav nav-pills hidden-sm">
        <vc:menu></vc:menu>
      </ul>
    </nav>
    <div class="panel-body">
      @RenderBody()
    </div>
  </div>
</div>

```

The footer section uses the Email Tag Helper to render an e-mail link:

```

<footer>
  <hr />
  <div class="container">
    <small class="text-muted">
      &copy; 2016 - SpyStore.MVC PBK Productions
      <email email-name="blog" email-domain="skimedica.com"></email>
    </small>
  </div>
</footer>

```

After the footer, the Environment Tag Helper is used to load the site's JavaScript files. If the site is running under Development mode, the raw files are loaded. If the site is running under Staging or Product, the minified and bundled files are loaded for local content, and the Microsoft CDN is used for framework files.

```

<environment names="Development">
  <script src="~/lib/jquery/dist/jquery.js" asp-append-version="true"></script>
  <script src="~/lib/bootstrap/dist/js/bootstrap.js" asp-append-version="true"></script>
  <script src="~/js/site.js" asp-append-version="true"></script>
</environment>
<environment names="Staging,Production">
  <script src="https://ajax.aspnetcdn.com/ajax/jquery/jquery-3.1.1.min.js"
    asp-fallback-src="~/lib/jquery/dist/jquery.min.js"
    asp-fallback-test="window.jQuery">
  </script>
  <script src="https://ajax.aspnetcdn.com/ajax/bootstrap/3.3.6/bootstrap.min.js"
    asp-fallback-src="~/lib/bootstrap/dist/js/bootstrap.min.js"
    asp-fallback-test="window.jQuery && window.jQuery.fn && window.jQuery.fn.modal">
  </script>
  <script src="~/js/site.min.js" asp-append-version="true"></script>
</environment>

```

The final code created the optional scripts section (after all of the framework files have been loaded) and closes out the body and html tags:

```
@RenderSection("scripts", required: false)
</body>
</html>
```

The Validation Scripts Partial View

A common mechanism to load related JavaScript files is to place them into a view to be loaded as a partial view. Create a new MVC View Page named `_ValidationScriptsPartial.cshtml` in the Shared directory. Clear out the existing code and replace it with this use of the Environment Tag Helper:

```
<environment names="Development">
  <script src="~/lib/jquery-validation/dist/jquery.validate.js"></script>
  <script src="~/lib/jquery-validation-unobtrusive/jquery.validate.unobtrusive.js"></script>
  <script src="~/lib/jquery-ajax-unobtrusive/jquery.unobtrusive-ajax.js"></script>
  <script src="~/js/validations/validators.js" asp-append-version="true"></script>
  <script src="~/js/validations/errorFormatting.js" asp-append-version="true"></script>
</environment>
<environment names="Staging,Production">
  <script src="https://ajax.aspnetcdn.com/ajax/jquery.validate/1.16.0/jquery.validate.min.js"
    asp-fallback-src="~/lib/jquery-validation/dist/jquery.validate.min.js"
    asp-fallback-test="window.jQuery && window.jQuery.validator">
  </script>
  <script src="https://ajax.aspnetcdn.com/ajax/mvc/5.2.3/jquery.validate.unobtrusive.min.js"
    asp-fallback-src="~/lib/jquery-validation-unobtrusive/jquery.validate.unobtrusive.min.js"
    asp-fallback-test="window.jQuery && window.jQuery.validator && window.jQuery.validator.
    unobtrusive">
  </script>
  <script src="~/lib/jquery-ajax-unobtrusive/jquery.unobtrusive-ajax.min.js"></script>
  <script src="~/js/validations/validations.min.js"></script>
</environment>
```

Now, any view that needs to add the validation scripts only needs to add the following into the markup:

```
@section Scripts {
  @{
    await Html.RenderPartialAsync("_ValidationScriptsPartial");
  }
}
```

The Add to Cart View

The Add to Cart view doubles as the Product Details view, as discussed in the “Building the Controllers” section. The view is shown in Figure 5-3.



Figure 5-3. The Add to Cart view

To create the view, add a new MVC Razor Page named `AddToCart.cshtml`. Clear out the templated code and add the following markup:

```
@model AddToCartViewModel
@{
    ViewData["Title"] = @ViewBag.Title;
}
<h3>@ViewBag.Header</h3>
<form method="post" asp-controller="Cart" asp-action="AddToCart"
    asp-route-customerId="@ViewBag.CustomerId" asp-route-productId="@Model.Id">
    @Html.EditorForModel()
</form>
@{
    if (ViewBag.CameFromProducts != null && ViewBag.CameFromProducts)
    {
        <div>
            <a onclick="window.history.go(-1); return false;">Back to List</a>
        </div>
    }
}
@section Scripts {
    @{
        await Html.RenderPartialAsync("_ValidationScriptsPartial");
    }
}
```

The code starts by strongly typing the view to the `AddToCartViewModel` class. After setting the `Title` and adding a header for the page, the view uses the Form Tag Helper to build the input form used for adding

the Product into the shopping cart. Inside the form, the `Html.EditorForModel` Razor Helper renders the `AddToCartViewModel.cshtml` template (shown shortly). This is a shortcut for the following line:

```
@Html.EditorFor(model=>model)
```

If the user came from the Product or the Cart page, the `ViewBag.CameFromProducts` value is true, and the Back to List link is provided. Otherwise, it is hidden. The final code block adds in the validation scripts.

The Display Templates

The `Boolean.cshtml` display template was added earlier. The sight uses one more shared display template for displaying properly formatted dates.

The DateTime Display Template

Create a new file named `DateTime.cshtml` in the `Views\Shared\DisplayTemplates` folder. Clear out any code in the template and add the following markup:

```
@using System.Threading.Tasks
@model DateTime?
@if (Model == null)
{
    @:Unknown
}
else
{
    if (ViewData.ModelMetadata.IsNullableValueType)
    {
        @:@(Model.Value.ToString("d"))
    }
    else
    {
        @:@(((DateTime)Model).ToString("d"))
    }
}
}
```

The template displays “Unknown” if the property is nullable and does not have a value. Otherwise, it makes sure the value is formatted as a short date.

The Editor Templates

In addition to the Boolean editor template, the site has one more shared editor template, the `AddToCartViewModel` template. This template is used by the Add to Cart view, as shown previously in Figure 5-3.

The AddToCartViewModel Editor Template

Add a new file named `AddToCartViewModel.cshtml` to the `Views\Shared\EditorTemplates` folder. Clear out the templated code and add the following markup:

```
@model AddToCartViewModel
@if (Model.Quantity == 0)
{
    Model.Quantity = 1;
}
<h1 class="visible-xs">@Html.DisplayFor(x => x.ModelName)</h1>
<div class="row product-details-container">
    <div class="col-sm-6 product-images">
        
        <div class="key-label">PRODUCT IMAGES</div>
    </div>
    <div class="col-sm-6">
        <h1 class="hidden-xs">@Html.DisplayFor(x => x.ModelName)</h1>
        <div class="price-label">Price:</div><div class="price">
            @Html.DisplayFor(x => x.CurrentPrice)</div>
        <div class="units">Only @Html.DisplayFor(x => x.UnitsInStock) left.</div>
        <div class="units"></div>
        <div class="product-description">@Html.DisplayFor(x => x.Description)</div>
        <ul class="product-details">
            <li><div class="key-label">MODEL NUMBER:</div> @Html.DisplayFor(x => x.ModelNumber)</li>
            <li>
                <div class="key-label">CATEGORY:</div>
                <a asp-controller="Products" asp-action="ProductList"
                    asp-route-id="@Model.CategoryId"
                    class="Category">@Model.CategoryName</a>
            </li>
        </ul>
        <input type="hidden" asp-for="Id" />
        <input type="hidden" asp-for="CustomerId" />
        <input type="hidden" asp-for="CategoryName" />
        <input type="hidden" asp-for="Description" />
        <input type="hidden" asp-for="UnitsInStock" />
        <input type="hidden" asp-for="UnitCost" />
        <input type="hidden" asp-for="ModelName" />
        <input type="hidden" asp-for="ModelNumber" />
        <input type="hidden" asp-for="TimeStamp" />
        <input type="hidden" asp-for="ProductId" />
        <input type="hidden" asp-for="LineItemTotal" />
        <input type="hidden" asp-for="CurrentPrice" />
        <input type="hidden" asp-for="ProductImage" />
        <input type="hidden" asp-for="ProductImageLarge" />
        <input type="hidden" asp-for="ProductImageThumb" />
        <div class="row cart-group">
            <label>QUANTITY:</label>
            <input asp-for="Quantity" class="cart-quantity form-control" />
            <input type="submit" value="Add to Cart" class="btn btn-primary" />
        </div>
    </div>
</div>
```

```



    <div asp-validation-summary="ModelOnly" class="text-danger"></div>
    <span asp-validation-for="Quantity" class="text-danger"></span>
  </div>
</div>

```

The Cart Views

Create a new folder named `Cart` under the `Views` folder. This folder will hold the views for the `CartController` Action methods. The view is shown in Figure 5-4.

Cart

Product	Price	Quantity	Available	Total
 Escape Vehicle (Air) Travel In a jam, need a quick escape? Just whip out a sheet of our patented P38 paper and, with a few quick folds, it converts into a lighter-than-air escape vehicle! Especially effective on windy days - no fuel required. Comes in several sizes including letter, legal, A10, and B52.	\$2.99	<input type="text" value="1"/> Update Remove	5	\$2.99
 Fake Moustache Translator Communications Fake Moustache Translator attaches between nose and mouth to double as a language translator and identity concealer. Sophisticated electronics translate your voice into the desired language. Wriggle your nose to toggle between Spanish, English, French, and Arabic. Excellent on diplomatic missions.	\$599.99	<input type="text" value="1"/> Update Remove	5	\$599.99
				\$602.98

[Checkout](#)

Figure 5-4. The Cart index page

The Index View

The Index view lists all of the `Orders` for the currently logged in Customer. Add a new MVC View Page to the `Cart` folder, clear out the existing code, and add the following markup:

```

@model CartViewModel
@{
    ViewData["Title"] = "Index";
    var cartTotal = 0M;
}
<h3>@ViewBag.Header</h3>
<div>
    <div id="foo" name="foo"></div>
    <table class="table table-bordered product-table">
        <thead>
            <tr>
                <th style="width: 70%;">Product</th>

```

```

        <th class="text-right">Price</th>
        <th class="text-right">Quantity</th>
        <th class="text-right">Available</th>
        <th class="text-right">Total</th>
    </tr>
</thead>
@for (var x = 0; x < Model.CartRecords.Count; x++)
{
    var item = Model.CartRecords[x];
    cartTotal += item.LineItemTotal;
    @Html.Partial("Update", item)
}
<tfoot>
    <tr>
        <th>&nbsp;</th>
        <th>&nbsp;</th>
        <th>&nbsp;</th>
        <th>&nbsp;</th>
        <th><span id="CartTotal">@Html.FormatValue(cartTotal, "{0:C2}")</span></th>
    </tr>
</tfoot>
</table>
<form asp-controller="Cart" asp-action="Buy" asp-route-customerId="@ViewBag.CustomerId">
    <input type="hidden" asp-for="@Model.Customer.Id" />
    <input type="hidden" asp-for="@Model.Customer.FullName" />
    <input type="hidden" asp-for="@Model.Customer.EmailAddress" />
    <input type="hidden" asp-for="@Model.Customer.Password" value="FillerData" />
    <input type="hidden" asp-for="@Model.Customer.TimeStamp" />
    <div class="pull-right">
        <button class="btn btn-primary">Checkout</button>
    </div>
</form>
</div>

```

The Razor block at the top creates a variable to hold the total price of the cart and sets the page Title. After standard layout markup, the next Razor block iterates through `CartRecords` in the `CartViewModel`. For each record, the `cartTotal` variable is updated with the `LineItemTotal`, and then the `CartRecord` is rendered using the `Update.cshtml` partial view. After rendering all of the records, the `cartTotal` is displayed. The final block of markup uses the Form Tag Helper to render the form to execute the Buy action on the `CartController`.

After the closing `div` tag, add the following Razor and JavaScript to the Scripts section:

```

@section Scripts
{
    @{
        await Html.RenderPartialAsync("_ValidationScriptsPartial");
    }
    <script language="javascript" type="text/javascript">
        function success(data, textStatus, jqXHR) {
            "use strict";
            updateCartPrice();
        }
    </script>
}

```

```

function error(jqXHR, textStatus, errorThrown) {
    "use strict";
    alert('An error occurred: Please reload the page and try again.');
```

```

}
function complete(jqXHR, textStatus) {
    "use strict";
    //Executes after the AJAX call is complete, regardless of success or failure
}
function getSum(total, num) {
    "use strict";
    return total + Math.round(num.innerText * 100)/100;
}
function updateCartPrice() {
    "use strict";
    var list = $('span[id^="rawTotal"]');
    var total = $.makeArray(list).reduce(getSum, 0);
    $('#CartTotal')[0].innerText = '$' + parseFloat(total).toFixed(2).toString().replace(/
(\d)(?=(\d\d\d)+(?!\d))/g, "$1,");
}
</script>
}

```

The first Razor block loads the validation scripts. The next block of JavaScript sets up the event handlers for the Ajax function (covered in the `CartRecordViewModel.cshtml` editor template). If the call succeeds, the total cost is updated. If the call fails, the error function instructs the user to reload the page. The complete method is for illustration purposes only—it doesn't do anything in this example. The method will execute after the Ajax call is completed, regardless of success or failure.

The `getSum` method is a reduce function that iterates through the array and returns the sum total of all elements. The `Math.Round` function (along with multiplying by 100 and then dividing by 100) is recommended when working with currency in JavaScript to prevent the well documented rounding issues. The final method gets all of the line item total elements, converts the list of elements to a true array, and then executes the reduce function using the `getSum` method. The last line utilizes a JavaScript trick to properly format a number into U.S. currency.

The Update Partial View

The Update partial view renders each individual cart record on the Index view. Add a new MVC View Page named `Update.cshtml`, clear out the content, and add the following markup:

```

@model CartRecordViewModel
<tr id="row_@Model.Id">
    <td>
        <div class="product-cell-detail">
            
            <a class="h5" asp-controller="Products" asp-action="Details"
                asp-route-id="@Model.ProductId">@Html.DisplayFor(model => model.ModelName)
            </a>
            <div class="small">@Html.DisplayFor(model => model.CategoryName)</div>
            <div class="small text-muted">@Html.DisplayFor(model => model.Description)</div>
        </div>
    </td>

```

```

<td class="text-right">
  @Html.DisplayFor(model => model.CurrentPrice)
</td>
<td class="text-right cart-quantity-row">
  @Html.EditorForModel()
</td>
<td class="text-right">
  @Html.DisplayFor(model => model.UnitsInStock)
</td>
<td class="text-right cart-quantity-row">
  <span id="rawTotal_@Model.ProductId" class="hidden">@Model.LineItemTotal</span>
  <span id="total_@Model.ProductId">@Html.DisplayFor(model => model.LineItemTotal)</span>
</td>
</tr>

```

Each `tr` is given a unique name based on the `Id` of the `CartRecordViewModel` so the Ajax function knows exactly what to replace. The `CartRecord` is rendered by the `CartRecordViewModel.cshtml` editor template.

The Editor Templates

The `CartController` has a single editor template, used to display the `Quantity` and the `Update` and `Remove` commands.

The CartRecordViewModel Editor Template

Create a new folder named `EditorTemplates` in the `Views\Cart` folder. Add a new MVC View Page named `CartRecordViewModel.cshtml` into this folder. Clear out the existing markup and add the following:

```

@model CartRecordViewModel
<form asp-controller="Cart" asp-action="Update" id="updateCartForm" method="post"
  asp-route-customerId="@Model.CustomerId" asp-route-id="@Model.Id"
  data-ajax="true" data-ajax-mode="REPLACE-WITH" data-ajax-update="#row_@Model.Id"
  data-ajax-method="POST" data-ajax-success="success" data-ajax-failure="error"
  data-ajax-complete="complete">
  <div asp-validation-summary="ModelOnly" class="text-danger"></div>
  <span asp-validation-for="Quantity" class="text-danger"></span>
  <input type="hidden" asp-for="Id" />
  <input type="hidden" asp-for="CustomerId" />
  <input type="hidden" asp-for="CategoryName" />
  <input type="hidden" asp-for="Description" />
  <input type="hidden" asp-for="UnitsInStock" />
  <input type="hidden" asp-for="ModelName" />
  <input type="hidden" asp-for="ModelNumber" />
  <input type="hidden" asp-for="ProductId" />
  <input type="hidden" asp-for="LineItemTotal" />
  <input type="hidden" name="@nameof(Model.TimeStampString)"
    id="@nameof(Model.TimeStampString)" value="@Model.TimeStampString" />
  <input type="hidden" name="@nameof(Model.LineItemTotal)"
    id="@nameof(Model.LineItemTotal)" value="@Model.LineItemTotal" />

```

```

<input type="hidden" asp-for="CurrentPrice" />
<input type="hidden" asp-for="ProductImage" />
<input type="hidden" asp-for="ProductImageLarge" />
<input type="hidden" asp-for="ProductImageThumb" />
<input asp-for="Quantity" class="cart-quantity" />
<button class="btn btn-link btn-sm">Update</button>
</form>
<form asp-controller="Cart" asp-action="Delete" asp-route-customerId="@Model.CustomerId"
      asp-route-id="@Model.Id" id="deleteCartForm" method="post">
  <input type="hidden" asp-for="Id" />
  <input type="hidden" asp-for="TimeStamp" />
  <button class="btn btn-link btn-sm">Remove</button>
</form>

```

The view uses Form Tag Helpers to create two different forms. The first for updating the Quantity and the second for removing a CartRecord completely. The following two hidden input elements don't use Input Tag Helpers, but use the old Razor syntax for their names and properties. This is to get around an Ajax caching issue with input Tag Helpers:

```

<input type="hidden" name="@nameof(Model.TimeStampString)"
      id="@nameof(Model.TimeStampString)" value="@Model.TimeStampString" />
<input type="hidden" name="@nameof(Model.LineItemTotal)"
      id="@nameof(Model.LineItemTotal)" value="@Model.LineItemTotal" />

```

In the Update form, there are tags that haven't been discussed before. The data-ajax tags are supplied by the jQuery-ajax-unobtrusive JavaScript library. If you used the Ajax Form Helpers in previous versions of ASP.NET MVC, this is the new and improved method for implementing an Ajax form.

Table 5-4 lists the data-ajax attributes introduced by the jQuery-ajax-unobtrusive.js library.

Table 5-4. Ajax Tags Supplied by the jQuery-ajax-unobtrusive JavaScript Library

HTML5 data-ajax Tag	Mapping to JavaScript Ajax Calls
data-ajax	True or False, enabling or disabling AJAX on the Form submission.
data-ajax-mode	How the AJAX call should affect the data-ajax-update element. One of Before, After, or REPLACE-WITH.
data-ajax-update	HTML element to update on the Ajax return call.
data-ajax-method	POST or GET
data-ajax-success	JavaScript method to call after a successful Ajax call.
data-ajax-failure	JavaScript method to call after a failed Ajax call. Maps to error method.
Data-ajax-complete	JavaScript method to call after the Ajax call completes.

The Orders Views

The `OrdersController` views are responsible for displaying the `Orders` and `OrderDetails` for the currently logged in `Customer`. Create a new folder named `Orders` under the `Views` folder.

The Index View

The `Index` view lists the top-level information for the `Orders`, as shown in [Figure 5-5](#).

Order History

ORDER NUMBER	DATE ORDERED	DATE SHIPPED	TOTAL	
0	3/14/2017	3/29/2017	4424.90	Order Details

Figure 5-5. The *Order History* view

Add a new MVC View Page named `Index.cshtml` to the `Views\Orders` folder. Clear out the templated code and replace it with the following:

```
@model IList<Order>
<h3>@ViewBag.Header</h3>
@for(int x=0;x<Model.Count;x++)
{
    var item = Model[x];
    <div class="order-card">
        <div class="order-card-heading">
            <div class="row">
                <div class="col-sm-2">
                    <label>Order Number</label>
                    <a asp-action="Details" asp-route-customerId="@item.CustomerId"
                        asp-route-orderId="@item.Id">@Html.DisplayFor(model => item.Id)
                    </a>
                </div>
                <div class="col-sm-2">
                    <label asp-for="@item.OrderDate"></label> @Html.DisplayFor(model => item.OrderDate)
                </div>
                <div class="col-sm-3">
                    <label asp-for="@item.ShipDate"></label> @Html.DisplayFor(model => item.ShipDate)
                </div>
                <div class="col-sm-3">
                    <label asp-for="@item.OrderTotal"></label> @Html.DisplayFor(model => item.OrderTotal)
                </div>
                <div class="col-sm-2 order-actions">
                    <a asp-action="Details" asp-route-customerId="@item.CustomerId"
                        asp-route-orderId="@item.Id" class="btn btn-primary">Order Details</a>
                </div>
            </div>
        </div>
    </div>
}
}
```





The view uses the Bootstrap grid system for the layout and a mix of Tag Helpers and Razor Helpers to display the data.

The Details View

The Details view shows the individual items in an order, as shown in Figure 5-6.

Order Details

DATE ORDERED 12/18/2016	DATE SHIPPED 1/2/2017
BILLING ADDRESS:	SHIPPING ADDRESS:
John Doe 123 State Street Whatever, UT 55555 P: (123) 456-7890	John Doe 123 State Street Whatever, UT 55555 P: (123) 456-7890

Product	Price	Quantity	Total
 Escape Vehicle (Water) Camouflaged as stylish wing tips, these 'shoes' get you out of a jam on the high seas instantly. Exposed to water, the pair transforms into speedy miniature inflatable rafts. Complete with 76 HP outboard motor, these hip heels will whisk you to safety even in the roughest of seas. Warning: Not recommended for beachwear.	\$1,299.99	2	\$2,599.98
 Fake Moustache Translator Fake Moustache Translator attaches between nose and mouth to double as a language translator and identity concealer. Sophisticated electronics translate your voice into the desired language. Wriggle your nose to toggle between Spanish, English, French, and Arabic. Excellent on diplomatic missions.	\$599.99	3	\$1,799.97
 Universal Repair System Few people appreciate the awesome repair possibilities contained in a single roll of duct tape. In fact, some houses in the Midwest are made entirely out of the miracle material contained in every roll! Can be safely used to repair cars, computers, people, dams, and a host of other items.	\$4.99	5	\$24.95
			\$4,424.90

[Back to Order History](#)

Figure 5-6. The Order Details view

Add a new MVC View Page named `Details.cshtml` to the `Views\Orders` folder. Clear out the existing code and replace it with the following:

```
@model OrderWithDetailsAndProductInfo
@{
    ViewData["Title"] = "Details";
}
<h3>@ViewBag.Header</h3>
<div class="row top-row">
    <div class="col-sm-6">
        <label asp-for="OrderDate"></label>
        <strong>@Html.DisplayFor(model => model.OrderDate)</strong>
    </div>
    <div class="col-sm-6">
        <label asp-for="ShipDate"></label>
        <strong>@Html.DisplayFor(model => model.ShipDate)</strong>
    </div>
</div>
<div class="row">
    <div class="col-sm-6">
        <label>Billing Address:</label>
        <address>
            <strong>John Doe</strong><br>
        </address>
    </div>
</div>
```

```

    123 State Street<br>Whatever, UT 55555<br>
    <abbr title="Phone">P:</abbr> (123) 456-7890
  </address>
</div>
<div class="col-sm-6">
  <label>Shipping Address:</label>
  <address>
    <strong>John Doe</strong><br>
    123 State Street<br> Whatever, UT 55555<br>
    <abbr title="Phone">P:</abbr> (123) 456-7890
  </address>
</div>
</div>
<div class="table-responsive">
  <table class="table table-bordered product-table">
    <thead>
      <tr>
        <th style="width: 70%;">Product</th>
        <th class="text-right">Price</th>
        <th class="text-right">Quantity</th>
        <th class="text-right">Total</th>
      </tr>
    </thead>
    <tbody>
    @for (int x=0;x<Model.OrderDetails.Count;x++)
    {
      var item = Model.OrderDetails[x];
      <tr>
        <td>
          <div class="product-cell-detail">
            
            <a asp-controller="Products" asp-action="Details"
              asp-route-id="@item.ProductId" class="h5">
              @Html.DisplayFor(model => item.ModelName)
            </a>
            <div class="small text-muted hidden-xs">
              @Html.DisplayFor(model => item.Description)
            </div>
          </div>
        </td>
        <td class="text-right">
          @Html.DisplayFor(model => item.UnitCost)
        </td>
        <td class="text-right">
          @Html.DisplayFor(model => item.Quantity)
        </td>
        <td class="text-right">
          @Html.DisplayFor(model => item.LineItemTotal)
        </td>
      </tr>
    }

```

```

</tbody>
<tfoot>
  <tr>
    <th>&nbsp;&nbsp;&nbsp;</th>
    <th>&nbsp;&nbsp;&nbsp;</th>
    <th>&nbsp;&nbsp;&nbsp;</th>
    <th class="text-right">
      @Html.DisplayFor(model => model.OrderTotal)
    </th>
  </tr>
</tfoot>
</table>
</div>
<div class="pull-right">
  <a asp-action="Index" asp-route-customerid="@Model.CustomerId" class="btn btn-
primary">Back to Order History</a>
</div>

```

The Products Views

The ProductsList view is responsible for displaying the product list pages as well as displaying the results of the Search Action method. The Communications Category product list is shown in Figure 5-7.

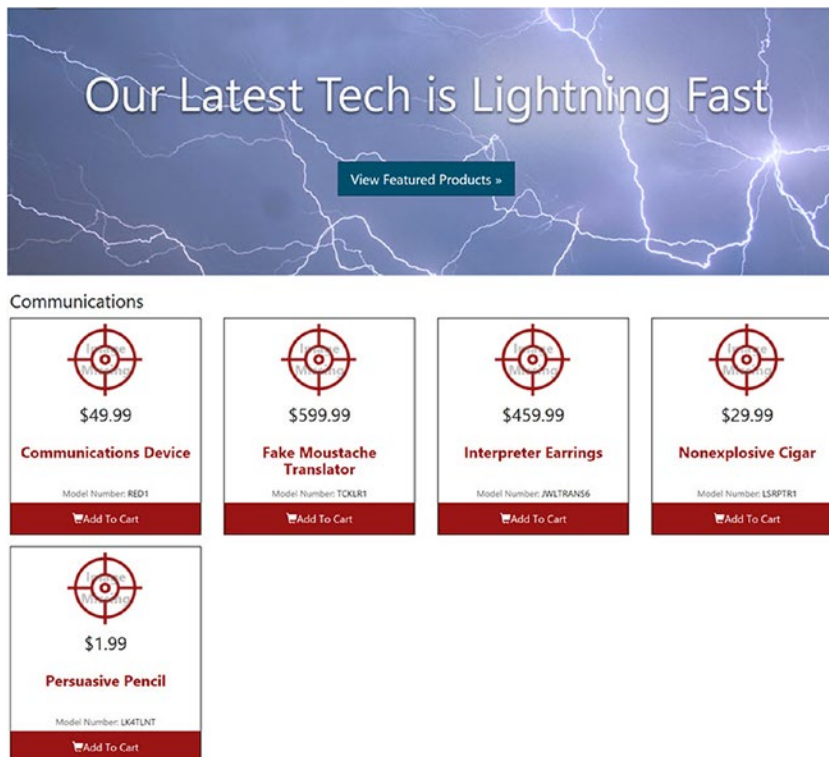


Figure 5-7. The product list featuring the Communications category

The ProductList View

Create a new folder named `Products` under the `Views` folder, then add a new MVC View Page named `ProductList.cshtml` to the `View\Products` folder. Clear out the existing content and replace it with this:

```
@model IEnumerable<ProductAndCategoryBase>
@{
    ViewData["Title"] = ViewBag.Title;
}
<div class="jumbotron">
@if (ViewBag.Featured != true)
{
    <a asp-controller="Products" asp-action="Featured" class="btn btn-info btn-lg">
        View Featured Products &raquo;</a>
}
</div>
<h3>@ViewBag.Header</h3>
<div class="row">
    @for (int x=0;x<Model.Count;x++)
    {
        var item = Model[x];
        @Html.DisplayFor(model => item)
    }
</div>
```

The view renders the `View Featured Products` link (as a button using Bootstrap styling) if the Action method is returning any data except the list of featured product. The View then loops through the records, using the `ProductAndCategoryBase` display template.

The Display Template

Create a new folder named `DisplayTemplates` in the `Views\Products` folder.

The ProductAndCategoryBase DisplayTemplate

Create a new file named `ProductAndCategoryBase.cshtml` in the `Views\Shared\DisplayTemplates` folder. Clear out any code in the template and add the following markup:

```
@model ProductAndCategoryBase
<div class="col-xs-6 col-sm-4 col-md-3">
    <div class="product">
        
        <div class="price">@Html.DisplayFor(x => x.CurrentPrice)</div>
        <div class="title-container">
            <h5><a asp-controller="Products" asp-action="Details" asp-route-id="@Model.Id">
                @Html.DisplayFor(x => x.ModelName)
            </a>
        </h5>
    </div>
```

```

<div class="model-number">
  <span class="text-muted">Model Number:</span> @Html.DisplayFor(x => x.ModelNumber)
</div>
@if (ViewBag.ShowCategory)
{
  <a asp-controller="Products" asp-action="ProductList" asp-route-id="@Model.
  CategoryId">@Model.CategoryName</a>
}
<a asp-controller="Cart" asp-action="AddToCart" asp-route-customerId="@ViewBag.
CustomerId"
  asp-route-productId="@Model.Id" class="btn btn-primary btn-cart">
  <span class="glyphicon glyphicon-shopping-cart"></span>Add To Cart
</a>
</div>
</div>

```

This completes the Core MVC Web Application. The next (and final section) discusses how to run the application in conjunction with the Core MVC RESTful service.

Running the Application

Now that all of the .NET code is complete, it's time to run the application and see the results of all of your hard work. If you were to just press F5 at this point, the application would fail to run with a long list of errors because the entirety of the work is split into three different solutions. While this is a better architectural choice than having everything in one (and enables the JavaScript frontends to reuse the EF and Core MVC RESTful projects as is), this means that there is a little setup needed to debug the site in Visual Studio. The Core MVC service (built in Chapter 3) must be running for the Core Web Application to work. There are two choices to run the Core MVC service in development—you can run it from Visual Studio or by using the .NET CLI. Of course, you could fully deploy the service app, but that is beyond the scope of this book.

Using Visual Studio 2017

Running from Visual Studio is by far the easiest, but it also consumes the most memory. If you have a beefy development machine, this isn't an issue. Load up the *SpyStore.Service* solution and press F5. This will launch the service app in Development mode running on port 8477 (if you changed the port in Chapter 3). Confirm the port by opening the *launchSettings.json* file and checking this block of JSON:

```

"iisSettings": {
  "windowsAuthentication": false,
  "anonymousAuthentication": true,
  "iisExpress": {
    "applicationUrl": "http://localhost:8477/",
    "sslPort": 0
  }
},

```

Next, open the `appsetting.json` file in the `SpyStore.MVC` application and update the following JSON to match the port number from above:

```
"WebServiceLocator": {
  "ServiceAddress": "http://localhost:8477/"
}
```

Using the .NET Command Line Interface (CLI)

The .NET Core CLI can be used to run a Core MVC application (either as a service or a web application) as long as the current directory is the same as the location of the `SpyStore.Service.csproj` file. The caveat is that in order for the sample data to load, the environment must be set to `Development`.

Open a command prompt and navigate to the directory where the `SpyStore.Service.csproj` file is located. Enter the following commands to set the environment and then run the app:

```
set aspnetcore_environment=development
dotnet restore
dotnet build
dotnet run
```

The first line sets the development environment to `Development`. The rest of the lines should be familiar to you now. The NuGet packages are restored, the code is compiled, and the app is run. In the downloads for this chapter, there is a command file named `run_development.cmd` that does just this.

If you followed along with the book, the port for the service app is `40001`. This can be checked by opening the `Program.cs` file and checking the code in the `Main` method:

```
public static void Main(string[] args)
{
  var host = new WebHostBuilder()
    .UseKestrel()
    .UseUrls("http://*:40001/")
    .UseContentRoot(Directory.GetCurrentDirectory())
    .UseIISIntegration()
    .UseStartup<Startup>()
    .Build();

  host.Run();
}
```

Confirm that the `SpyStore.MVC` app is using port `40001` in the `appsetting.json` file:

```
"WebServiceLocator": {
  "ServiceAddress": "http://localhost:40001/"
}
```

Using the Combined Solutions

I realize there are a lot of moving parts in this application, including three solutions, custom NuGet packages, and other configuration details. In the downloadable code for the chapter are two folders containing the applications by direct referencing projects instead of using local NuGet folders.

The folder `SpyStore.Service.Combined` contains the Core MVC RESTful service. The second is `SpyStore.MVCComplete`, which contains the Core MVC Web Application.

The port configurations need to be checked (as discussed in the previous sections) to make sure the Core MVC Web Application can find the RESTful service.

■ **Source Code** The complete `SpyStore.MVCComplete` solution can be found in the `Chapter_05` subdirectory of the download files.

Summary

This chapter completes the coverage of .NET in this book by finishing the `SpyStore.MVC` Web Application. Tag Helpers, one of the two big new features in Core MVC, are introduced right from the top. Next, all of the controllers are built, including a deeper discussion on model binding.

The next section discussed server side and client side binding, creating two custom validation attributes and the client side JavaScript.

Next, View Components, the other big new feature in Core MVC, are introduced, and a custom View Component is added to the project.

The next big section covers all of the views in the application, and this section completed the `SpyStore.MVC` Web Application. The final section demonstrates how to run the application.

PART II



Client-Side Tooling and JavaScript Frameworks

CHAPTER 6



JavaScript Application Tools

Since the beginning of the Internet, web sites have been based on two components, the server and the client. While the server-side is, and has been, managed by numerous frameworks and languages, the client side has predominately been dominated by JavaScript. In 2009, Node.js (<http://nodejs.org>) was written as a framework for creating I/O optimized applications, such as web servers, using JavaScript and executed by Google's V8 JavaScript engine. The release of Node.js and its accompanying package manager (the Node Package Manager—NPM) created a mechanism to use JavaScript on the desktop and server and boosted popularity of the command line tools within the web development space, while also offering a much needed outlet for Open Source Software (OSS) developers to distribute their code to the community. This chapter focuses on the basic usage techniques of some these tools and is meant to help readers get acquainted with the tools that will be used throughout the remainder of the book.

What Tools Are We Covering?

As noted, there are a number of tools that have been created for web development. It would be frivolous to cover all of them. For the purposes of this chapter, only the tools used henceforth will be covered. That is not intended to discredit any of the tools not mentioned. There are many great tools and further research is recommended whenever making large decisions about choosing frameworks and tools for your project. The remainder of this chapter focuses on the following tools.

- Node.JS (<http://nodejs.org>): A cross-platform runtime environment for executing server-side JavaScript.
- Bower (<http://bower.io>): A package manager optimized for client-side frameworks. Built on Node.js and Git.
- Gulp (<http://gulpjs.com>): A streaming build system that runs tasks on code files and is built on Node.js.
- SystemJS (<https://github.com/systemjs/systemjs>): A module loader that provides a shim in the browser to rewrite and enable module loading functionality.
- WebPack (<https://webpack.github.io/>): A build-time module loader that bundles JavaScript files into physical bundles.

Node.js

Node.js is the core of this chapter; the other tools discussed all rely on it. Node.js is a cross-platform runtime environment that executes JavaScript code on a server or desktop machine using the power of Chrome V8 JavaScript interpreter. Since its release, tools such as web servers, service hosts, automated test runners, and numerous web development tools have been created using it. Additionally, the package manager, NPM, which comes bundled with Node.js, has created a way for developers to distribute and consume code modules and increase their development productivity tremendously.

This book covers the following two ways to install Node.js.

- Install manually using the downloadable installer file
- Install using command line driven Chocolatey package manager for Windows

If Node.js v4.0 or greater is already installed, you can skip this section. Otherwise, use one of these mechanisms to get Node.js working. Feel free to jump around to the appropriate section and continue to the “Getting Started with Node.js” section once your machine is set up. Given the Visual Studio focus of this book, installation using a Mac is not covered. Many of the instructions that follow are similar for the Mac, but it is recommended to check the online documentation if you are using a Mac as your primary machine.

Manually Installing Node.js

There are a number of ways to install Node.js. The primary mechanism for installing Node.js is to follow the instructions on the Downloads page, shown in Figure 6-1, of the Node.js site (<https://nodejs.org/en/download>). From there you can download and run the installer, which will walk you through the steps of installation.

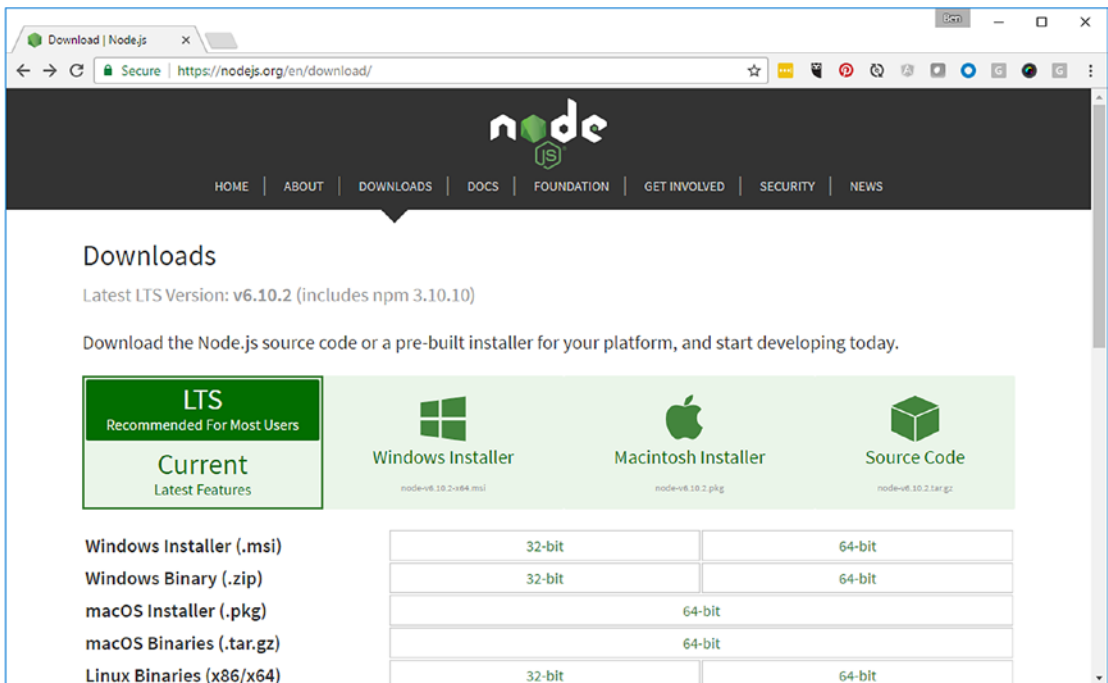


Figure 6-1. Node.js Downloads page

After installation is complete, you should have the node command on your machine. To test this, open a command prompt and type `node -v`. If the command returns the version number, you can proceed to the “Getting Started with Node.js” section. Otherwise, verify your installation by checking your Program Files directory or your PATH variable. Restarting your computer or the command window process can often resolve the problem.

Installing Node using the Chocolatey Package Manager

In addition to the downloadable installer, users can also install Node.js via a command line driven package manager. This is common on other platforms, but some Windows users may not know that this is available using a tool called Chocolatey, which is available at <http://chocolatey.org/>.

In order to use this technique to install Node, you first need to install Chocolatey. The instructions from the web site are very simple; you just paste the following command into an administrator command prompt and Chocolatey will be available.

```
@powershell -NoProfile -ExecutionPolicy Bypass -Command "iex ((new-object net.webclient).DownloadString('https://chocolatey.org/install.ps1'))" && SET PATH=%PATH%;%ALLUSERSPROFILE%\chocolatey\bin
```

After Chocolatey is installed, enter the following command. It should install Node.js and make it immediately available on the command line. Once Chocolatey completes installing Node.js, you can proceed to the “Getting Started with Node.js” section.

```
choco install nodejs
```

Setting Up Visual Studio to Use the Latest Version of Node

Since the release of Visual Studio 2015, the product has come bundled with its own version of Node.js and other command line based web tools. This is useful for users who want to install Visual Studio and automatically take advantage of these tools. Unfortunately, the version of Node.js installed with Visual Studio may not be the latest version. This can cause some issues specifically in the case where code uses features dependent on ECMAScript 6, which wasn’t supported until Node.js v4 (at the time of publishing Visual Studio ships with Node.js v0.10.31, and yes, the versions of Node.js jumped from v0.XX to v4).

To determine the version of Node.js that is shipping with Visual Studio, open a command prompt to the following location.

```
C:\Program Files (x86)\Microsoft Visual Studio\2017\Professional\Web\External
```

From there, execute the command `node -v` on the command line. This should tell you the node version that Visual Studio is currently using.

■ **Note** During the writing of the book, the gulp script in the Task Runner of Visual Studio was throwing the error `ReferenceError: Promise is not defined`. Promises are defined as part of ES6, which only works with Node.js v4 or greater. Projects using these ES6 features will have to make sure Visual Studio is using the latest version of Node.js.

Mads Kristensen has a blog post¹ that describes this issue in depth. The easiest way to make Visual Studio use the latest version of Node.js is to install Node.js on your machine using one of the previously described mechanisms. This should include a reference to the `node` command as part of the `PATH` environment variable. Then in Visual Studio, under **Tools** ► **Options** ► **Projects and Solutions** ► **Web Package Management** ► **External Web Tools**, you can arrange the priority of the paths so that the global `PATH` reference has higher precedence than the path where the bundled version of node exists. Figure 6-2 shows the correct setting for these paths to ensure the Visual Studio and your command line are using the same version of node.

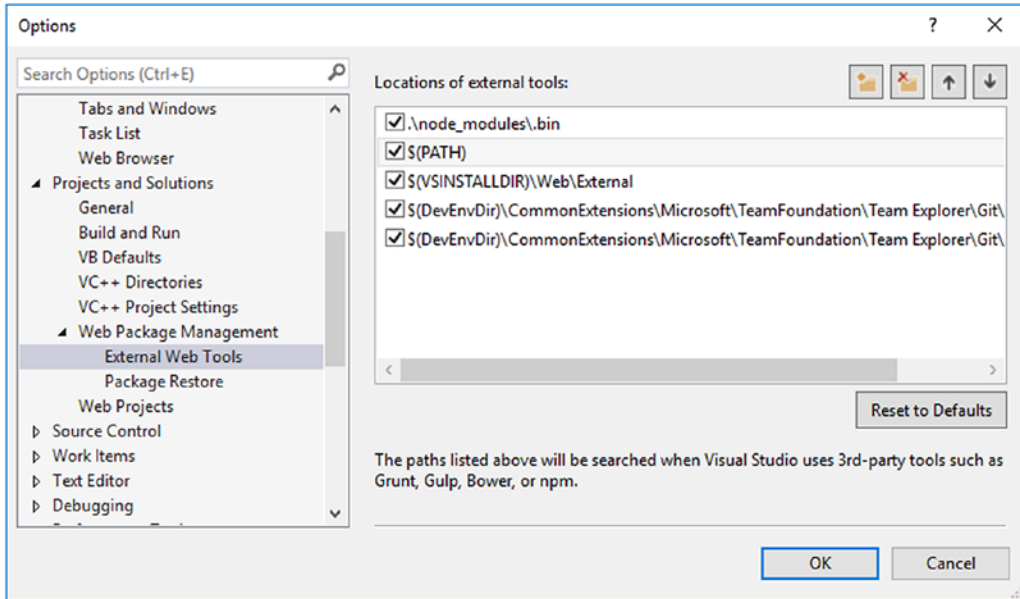


Figure 6-2. Visual Studio option for external web tools (NPM, Bower, Gulp, etc.)

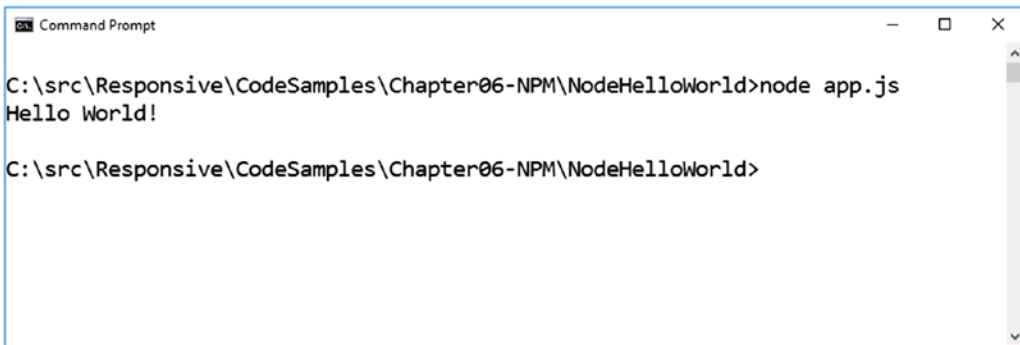
Getting Started with Node.js

Now that Node.js is installed, you are free to run JavaScript code on your desktop. On par with any getting started tutorial is an exercise in creating a basic Hello World application. To begin, create a new directory. In this new directory, create a new file called `app.js`. In this file, enter the following code:

```
console.log("Hello World");
```

This code should look very familiar to code you would write in the browser to log something. That's because it's just JavaScript code. Once the `app.js` file with this code is saved, open a command prompt to the new directory and run `node app.js`. You should see the same output pictured in Figure 6-3.

¹<https://blogs.msdn.microsoft.com/webdev/2015/03/19/customize-external-web-tools-in-visual-studio-2015/>



```

Command Prompt
C:\src\Responsive\CodeSamples\Chapter06-NPM\NodeHelloWorld>node app.js
Hello World!

C:\src\Responsive\CodeSamples\Chapter06-NPM\NodeHelloWorld>

```

Figure 6-3. Node.js result for Hello World sample project

■ **Note** Visual Studio Code offers an excellent cross-platform development and debugging experience for Node.js. While this book focuses on Visual Studio 2017, if you have Visual Studio Code installed, you can open the previous example and run it with great results. Simply open the folder and, after it has loaded, click F5 to run the application. The first time you run the project you will be prompted to configure your launch settings. Select Node.js for the debugging environment, then click OK to confirm.

What's great about Node.js is that any JavaScript syntax will also work here. You can try other code in this `app.js` file. You can also experiment directly with Node by just typing `node` on command line to enter the REPL, or Read, Eval, Print, Loop. As it states, REPL reads the code on the prompt, runs the JavaScript `eval` command on that code, prints the output to the output window, and then loops, or repeats.

Introduction to NPM

While JavaScript itself is very powerful, much of the power of Node.js comes from modules that you can include with your project. To manage these modules in your application, Node.js uses a companion tool called the Node Package Manager, or NPM for short. NPM is installed as part of Node.js, so if you already have Node.js installed you should be good to go.

In this next example, NPM will be used to bring in a module called `request`, which can read the contents of a web request. This module will then be used in our code to read the results from a web server. To start, return to the command prompt and navigate to the project directory. Within this directory, enter the following command:

```
npm install request
```

This command goes out to the `npm` registry and searches for the module `request`. If it finds the module, it installs it to the `node_modules` folder, reads the package descriptor, and then proceeds to install all of its dependencies. This process repeats until all packages and all dependencies are resolved.

NPM installs modules, by default, to the `node_modules` folder. This folder contains a hierarchy of module dependencies. Figure 6-4 shows the folder after installing `request`.

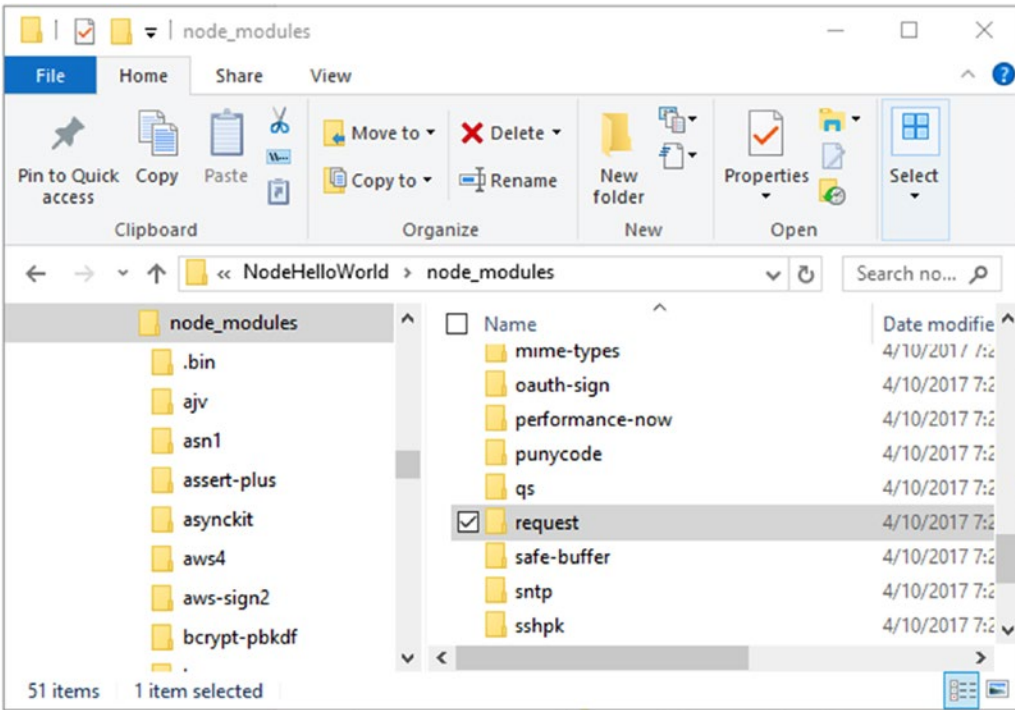


Figure 6-4. Node modules folder with hierarchy of packages noted on left side folder tree

■ **Note** NPM and Node.js were originally intended for server-side use and the dependencies can get pretty large. This is not particularly suited for frontend web application libraries. Bower, which is a different package management solution, attempts to resolve this and is covered in more depth later in this chapter.

Once the request module has been added, it can be used in your code using the `require('request')` function. The `require` command is used to load modules and is covered later in this chapter. Open the `app.js` file and replace your Hello World code with the following code. Then save the file.

```
var request = require('request');
request('http://www.npmjs.com', function (error, response, body) {
  if (!error && response.statusCode == 200) {
    console.log(body)
  }
})
```

Back on the command prompt in the project directory, enter the command `node app.js` again. This will go out to the <http://www.npmjs.com> site and download the HTML. Then in the callback function, the code checks the response and writes out the HTML to the command window.

Saving Project Dependencies

While the previous example works fine on the current machine, it is not set up to run on other machines yet. For many reasons, including file size and licensing, it is not recommended to store and transmit the actual `node_modules` folder. In the previous example, the dependency for `request` was installed in the `node_modules` folder, but no record of that installation was stored. So, if the project were opened on another machine, it would receive an error attempting to load the required module. Users expect to restore packages when they download a project but the dependencies have to be stored in the package manifest in order for NPM to know what to install.

■ **Note** When creating a project on GitHub, the default `.gitignore` file for JavaScript projects ignores the `node_modules` folder. This means that if you download a JavaScript project from online or someone else downloads your code, one of the first steps is to restore all of the `node_modules`, or dependencies, needed by the project using the `npm install` command.

This is where the `package.json` file comes into play. `Package.json` is the application manifest for any Node.js application. This file includes many important details about the project, including what dependencies exist. The `package.json` file is nothing more than a text file and can be created by hand, but it's often easier to open a command prompt in the directory and call the following command.

```
npm init
```

This command will take you through a wizard asking questions on how to populate the `package.json` file. To select the defaults just keep pressing enter until there are no more questions. After running `npm init` on the project folder, `npm` creates a `package.json` file that looks something like Figure 6-5.

```

1  {
2    "name": "nodehelloworld",
3    "version": "1.0.0",
4    "description": "",
5    "main": "app.js",
6    "dependencies": {},
7    "devDependencies": {},
8    "scripts": {
9      "test": "echo \"Error: no test specified\" && exit 1"
10   },
11   "author": "",
12   "license": "ISC"
13 }
14

```

JSON file length: 255 lines: 14 Ln: 1 Col: 1 Sel: 0|0 UNIX UTF-8 INS

Figure 6-5. Default `package.json` after `npm init`

Now that the `package.json` file is set up, it can be used to track the dependencies for the project. Dependencies can either be added manually to the `package.json` or they can be added by `npm` via installing or linking. Now that the `package.json` file exists, reinstall the `request` package, except this time install it using the `--save` flag.

```
npm install request --save
```

Running this command will update your `package.json` and add something similar to the following.

```
"dependencies": {
  "request": "^2.72.0"
},
```

By default, `npm` stores the dependency version number as the current package version or anything compatible. This is denoted by the `^` sign on the version number for the package. The version numbers are managed in a *semver* format, which stands for semantic versioning. A full list of formats is available at <https://github.com/npm/node-semver>.

Once the dependencies for the project have been configured, you are free to distribute your source code without the extra bloat of the package dependency files. When the project is downloaded or the `node_modules` folder is deleted, the code will not work and the packages will need to be restored. To restore the packages for the project, you just need to run the following command.

```
npm install
```

This `npm install` command without any package name argument reads the `package.json` and installs all the packages that are listed as dependencies.

Executable Packages

In addition to packages that can be used by your projects, NPM can also be used to install packaged executables. The executables get installed to the `node_modules` directory just like the other packages. In addition to the module, a `.cmd` script is created under the `node_modules\``.bin` directory.

To see this in action let's install a great package that turns any directory into a web server. Like other packages, it first needs to be installed. To install the package, open a command line to the project directory and enter the following command.

```
npm install http-server
```

Now that this package is installed, take a look at the `node_modules` directory in Figure 6-6. Notice that a `.bin` directory is created and inside that directory there is an `http-service.cmd` script.

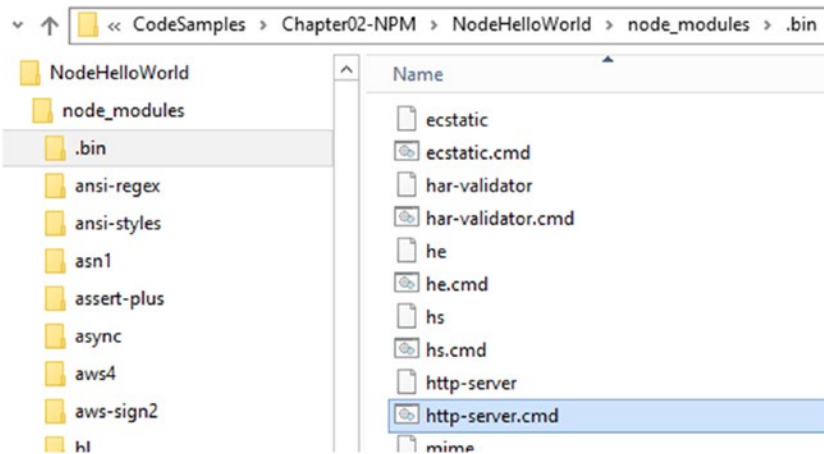


Figure 6-6. `node_modules` directory with executable packages included

To run that command, enter the following in your command line for the project directory.

```
.\node_modules\.bin\http-server -o
```

This will host a server in the project directory under port 8080. By providing the `-o` argument it will automatically open the site in your browser (Figure 6-7), but if not, you can just open your browser to `http://localhost:8080`.

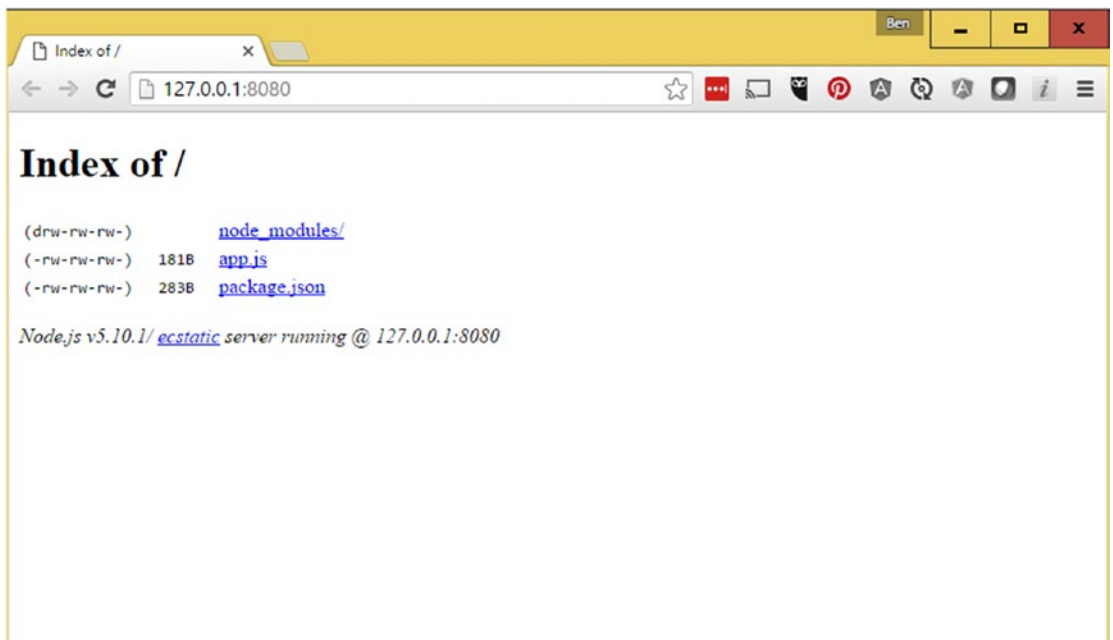


Figure 6-7. The `NodeHelloWorld` project folder running in a browser using the `http-server` package

Installing Packages Locally vs. Globally

Having executable packages in your project facilitates many tasks for development, but in some cases you might want to have the package accessible everywhere, not just in your project folder. Luckily, NPM has an option to install packages globally. This is how Bower, Gulp, and WebPack are installed. Essentially, what installing globally does is install the package to %UserProfile%\AppData\Roaming\npm, a path that was added to the PATH environment variable when Node.js was installed.

To install the package globally from the previous example, use the following command. The `-g`, or `--global`, tells npm to install the package globally on the machine.

```
npm install http-server -g
```

The purpose of this section is to give you an understanding of what Node.js is and what it's like to manage and use different packages, both locally to a project and globally on your system. While you can write some powerful code with Node.js this book focuses on other frameworks and the knowledge of Node.js and its tools are all that's required.

Node.js and NPM are extremely powerful and, with new packages being released daily, their power is expanding. The remainder of this chapter focuses on the other tools that will be used throughout this book.

Bower

While Node.js was originally intended for the server, its popularity in the JavaScript community and its package management capabilities made it quite alluring for client-side developers and frameworks as well. Unfortunately, due to some core package storage strategies, npm is not ideal for use in the browser. There are numerous techniques to circumvent these issues. As an alternative to NPM, specifically designed for client-side frameworks that are ultimately deployed to the web, Bower <http://bower.io> was released.

To fully understand the implications of using NPM v2 or earlier rather than the client-side framework optimized Bower, it's best to look at it in terms of an example. Take two packages that both have a dependency on jQuery, `jquery-daterangepicker` and `jquery-file-upload`. When installing these two packages with NPM, the resulting folder structure for the modules would look something like Figure 6-8.

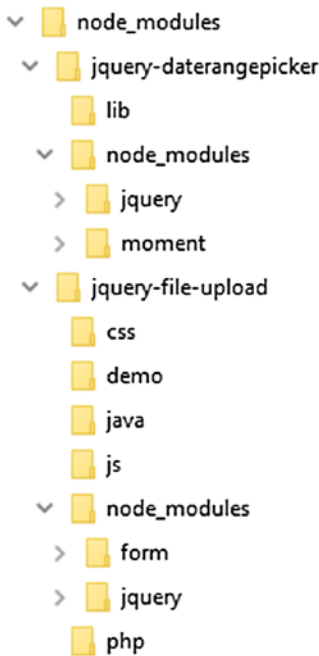


Figure 6-8. The `node_modules` folder with duplicate jQuery dependencies nested within each project

Installing the same two packages via Bower would result in the folder structure shown in Figure 6-9.

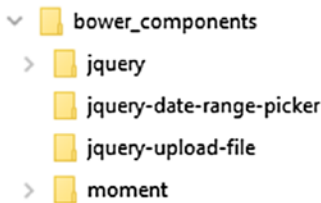


Figure 6-9. The `bower_components` folder showing how dependencies are flattened

■ **Note** NPM version 3 has made some improvements to its dependency resolution strategy, <https://docs.npmjs.com/how-npm-works/npm3>. These features challenge much of Bower's value proposition, but it still remains popular. There are also options within `npm`, such as `browserify` and `dedupe`, which will force `npm` to work better with frontend packages.

Notice the way jQuery is stored in both of the previous structures; the difference between them centers on nested dependencies. In NPM, nested dependencies are connected and versioned as part of their parent. This works fine for the desktop and server components. With Bower, on the other hand, all dependencies are resolved globally and stored in a flat structure. This cuts down on the duplication and surfaces conflict resolution to package install time.

Cutting down on duplication is extremely important for performance on web applications. It also cuts down on the potential bugs and the number of files that need to be deployed to webserver.

Bower and Visual Studio

Like Node.js, a version of Bower and its components are bundled with Visual Studio. If you plan on using the Bower package management tools built-in to Visual Studio and described later in this chapter, there is no need to install Bower using the following instructions. If, in addition to Visual Studio, you want Bower to be accessible from the command line, it is recommended that you follow these installation instructions.

Installing Bower Prerequisites

Installing Bower is so simple that the instructions are literally four lines long. While installing Bower is easy, it does require a few prerequisites in order to work properly. The first prerequisite is Node.js, which was covered previously in this chapter. The second prerequisite to use Bower is Git. If Git is already installed, just verify that it is in your PATH environment variable and accessible from the command line. If the command works, then proceed to the Installing Bower section.

Installing Git

There are a number of options for installing Git. For the latest instructions, it is best to view the official instructions on the Git web site at <https://git-scm.com/book/en/v2/Getting-Started-Installing-Git>. In the case of Windows, you can download an executable from the Git web site and install it manually. This installer will prompt a few installation questions about file type handling and line ending characters. The screens provide detailed instructions, so pick the best option for you and your team and continue through the installation.

Alternatively, Chocolatey can be used to install Git, similarly to the node example from before. After Chocolatey is installed, open a command prompt with administrator access. In that prompt, enter the following command.

```
choco install git
```

This will kick off the Chocolatey installer, which will download the tools and install them on your machine. After the installation completes, enter the command `git` on the command line. This should provide the Git help documentation if it's installed properly. If you get a message stating that the command wasn't recognized, try closing all instances of the command prompt and restarting them. If you still have issues beyond that, you may need to make sure the path where Git is located is in the PATH environment variable. By default, Git is installed under `C:\Program Files\Git\cmd`.

Installing Bower

After all the prerequisites are installed, you can install Bower. Official instructions can be found online at <http://bower.io>. In keeping with the simplicity of their instructions, here is the one line command needed to install Bower.

```
npm install -g bower
```

Using Bower

Now that Bower is installed, the command should be accessible from the command prompt. Using Bower is comparable to using `npm`. Many of the commands should be familiar.

As you do with NPM, you can install packages straight into a new folder by using the `bower install` command. Although, if you are building a project that will ever be used by other developers, you should initialize the directory and save references to the packages needed. To do this, open a command prompt to a new folder for the project and run the following command.

```
bower init
```

This command will prompt some basic questions to describe your project. To accept the defaults, just press Enter until you have returned to the command prompt. After answering the questions, there will be a `bower.json` file (Figure 6-10) in the project directory. This file contains configuration information about your project and its dependencies. Like the `package.json` file, this file will also be used in the event that the project is published.



```

1  {
2    "name": "BowerHelloWorld",
3    "homepage": "https://example.com",
4    "authors": [
5      "Ben Dewey <ben@example.com>"
6    ],
7    "description": "",
8    "main": "",
9    "license": "MIT",
10   "ignore": [
11     "**/*.*",
12     "node_modules",
13     "bower_components",
14     "test",
15     "tests"
16   ],
17   "dependencies": {}
18 }

```

Figure 6-10. The `bower.json` configuration file for a sample project

Installing Bower Packages

Many of the frontend JavaScript frameworks are available from Bower. To see a full list of packages and to search for different options, go to the Bower web site at <http://bower.io> (Figure 6-11).

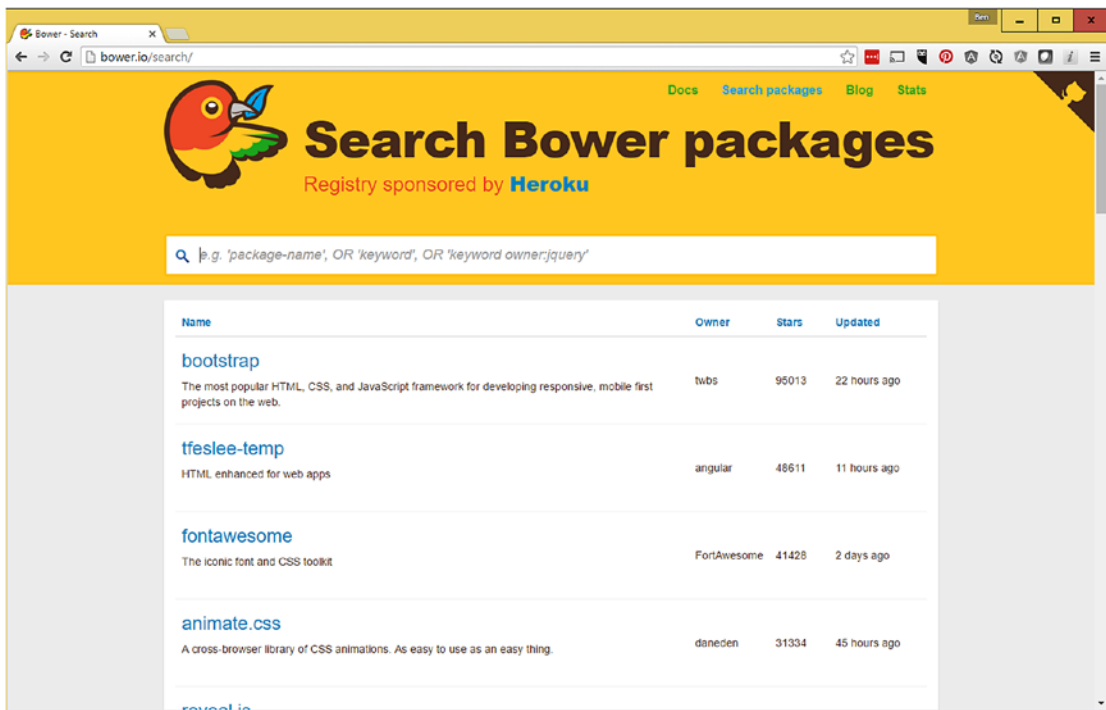


Figure 6-11. The search packages page of the Bower web site

After locating the package in the registry, open a command prompt to the project root directory. From there, enter the command `bower install <pkgname> --save` where `<pkgname>` is the name of the package to be installed. The following is an example of the command that would be used to install bootstrap in the selected project. Similarly to NPM, using the `--save` flag when installing will record the package as a dependency entry in the `bower.json` file.

```
bower install bootstrap --save
```

■ **Note** The Visual Studio team has included a number of features with Visual Studio to simplify the use of these tools in the IDE. For example, if there are packages, either `npm`, Bower, or NuGet packages, installed in a project, the packages will automatically be restored when the project is opened.

Installing Bower Packages Using Visual Studio

In addition to using Bower on the command line, Visual Studio 2015 has included a package management experience in the IDE that works in a similar way to the NuGet tools. In order to see this in action, you need a new ASP.NET Core project as a starting point.

Start by opening Visual Studio and selecting ASP.NET Core Web Application (ASP.NET Core) from New Project section of the Start page or select More Project Templates and find it in the list. Enter a name to match your sample project then click OK. The New ASP.NET Project dialog will appear. From the dialog, select the Empty ASP.NET template on the top left of the screen, as shown in Figure 6-12, and click OK.

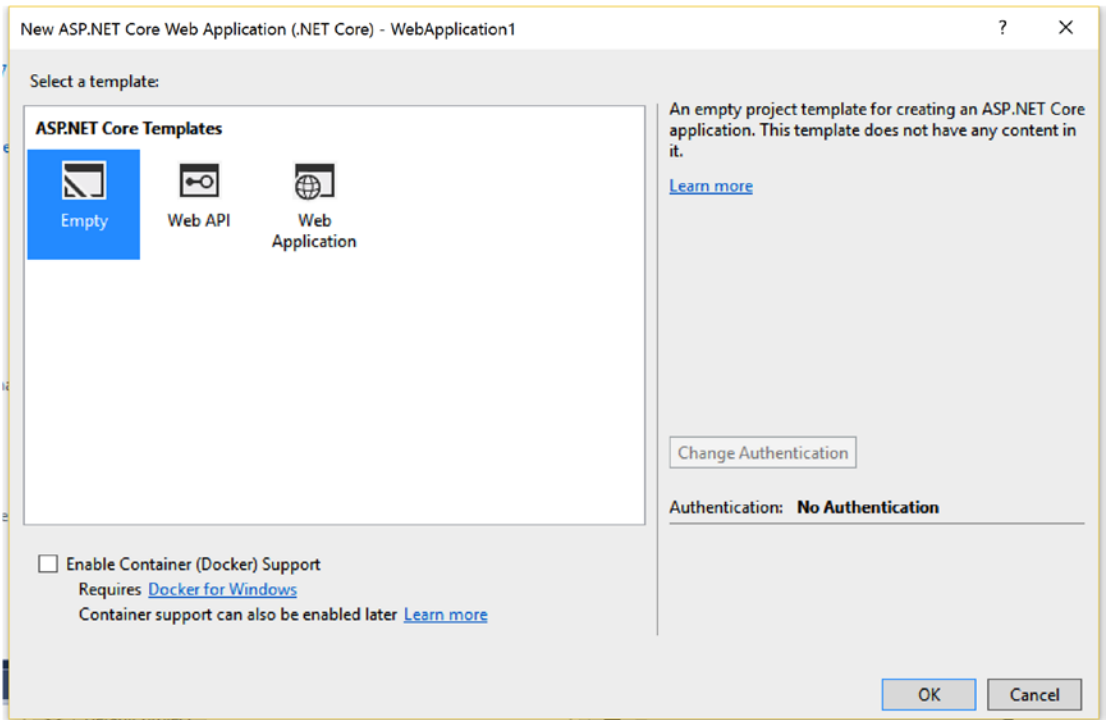


Figure 6-12. *New ASP.NET project dialog*

Once a new project has been created, right-click on the project in Solution Explorer and select Manage Bower Packages. This will launch the Visual Studio Bower Package Manager UI shown in Figure 6-13.

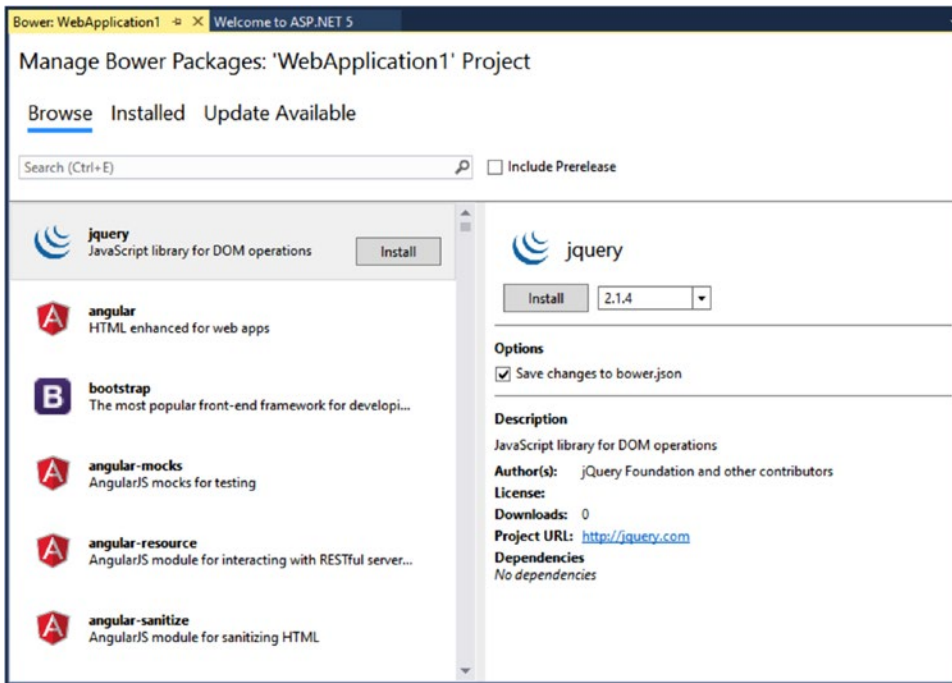


Figure 6-13. Manage Bower Package UI in Visual Studio

■ **Note** Examples here show the Manage Bower Package UI for ASP.NET Core projects. Managing Bower packages in Visual Studio is available to both ASP.NET and ASP.NET Core project types.

Within the Manage Bower Packages UI there are three tabs—Browse, Installed, and Update Available. The Browse tab provides a way to search the registry and install packages from online. The Installed tab shows a list of packages that are already installed with your project. The packages don't have to actually be installed as part of your `bower.json` file. The bower packages are just read from the `bower_components` folder. The final tab, one that makes using Visual Studio so powerful, is the Update Available tab. This tab shows any particular package where a newer version exists on the server and provides an easy one-click mechanism to upgrade the package. This is a feature that is not easily achievable from the command line tools.

■ **Note** At the time of publishing, Visual Studio does not provide a package management solution for NPM packages similar to the Bower package manager. Visual Studio does provide a list of Bower and NPM packages as dependencies in Solution Explorer of ASP.NET Core projects, but nothing to edit or view NPM packages.

Deciding when to use Bower and NPM can be tricky. Many packages are published in both places and there are numerous articles online that give conflicting advice. Generally speaking though, Bower is for client side frameworks and NPM is for all other packages. To make this decision even more confusing, there is a newcomer to space named JSPM (<http://jspm.io>), for JavaScript Package Manager. They have goals of resolving the concerns with NPM and Bower by creating what they call a frictionless package management

solution that is designed to work with module loaders that will be discussed later. At the time of publishing, many frameworks hadn't fully adopted JSPM over NPM and Bower, but given the speed of change in the JavaScript ecosystem, it wouldn't be surprising to see shifts in popularity over time. For the most part though, these tools have overlapping functionality and web developers can be successful with any of them.

Gulp

Unlike .NET, and other managed languages, JavaScript does not have a compiler. There is no step that forces you to create an executable to run your site. That being said, many frameworks and tools have been created to check and verify JavaScript code. The frameworks and tools need to execute on top of the project code and manipulate it in some way so it is ready for production use. These tasks are typically handled by a task runner, which defines tasks that need to occur on the code as part of a build step.

Gulp (<http://gulpjs.com>) is a popular task runner for building web applications. Gulp is a code-based, rather than configuration-based, framework that leverages the built-in Node Stream functionality. Their web site touts four core benefits:

- Gulp is easy to use, because it uses JavaScript code over custom configuration files.
- Gulp is efficient, because it uses streams with no intermediate I/O writes.
- Gulp is high quality, because it enforces standards on plugins.
- Gulp is easy to learn, because of its simplicity.

Gulp can be used for a number of different tasks. Here is a list of possible tasks you might encounter that would benefit from the automation of a task runner such as Gulp.

- Copying files from one location to another.
- Concatenating multiple JavaScript files into one.
- Verifying JavaScript syntax is correct and meets coding guidelines.
- Adding version and header information to code files.
- Injecting code or HTML-based on files in a directory.
- Transpiling JavaScript or TypeScript code.
- Preparing packages for deployment to package management registries.
- Compiling LESS/SASS code in to CSS files.
- Minifying JavaScript and CSS code to shrink file size for quicker downloads.
- Performing framework specific functions, such as wrapping HTML template code in angular template directives.

For a complete list of plugins for Gulp, see the Gulp site. It contains a list of available plugins (<http://gulpjs.com/plugins>). You can also search the NPM gallery for packages that start with gulp-. The next few sections of this chapter provide examples of using Gulp in the context of the apps that are created for the book. Before running the first example, Gulp needs to be installed. The next section describes installing Gulp using NPM.

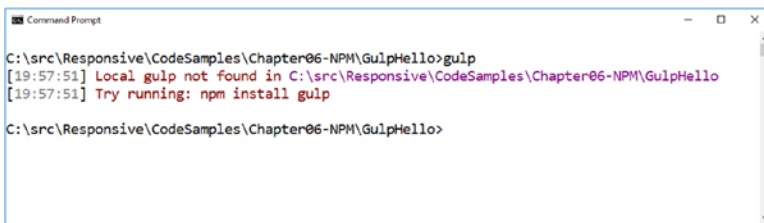
Installing Gulp

Installing Gulp, and all the Gulp plugins for that matter, is very easy once NPM is installed. Using NPM you can globally install the `gulp-cli`, or the Gulp command line interface, which makes the command available on your system. Open the command prompt to any directory and run the following command to install Gulp.

```
npm install -g gulp-cli
```

This command adds the `gulp` command to the `PATH` variable on the current system. If you run the `gulp` command in a directory, at this point, you will get an error message informing you that Gulp is not installed locally, as seen in Figure 6-14. From there you need to navigate to a new project directory, called `HelloGulp`, and install Gulp with your project using the following commands.

```
npm init
npm install gulp --save
```



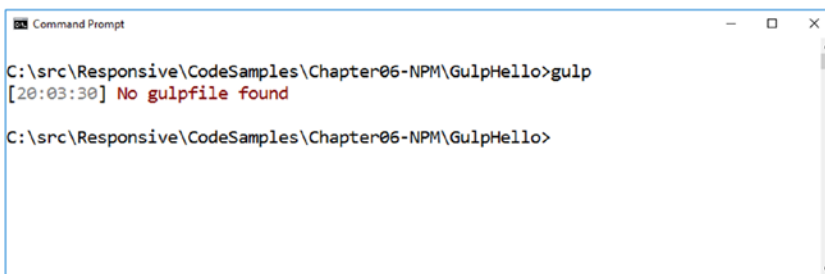
```

Command Prompt
C:\src\Responsive\CodeSamples\Chapter06-NPM\GulpHello>gulp
[19:57:51] Local gulp not found in C:\src\Responsive\CodeSamples\Chapter06-NPM\GulpHello
[19:57:51] Try running: npm install gulp
C:\src\Responsive\CodeSamples\Chapter06-NPM\GulpHello>

```

Figure 6-14. Gulp install error when Gulp isn't installed locally

Now that the project directory is initialized and Gulp is installed, you should be able to run Gulp again. This time a new message should appear stating `No gulpfile found`, as seen in Figure 6-15.



```

Command Prompt
C:\src\Responsive\CodeSamples\Chapter06-NPM\GulpHello>gulp
[20:03:30] No gulpfile found
C:\src\Responsive\CodeSamples\Chapter06-NPM\GulpHello>

```

Figure 6-15. Gulp error when no `gulpfile` exists in the current directory.

This message means that Gulp is installed and working properly. This segues into the next section on creating your first gulp tasks in the `gulpfile`.

Copying Files Using Gulp

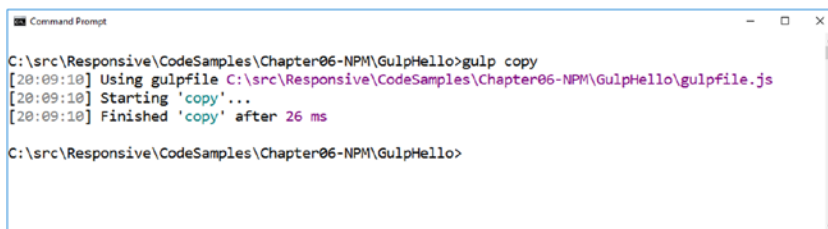
One of the first and most basic commands to do with Gulp is copy files. Whether you are moving files from your `node_modules` or `bower_components` folder to your `wwwroot` folder or copying files to some output directory, the command is the same.

To get started, navigate to your `HelloGulp` folder and create a new text file called `hello.txt`. This file can contain any text; it'll just be used to demonstrate copying. Next you'll need to create a `gulpfile.js`, which will act as the definition file for your tasks. Into this new `gulpfile.js`, paste the following code.

```
var gulp = require('gulp');

gulp.task('copy', function() {
    return gulp.src('hello.txt')
        .pipe(gulp.dest('./output/'));
});
```

The first line of the code specifies that the code requires Gulp to run. After that, it calls a function that defines the task to be executed. A task has a name, logic, and optional dependent or child tasks, which will be discussed in later sections. In the previous example, the task is named `copy`. The next argument is a function that is executed when the task is run. The return type of the function is intended to be a Node.js Stream object. A stream is ideally suited for this task because it doesn't require writing any intermediate files to disk and it simply provides a definition, or pipeline, of activities to be completed. The actual execution is handled by the task runner after the function has returned. Each activity in the pipeline handles inputs and returns outputs, which get used as inputs in the next activity. In the case of the previous example, the initial stream is defined by the `gulp.src("hello.txt")` call. This tells Gulp that the input for the pipeline is the `hello.txt` file. This file then is passed through to the next activity in the pipeline via the `pipe` command. In this simple task the only other activity in the pipeline is to write the file to a new location, as specified by the `gulp.dest('./output/')` command. Figure 6-16 shows an example of this task running. If you run this command, you will notice that the `hello.txt` file is copied to a new directory called `output`.



```
Command Prompt
C:\src\Responsive\CodeSamples\Chapter06-NPM\GulpHello>gulp copy
[20:09:10] Using gulpfile C:\src\Responsive\CodeSamples\Chapter06-NPM\GulpHello\gulpfile.js
[20:09:10] Starting 'copy'...
[20:09:10] Finished 'copy' after 26 ms

C:\src\Responsive\CodeSamples\Chapter06-NPM\GulpHello>
```

Figure 6-16. The result of running the first Gulp copy example

■ **Note** Grunt (<http://gruntjs.com>) is another popular task runner that is based on configuration syntax. Due to the speed of Gulp, the recent popularity in the community, and the preferences of the authors, this book has chosen to focus on Gulp for its examples. Readers are encouraged to evaluate both Gulp and Grunt to best understand the pros and cons of each.

Dependencies in Gulp

Oftentimes during a build process, you might need to perform a sequence of tasks with some certainty that a previous step was completed successfully. An example is the need to compile some SASS code into CSS prior to copying it to an output directory or copying dependent files to a reference location prior to building a project. Fortunately, this is built into Gulp using the second parameter of a provided overload.

To demonstrate this, the copy task is going to be extended to clean up, or delete, the contents of the output directory, before running its copy logic. The cleanup task requires an additional module, `del`, which needs to be installed via NPM before it can be used. To add this module, run the following command on the command line.

```
npm install del --save
```

Now that the required `del` module has been installed, open the `gulpfile` and modify it to include the following code.

```
var gulp = require('gulp');
var del = require('del');

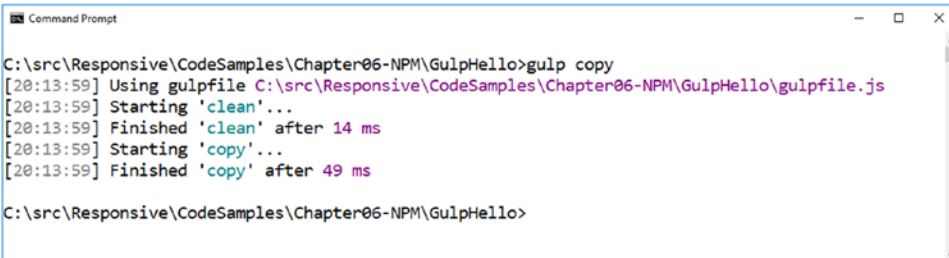
gulp.task('clean', function() {
    return del('./output/');
});

gulp.task('copy', ['clean'], function() {
    return gulp.src('hello.txt')
        .pipe(gulp.dest('./output/'));
});
```

What's different about this task than the previous example is the second parameter. Instead of an anonymous function like before, the copy task contains an array of string representing dependent tasks as its second parameter and the task function is supplied in the third parameter. This means that when the copy task is called, it will call `clean`. To see this example in action, open the command line and run the following command.

```
gulp copy
```

Figure 6-17 shows the output of running this command. Even though the copy task was the only task requested, the clean task was completed prior to executing the copy task as it was defined as a dependency.



```
Command Prompt
C:\src\Responsive\CodeSamples\Chapter06-NPM\GulpHello>gulp copy
[20:13:59] Using gulpfile C:\src\Responsive\CodeSamples\Chapter06-NPM\GulpHello\gulpfile.js
[20:13:59] Starting 'clean'...
[20:13:59] Finished 'clean' after 14 ms
[20:13:59] Starting 'copy'...
[20:13:59] Finished 'copy' after 49 ms

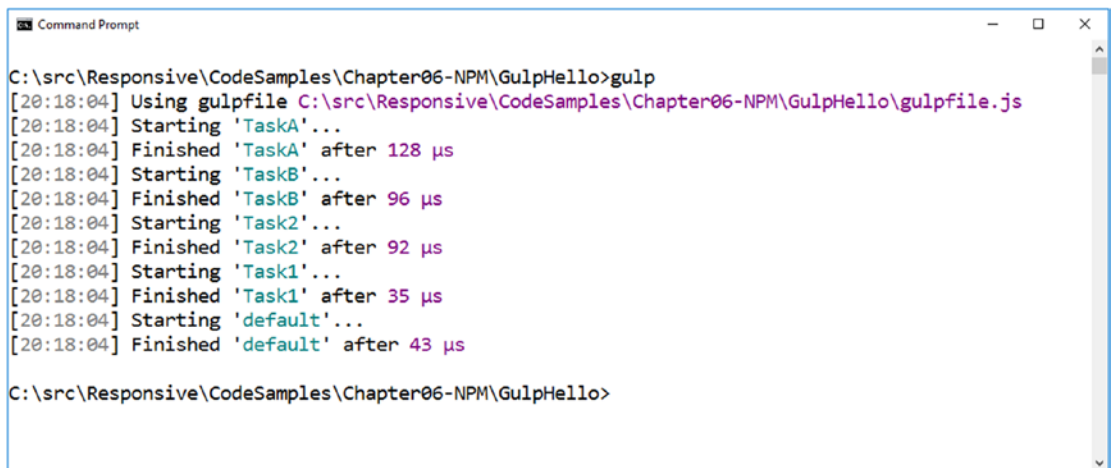
C:\src\Responsive\CodeSamples\Chapter06-NPM\GulpHello>
```

Figure 6-17. The result of running the first Gulp copy example with the dependency task defined

This is a very basic usage of dependent tasks, but dependencies can very easily get complex in their definition. Gulp is very smart in the way it handles its dependencies. Take the hierarchy of dependencies outlined here.

- `gulp.task('default', ['Task1', 'Task2']);`
 - `gulp.task('Task1', ['TaskA', 'Task2']);`
 - `gulp.task('Task2', ['TaskA', 'TaskB']);`
 - `gulp.task('TaskA', ...);`
 - `gulp.task('TaskB', ...);`

While the default task requires Task1 and Task2, Gulp actually parses all the children tasks down into TaskA and TaskB and determines that TaskA and TaskB plus Task2 all need to be run prior to running Task1. Figure 6-18 shows the actual result from this example and shows the order of execution for the tasks.



```

Command Prompt
C:\src\Responsive\CodeSamples\Chapter06-NPM\GulpHello>gulp
[20:18:04] Using gulpfile C:\src\Responsive\CodeSamples\Chapter06-NPM\GulpHello\gulpfile.js
[20:18:04] Starting 'TaskA'...
[20:18:04] Finished 'TaskA' after 128 μs
[20:18:04] Starting 'TaskB'...
[20:18:04] Finished 'TaskB' after 96 μs
[20:18:04] Starting 'Task2'...
[20:18:04] Finished 'Task2' after 92 μs
[20:18:04] Starting 'Task1'...
[20:18:04] Finished 'Task1' after 35 μs
[20:18:04] Starting 'default'...
[20:18:04] Finished 'default' after 43 μs

C:\src\Responsive\CodeSamples\Chapter06-NPM\GulpHello>

```

Figure 6-18. The result of running the Gulp example with nested tasks

■ **Note** Another common example for using Gulp is concatenation, which is the process of combining multiple JavaScript or CSS files into one file. With the HTTP2 specification starting to be adopted by browsers and web servers, this task is losing relevancy. For this reason, the code in this book does not cover concatenation. For more information, see the HTTP2 documents at <https://http2.github.io/>

Task Runner Explorer Within Visual Studio

Another new tool that has been added to Visual Studio is called the Task Runner Explorer. It is available under View ► Other Windows ► Task Runner Explorer. This tool reads the tasks from Gulp or Grunt files and provides a way in Visual Studio to double-click and execute a task. Figure 6-19 shows an example of the Task Runner Explorer for a sample web application.

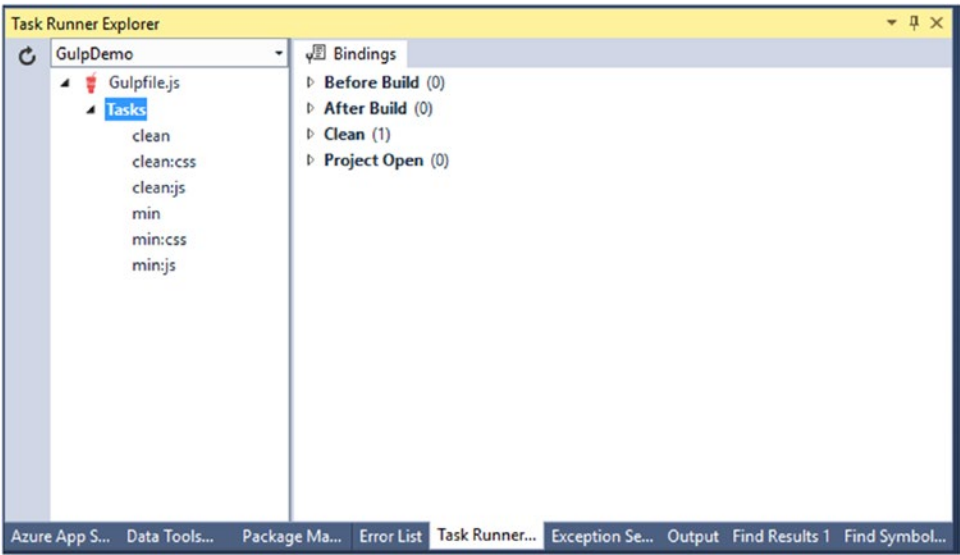


Figure 6-19. The Task Runner Explorer from Visual Studio

On the left side of the Task Runner Explorer are the tasks from the `gulpfile.js`. Double-clicking on them opens a new tab on the right side and displays the output from the task in that new window.

The Task Runner Explorer provides a feature that allows developers to bind the gulp tasks to build operations on the project. Right-clicking on the task on the left side displays options for bindings. Under the Binding drop-down are options to select and deselect a binding association with the selected task. The available options for binding are Before Build, After Build, Clean, and Project Open.

Gulp Conclusion

There are numerous possibilities for Gulp tasks. Fortunately, the Gulp documentation provides examples of many of these tasks in a format they call *recipes*. These recipes, shown in Figure 6-20, can be found online at <https://github.com/gulpjs/gulp/tree/master/docs/recipes>.

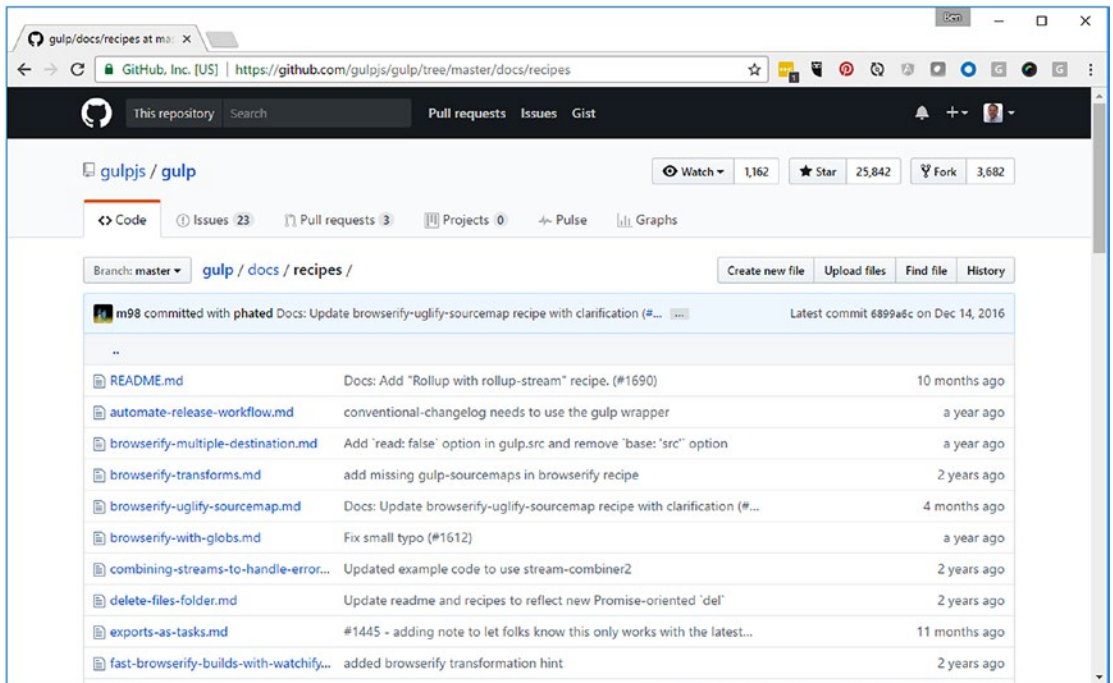


Figure 6-20. The documentation on Gulp includes recipes for starting ideas

For traditional .NET developers, there is definitely a learning curve for these tools such as NPM, Bower, and Gulp. The trend seems to be around creating a command line tool to execute some logic, then creating UIs and extensions on top of these command line tools to improve the experience. The Visual Studio team is continuing to add features at a rapid cadence and these features expose more and more of these command line tools in a user friendly way. For now, it's important for developers to understand both worlds, the command line and the frontend UIs.

Module Loaders

Modularity is a core concept in JavaScript and it allows us to build larger and more complex applications. The latest versions of JavaScript support modules natively as part of the loader specification (<https://github.com/whatwg/loader/>). Unfortunately, at the time of publishing, modern browsers do not support modules. Therefore, if you want to use modules, you have to use a third-party module loader.

In terms of module loaders, there are two options that will be covered—SystemJS, which is an online module loader for the browser, and WebPack, which is a build time pre-bundling tool that combines the modules ahead of time into a single file that can be loaded from the file system. This section will cover both module loaders.

■ **Note** ASP.NET has long had a feature called ASP.NET Bundling. These features remain in ASP.NET Core as part of the visual studio gallery item called Bundler & Minifier, available at <https://marketplace.visualstudio.com/items?itemName=MadsKristensen.BundlerMinifier>. While this feature is available and can have similar results, if you manually include all the files in the bundle, it lacks the support for reading the JavaScript and determining its required imports based on the code.

What Is a Module

A module in JavaScript is a way to represent a small unit of reusable code. They are critical to maintainability, when building large applications in JavaScript. In fact, many of the single page app frameworks covered in this book are based on modules. Module have also been used previously in the chapter even if it wasn't apparent. Any time the `require` command was used previously, that was a module.

In order to understand this concept, let's take a simple calculator. This calculator, as seen in the following code and saved as `calculator.js`, represents the reusable calculator code that will be used by the module loader.

```
module.exports = (function() {
  function Calculator() {}
  Calculator.prototype.add = function(left, right) {
    return left + right;
  };

  return Calculator;
})();
```

This code represents an ES5 version of the class in JavaScript that has methods. In this case, there is an `add` method that will be used by our application to add two numbers. On the first line of this code, it sets the `module.exports` value equal to the `Calculator`. This command defines the calculator as a reusable module for other code and allows the calculator to be imported into other parts of the application.

The following code represents a simple application that uses the calculator. For the purposes of this example, assume that this code is in a file called `app.js`.

```
var Calculator = require('./calculator.js');

var calculator = new Calculator();

var addButton = document.getElementById("addButton");
addButton.addEventListener('click', function() {
  var var1Field = document.getElementById('var1');
  var var2Field = document.getElementById('var2');
  var resultField = document.getElementById('result');

  var var1 = var1Field.value;
  var var2 = var2Field.value;
  var result = calculator.add(+var1, +var2);
  resultField.value = result;
});
```


Notice that the app class uses the calculator from the previous code via its module. To do this it loads the calculator using the `require` statement. This sets the resulting variable, in this case `Calculator`, to the `module.exports` from the previous example. From there that `Calculator` class reference can be used to create a new instance of the `Calculator` class called `calculator`, which can then be used to call the `add` method.

The final component to this simple JavaScript calculator application is the HTML. The following code represents the HTML needed for the application.

```
<!doctype html>

<html lang="en">
<head>
  <meta charset="utf-8">

  <title>Calculator Example with SystemJS</title>
</head>
<body>
  <div class='container'>
    <h1>Calculator Example with SystemJS</h1>

    <input id="var1" />
    <label>+</label>
    <input id="var2" />

    <button id="addButton">=</button>

    <input id="result" />

    <!--Script to load modules and app.js -->
  </div>
</body>
</html>
```

The HTML for the application is very simple. There are essentially three text fields—two for input and one for output—and a button. They are all linked by IDs that are used by the `app.js` file.

Unfortunately, if `script` tags to the previous two snippets of code—`calculator.js` and `app.js`—were added, the browser would throw an error. This is because it doesn't know what the `require` statement means. The latest browsers, at the time of publishing this book, do not support module loading. This is where module loaders come into play. For both the SystemJS and WebPack module loader examples, this calculator code will be used.

SystemJS

SystemJS is a configuration based module loader. SystemJS provides a shim for the browser that modifies the JavaScript files that are imported and provides a mechanism to enable exporting and requiring of modules.

In order to enable SystemJS for this application, the SystemJS library has to be added to the project. This code can be found in the SystemJS folder of the example code for this chapter. In the case of the example code, the SystemJS library is added via NPM. To do this open a new directory and include the three files—`app.js`, `calculator.js`, and `index.html`—to it. From there, open the command prompt and run the following command.

```
npm install systemjs
```

This will download the `systemjs` library into the `node_modules` folder where it can be used. Once the library has been downloaded, include the following `script` tag in the bottom of the HTML file where the comment states.

```
<script src="node_modules/systemjs/dist/system.src.js"></script>
```

The minimal configuration needed in this case is to tell SystemJS what the root file is for the application. In this case, code is added to tell SystemJS to load the `app.js` file. From there, SystemJS will investigate the `app.js` file and find all the dependencies and ensure they are loaded. The following code instructs SystemJS to load the `app.js` file.

```
<script>
  System.import('app.js').catch(console.error.bind(console));
</script>
```

With these two `script` tags, the entire application will be loaded. In larger applications, this can be extremely powerful, by loading numerous application and third-party scripts without any additional code. In addition, this code sets some basic error handling to ensure errors are written out to the console.

Now that the `index.html` file has been modified to load the application using SystemJS, the folder can be hosted and launched via the `http-server` package discussed earlier in this chapter. If you're using the example code, the `http-server` call has been included in the `scripts` section of the `package.json` file, as shown in the following code.

```
"scripts": {
  "serve": "http-server -o"
},
```

In order to execute this command, use the `npm run` command for the `serve` task as shown in the following code.

```
npm run serve
```

To understand what SystemJS is doing, open the developer tools in your browser and inspect the results of the `app.js` and `calculator.js` files that are loaded. Looking at the `calculator.js` file, you'll see that the original file has been modified and has some additional code wrapped around it. The following code shows the changes that SystemJS has made.

```
(function(System, SystemJS) {(function(require, exports, module, __filename, __dirname,
global, GLOBAL) {
  // Original Calculator.js code removed for brevity.
}).apply(__cjsWrapper.exports, __cjsWrapper.args);
})(System, System);
```

Not only has SystemJS inspected the `app.js` and determined that it needed to load the `calculator.js` file, but it has provided implementations for the code that would have previously thrown errors, such as `module`, `exports` and `require`. These alternate implementations, or shims, are provided in the `__cjsWrapper.args` parameter.

This code provides a nice alternative to manually including multiple `script` tags for everything your application needs. In addition, it allows developers to take steps in the right direction in terms of maintainability and modularity of code within JavaScript applications. This example is intended to demonstrate the concepts of module loading with SystemJS. There are additional configuration options and numerous ways to structure your application. For additional information on SystemJS, review the documentation online at <https://github.com/systemjs/systemjs>.

Webpack

In the previous example, SystemJS was used to provide an in-browser shim to load modules. While this accomplished the goals, it can cause some strain on the browser as it needs to load multiple files. Depending on the size of the application, including its dependencies, there can be hundreds of scripts that need to be loaded. WebPack has gained popularity in recent time and offers module-loading capabilities at build time. It concatenates the numerous files that the application needs into a concise bundle or set of bundled files.

Unlike SystemJS, WebPack is a command line build tool that works on the developer's machine or a build machine. To get started with WebPack, use the following command to install WebPack globally on your system.

```
npm install webpack -g
```

Webpack also needs to be installed in the project directory. To do this, create a new directory and include the three calculator files from the prior section—`app.js`, `calculator.js`, and `index.html`. The example code for this chapter also contains a `WebpackCalculator` folder, which contains the completed code for this sample. After the directory is created, navigate to the directory from the command line and execute the following command to install WebPack locally to the project.

```
npm install webpack
```

Now that WebPack is installed, it needs to be configured. Unlike SystemJS, which used the `index.html` file to configure the module loader, WebPack runs on the developer's machine so it needs to be configured separately. To configure WebPack, create a new file called `webpack.config.js` and include the following code.

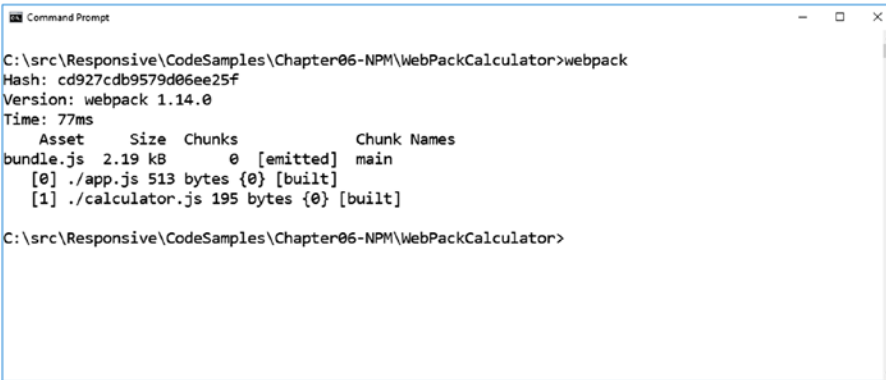
```
module.exports = {
  entry: "./app.js",
  output: {
    path: __dirname + "/dist",
    filename: "bundle.js"
  }
};
```

There are two configuration options in the WebPack config. First is the entry point, `app.js`. WebPack scans for any required files starting with the entry point, just like SystemJS did in the previous example. The second configuration option is the output. This tells WebPack where it should write the contents of the application after it has combined them. In this case, all the JavaScript in the application will be writing out to a file called `dist/bundle.js`. There are a number of different configuration options, including using different types of loader. Information about additional configuration options can be found online on the WebPack web site at <https://webpack.github.io/>.

Now that WebPack is installed and configured, it can be executed using the `webpack` command. To execute WebPack and generate the `bundle.js` file from the application code, run the following command.

```
webpack
```

Figure 6-21 shows the output message that is displayed when calling the `webpack` command.



```

C:\src\Responsive\CodeSamples\Chapter06-NPM\WebPackCalculator>webpack
Hash: cd927cdb9579d06ee25f
Version: webpack 1.14.0
Time: 77ms
   Asset      Size  Chunks             Chunk Names
bundle.js  2.19 kB      0  [emitted]  main
   [0]  ./app.js  513 bytes {0} [built]
   [1]  ./calculator.js 195 bytes {0} [built]
C:\src\Responsive\CodeSamples\Chapter06-NPM\WebPackCalculator>

```

Figure 6-21. The output message when running the webpack command on the calculator project

The last step needed is to include a script tag for this newly generated `bundle.js` file. The following code shows the line that needs to be included in the bottom of the `index.html` file.

```
<script type="text/javascript" src="dist/bundle.js" charset="utf-8"></script>
```

At this point the application can be run using `http-server` like the previous example. If you are using the example code, there is a built-in script included as part of the `package.json`, which allows you to run the command via NPM. The following command will run the application and launch the browser to the calculator app.

```
npm run serve
```

It's great to see the module loader working with just a single file and minimal configuration, but how does this work? The following code shows the `bundle.js` file that was generated for the calculator projects.

```

/*****/ (function(modules) { // webpackBootstrap
/*****/      // The module cache
/*****/      var installedModules = {};

/*****/      // The require function
/*****/      function __webpack_require__(moduleId) {

/*****/          // Check if module is in cache
/*****/          if(installedModules[moduleId])
/*****/              return installedModules[moduleId].exports;

/*****/          // Create a new module (and put it into the cache)
/*****/          var module = installedModules[moduleId] = {
/*****/              exports: {},
/*****/              id: moduleId,
/*****/              loaded: false
/*****/          };

/*****/          // Execute the module function
/*****/          modules[moduleId].call(module.exports, module, module.exports, __
webpack_require__);

```

```

/*****/
/*****/
// Flag the module as loaded
module.loaded = true;

/*****/
/*****/
// Return the exports of the module
return module.exports;
/*****/
}

/*****/
/*****/
// expose the modules object (__webpack_modules__)
__webpack_require__.m = modules;

/*****/
/*****/
// expose the module cache
__webpack_require__.c = installedModules;

/*****/
/*****/
// __webpack_public_path__
__webpack_require__.p = "";

/*****/
/*****/
// Load entry module and return exports
return __webpack_require__(0);
/*****/ } }
/*****/ (
/* 0 */
/**/ function(module, exports, __webpack_require__) {
    var Calculator = __webpack_require__(1);
    var calculator = new Calculator();
    var addButton = document.getElementById("addButton");
    addButton.addEventListener('click', function() {
        var var1Field = document.getElementById('var1');
        var var2Field = document.getElementById('var2');
        var resultField = document.getElementById('result');
        var var1 = var1Field.value;
        var var2 = var2Field.value;
        var result = calculator.add(+var1, +var2);
        resultField.value = result;
    });
/**/ },
/* 1 */
/**/ function(module, exports) {
    module.exports = (function() {
        function Calculator() {}
        Calculator.prototype.add = function(left, right) {
            return left + right;
        };
        return Calculator;
    })();
/**/ }
/*****/ ]);

```

The first thing to notice at the top of the file are a bunch of lines that start with `/******/`. This is the WebPack code and contains similar shim code as the SystemJS project. In addition, the contents of the `app.js` and `calculator.js` have been included as part of this file, as denoted by the 0 and 1 comments in the code. Instead of wrapping these calls with hooks, WebPack modifies the files, as you scan down the example code notice that `require('./calculator.js')` from the `app.js` file has been replaced with `__webpack_require__(1)`.

Comparing this code, and the architecture of WebPack, to SystemJS, you'll notice that WebPack combines all the code into this single file. This means that all the work of looking up the module dependencies is completed when the command is run and written out to this file as opposed to by the client in the browser. There are some benefits to that from a performance perspective, but what might not be obvious are the caching benefits that might be received from having third-party modules loaded in a consistent versioned state. By combining all the code into a single file, any change, even insignificant changes, can invalidate the cache of the entire application. There are ways around this, in fact the `angular-cli`, which is not covered in this book, but uses WebPack under the covers and creates separate bundles for third-party scripts and for application scripts.

Despite their differences, SystemJS and WebPack provide similar functionality. These sections have just scratched the surface on what is possible with these tools. They can also be used to load more than just JavaScript, such as style sheets and images. For more information, see the documentation online at <https://webpack.github.io/>.

■ **Source Code** This chapter's source code can be found under the `Chapter_06` subdirectory of the download files.

Summary

Hopefully, this chapter has provided a basic introduction to the tools used for web development so they aren't completely foreign when they come up in later chapters. There is much more to explore on these topics by visiting the respective web sites and navigating their tutorials.

CHAPTER 7



Introduction to TypeScript

TypeScript is a relatively new language designed and built “on top” of JavaScript. The syntax of TypeScript is the same as JavaScript and any valid JavaScript can also be considered TypeScript. As a language, TypeScript contains several powerful features not included in native JavaScript, including strong typing, object oriented capabilities via classes and interfaces, generics, and access to the newest ECMAScript features. Many of these new features will make TypeScript feel much more comfortable to C# developers than native JavaScript. At its lowest level, TypeScript is a JavaScript transpiler. A *transpiler* is a source to source compiler. Transpilers take a language and add some keywords and functionality to it. Then, at build time, those extra keywords and functionality are compiled back to the original language. To sum up, TypeScript adds features to JavaScript and then compiles those features back to pure JavaScript for deployment. Since the output is standard JavaScript, it runs on any platform without any changes or special requirements.

The concept of writing your JavaScript in a (slightly) higher level language like TypeScript and then transpiling it into JavaScript prior to deployment has a huge number of advantages. The act of transpiling your TypeScript code to JavaScript also provides the point where the compiler can notify the developer of any errors or warnings in the code. This simple concept of “breaking” the build through something as simple as an incorrect variable name is something that is worth its weight in gold to developers new to JavaScript.

Years ago, developers attempted to limit the amount of JavaScript they utilized in their applications as it lacked many of the key features they had learned to rely on in other, seemingly more robust, platforms. JavaScript was un-typed, didn’t support standard object oriented capabilities, and, due to the lack of robust tooling and IDEs, was difficult to write and maintain. Over time, and due to JavaScript’s wide availability and growing adoption, many of these hurdles have been overcome by innovative developers and companies rushing to fill this void. Today, JavaScript enjoys some of the most robust tooling support of any language and, thanks to transpilers like TypeScript, many of the other perceived limitations have been addressed as well. Now, building large and complex applications in JavaScript is not only exponentially easier than at any time in the past but is rapidly becoming the norm.

Why TypeScript?

Maybe the real question is: *Why not TypeScript?* Using TypeScript does not limit the types of devices or browsers in which your code runs as the resulting output is plain JavaScript. Thus, you get all the advantages of a more robust set of language features without any limitation to the reach of your application. You can easily integrate your favorite third-party JavaScript libraries with TypeScript.

■ **Note** TypeScript is a great way for C# developers to learn JavaScript. If you do not have much experience writing large-scale JavaScript, learning JavaScript by starting with TypeScript is a great way to leverage many of the concepts you already know.

One of the challenges with JavaScript development is structuring the code once the application grows. Adding small client side tweaks to an application is easy with straight JavaScript. However, once you try to move everything to the client, which is typical with single page applications (SPAs), things get challenging in a hurry. TypeScript helps by adding keywords and features that promote SOLID development practices on the client. Things like type safety, inheritance, and modules/namespaces make large complex client side code easier to structure, develop, and maintain.

The keywords and features added by TypeScript help the developer provide intent. Meaning the developer can tell the TypeScript compiler that a variable is intended to be a number or a string. This helps IDEs provide IntelliSense and call out design time compilation errors. For example, if you try to assign a string into a variable defined as a number, the compiler will give you an error. IDEs like Visual Studio, which is compiling in the background, will give you instant feedback.

TypeScript allows developers to take advantage of new ECMAScript features that modern browsers do not yet support. Compiler support for new features happens faster than browser support for these same features. Similarly, TypeScript is also very helpful to projects that must support old browser versions. It allows you to select the ECMAScript version you need to support and will compile to the optimal code to support that version.

Finally, TypeScript enjoys a large ecosystem and support network. TypeScript was initially designed and developed at Microsoft by a team ran by Anders Hejlsberg (who also invented the C# language). The TypeScript language design and transpilers are all open source. Although initially developed at Microsoft, the TypeScript community has grown significantly outside the walls of Microsoft. Microsoft's own development tools such as Visual Studio and Visual Studio Code both provide very robust TypeScript integration. In addition, many other tool vendors have built TypeScript integration deeply into their products. Browsers such as Google's Chrome provide native TypeScript debugging within their dev tools. Alternative IDEs, such as JetBrains' WebStorm, provide powerful support for TypeScript development and debugging.

TypeScript Basics

An Overview of TypeScript Syntax

Before we begin to write some of our own TypeScript, let's review some of the basics of the TypeScript syntax. In doing so we will compare how it enhances standard JavaScript and differs from languages such as C#. Describing the full extent of the TypeScript language is beyond the scope of this introductory chapter, but it is very easy to demonstrate some of the key concepts you will find most useful in adopting the language in your own projects.

Datatypes

One of the first obvious differences between TypeScript and standard JavaScript is the support for specifying datatypes for variables, method parameters, and method return types. TypeScript supports a relatively small number of native datatypes, but ability to define your own classes or interfaces allow you to constrain any variable or parameter to whatever type you like.

The most common native datatypes supported by TypeScript are:

- `boolean`
- `number`
- `string`

This list of native datatypes may seem short compared to other languages you may be familiar with (and there are a few other keywords that can be utilized as datatypes, which we'll discuss later). Since all TypeScript must compile to native JavaScript, there wasn't a large advantage in supporting a more granular

breakout of native datatypes. The object oriented and modular capabilities of the TypeScript language allowed others to provide customized types as needed for things like dates, powerful collections, etc. TypeScript also natively supports arrays, nullable variables, generics, and many other features you may find familiar in defining the correct datatypes for your applications.

At its core, the typing system in TypeScript works and behaves very much like a developer would expect based on experiences from strongly typed languages such as C#. Once a variable or parameter is declared, its usage and assignments are enforced to be correct. Like C#, TypeScript also supports the ability to instantiate a variable on the same line as its declaration and to infer the datatype from its instantiation.

TypeScript variables are declared the same as in JavaScript (via `let` or `var`) but a type can be specified after the declaration by decorating the variable with a datatype. A colon (`:`) character is used to separate the variable name from the datatype.

Let's review the following lines of code, which demonstrate some of the most basic means of declaring typed variables in the TypeScript syntax:

```
let x: number;      // declare variable x as a number

var y: string;     // declare variable y as a string

x = 12;           // valid assignment

x = "Hello";      // will cause a compile error

let a: number = 10; // specifying the type of number is redundant, it will be inferred from
                    // the assignment

var b = "Hello";   // b is inferred to be a string
```

As you can see from the previous code, the general syntax for specifying a “type” for a variable in TypeScript is relatively straightforward. If no type is specified for a variable it is assumed to be a special type called `any`. The `any` keyword is also valid in TypeScript anywhere you do not want a specific type and would prefer a variable be allowed to be assigned to anything. The following code demonstrates the use of the `any` keyword in TypeScript:

```
var x: any;

x = 123;          // initial assignment to a number

x = "Hello";     // assigned same variable to a string

var y;           // with no specific datatype specified y is inferred to be any

y = "World";
```

In the previous code, the declaration of variable `y` with no specific datatype tells the TypeScript compiler to infer its datatype as `any`. Since this is how all native JavaScript variables are defined, the use of `any` when a datatype is not specified is what allows existing JavaScript code to function within TypeScript files.

Just as TypeScript supports declaring types on basic variables, the same syntax is utilized for specifying types on method parameters and return types. See the following simple method declaration as an example:

```
calculateArea(length: number, width: number): number {
    return length * width;
}
```

In this method example, the function accepts two parameters and both are declared to have the datatype of `number`. The return type of the function is also declared to be a `number`. In the TypeScript syntax, the return type comes at the end of the method declaration.

As with variable declarations, method parameters can specify a default value. Parameters specifying a default value and coming after all required parameters are essentially treated as optional to the caller of the method.

TypeScript also supports declaring parameters as optional by using the `?` symbol. Optional method parameters specified with the `?` symbol must come after all required parameters. See the following implementation of the `searchPeople` method:

```
function searchPeople(searchFilter: string, maxResults?: number): string[] {
    if (maxResults == undefined)
    {
        maxResults = 10;
    }
    return null;
}
```

In this function implementation, the `maxResults` parameter is specified as optional with the `?` symbol after its name. In this scenario, no default value is given. When optional parameters are specified in this manner and no value is passed from the caller, the variable is assigned a value of `undefined`. This can be checked for by using the `undefined` keyword in TypeScript, as seen in the previous code.

In TypeScript, the `undefined` datatype is one of two special datatypes along with `null`. Both serve specific purposes in the language and are treated differently. The `null` datatype is likely familiar from its usage in languages such as C#. In earlier versions of TypeScript, variables of any datatype could be assigned as `null`. As of the release of TypeScript 2.0, an optional compiler setting called `strictNullChecks` allows this behavior to be flipped and no variable be assigned to `null` unless otherwise specified.

One final feature worth discussing when talking about datatypes within the TypeScript language is the ability to utilize “union” types. Let’s look at the following code:

```
var m: string | number;

m = "Go";

m = 123;

m = null; // error, cannot assign to null
```

In the same lines of code, the variable `m` is declared to have a union datatype, which specified it can either be assigned to a `string` or a `number`. Assignment to any other datatype will produce an error. Assuming the compiler is set to enforce `strictNullChecks`, then an assignment of the variable `m` to type `null` would also produce an error.

To achieve the ability to specify null values when appropriate, we can use the union datatype concept and specify `null` as one of the types a variable can support. This can be seen in the following code:

```
var o: string | null;

o = "Huskers";

o = null; // valid, o can be assigned to null
```

Union datatypes can be specified anywhere a regular datatype is specified: variables, method parameters, method return types, etc. When accessing variables declared as a union datatype, you will only be able to access members that are common among each datatype in the union.

TypeScript also provides some useful type guards capabilities that are especially useful when dealing with union datatypes. These include the use of the `typeof` and `instanceof` operators from JavaScript. The TypeScript compiler will note that a particular usage of a variable exists within a block of code using `typeof` or `instanceof` to ensure the variable is of a specific datatype. Then, within this block, access to type-specific functionality will be available and safe. The following example demonstrates this with the use of the `typeof` operator.

```
var data: string | number | null;

data = 12;

if (typeof data === "number")
{
    // TypeScript treats 'data' as a number in this block
    var asCurrency = data.toFixed(2);
}

data = "Kevin";

if (typeof data === "string")
{
    // TypeScript treats 'data' as a string in this block
    var isValid: boolean = data.startsWith("K");
}
```

Another very useful feature of the TypeScript language is the ability to specify datatypes as enumerations. The following code demonstrates a simple enumeration:

```
enum Status {
    Active,
    Inactive,
    Unknown
}

var _s = Status.Active;
```

While this code is simple, enumerations in TypeScript are much more advanced. They support both constant and calculated enums, flags, and other powerful features.

One final point worth making regarding specifying datatypes in TypeScript is the fact that the resulting JavaScript does not contain any of the type information specified. During the TypeScript compilation process, all typing rules are enforced and validated. When the resulting JavaScript is generated, all the typing information is stripped out of the code. Since JavaScript does not have the concepts of native datatypes, it only makes sense that TypeScript validate and enforce these rules up front and then emit un-typed JavaScript.

Core Constructs

We won't spend much time on the core language syntax of TypeScript because, as mentioned, the syntax of TypeScript is just JavaScript. This means that the core language constructors for such things as for loops and conditional statements are identical to JavaScript.

Classes

Now that we've provided a understanding of how the type system of TypeScript works, let's move on to another key and powerful feature of the TypeScript language: object oriented (OO) features such as classes, interfaces, and inheritance. Experienced JavaScript developers could simulate these capabilities via prototype-based inheritance and ECMAScript 6 will bring classes and inheritance to JavaScript but. With TypeScript, developers can take advantage of these features today.

■ **Note** Classes and these related keywords are not available in ECMAScript 5. They have been added to the ECMAScript 6 standard. This means that once modern browsers implement the new standard you will no longer need TypeScript to compile these keywords. The nice thing about TypeScript is that you can take advantage of these features now and they will run in current browsers.

This section is not intended to go into any of the core principles and patterns of OO but is instead intended to provide an overview of the implementation details for utilizing the core OO constructs in TypeScript.

To jump right in and begin to see how OO capabilities are provided in TypeScript, look at the following class:

```
class Person {
    constructor() {
    }

    nameFirst: string;
    nameLast: string;

    getDisplayName(): string {
        return this.nameFirst + " " + this.nameLast;
    }
}
```

In this code snippet, we have declared a simple class named `Person` and added a few properties and a method. The syntax of defining a class begins through use of the TypeScript keyword `class` and then adding the appropriate members to the body of the class.

TypeScript supports the common `public`, `private`, and `protected` modifiers. The `public` modifier is the default. This differs from C# where classes and their members are considered `private` by default and developers must explicitly declare specific members as `public`. In the `Person` class, no modifiers were specified so the class and all its members have defaulted to `public`.

The syntax for declaring a constructor in a TypeScript class is through the use of a specially named function called `constructor()`. Constructors can specify any number of parameters as needed.

The `Person` class specifies two fields called `nameFirst` and `nameLast`. Consumers of the class can assign these fields to any valid string value. The values themselves are utilized in the `getDisplayName()` function to return a properly formatted full name. Note the use of the `this` keyword in the `getDisplayName()` function to properly access class-level members outside of the function itself. In C#, the `this` keyword is not always required if it can be inferred. TypeScript requires that developers specify `this` when accessing any other class level member.

Utilizing classes in C# should also be familiar to most developers as the `new` keyword is used, as in the following example:

```
var _person = new Person();

_person.nameFirst = "Michael";
_person.nameLast = "Jordan";

console.debug(_person.displayName());
```

This code simply instantiates a new instance of the `Person` class, assigns some values to its fields, and then executes a call to the `getDisplayName()` function to retrieve a formatted name.

TypeScript classes can expose data as fields (as implemented with `nameFirst` and `nameLast` in the example) or developers can specify their own `get` and `set` accessors to control access to data. This is the equivalent of exposing the data as a property in C#. The following example demonstrates the syntax for providing accessors to a specific class field.

```
private _description: string;

get description() {
    return this._description;
}

set description(value: string)
{
    this._description = value;
}
```

While this example of a property does not specify any business logic in either the `get` or `set` function, it does demonstrate the encapsulation of the data stored in the `private _description` field.

Inheritance

Now that we have reviewed the basic syntax for a simple TypeScript class, let's look at how inheritance is implemented with the following example:

```
abstract class BaseModel {

    constructor()
    {
    }

    id: string;
```

```

    protected logAction()
    {
    }
}

class Person extends BaseModel {

    constructor() {
        super();
    }

    nameFirst: string;
    nameLast: string;

    getDisplayName(): string {
        return this.id + ": " + this.nameFirst + " " + this.nameLast;
    }
}

```

The previous code implements a simple base class called `BaseModel`. The declaration of the `BaseModel` class does specify the optional `abstract` keyword to indicate that we intend for `BaseModel` to serve as a base class for others to derive from and not to ever be instantiated directly. This capability is identical to that found in C#. `BaseModel` provides an empty constructor and a few members.

The `Person` class then inherits functionality from `BaseClass` using the `extends` keyword. The only other addition to the `Person` class itself is the inclusion of a call to a method called `super()` inside the constructor. In TypeScript, if a base class implements a constructor then all derived classes must explicitly call their base classes constructor using the specially named `super()` function. If a derived class does not include a call to `super()` in its own constructor, the compiler will throw an error. If a base classes constructor includes parameters, then these parameters must be specified in the call to `super()` from the derived classes constructor.

One nice, yet somewhat confusing to some, feature of the TypeScript language is the ability to automatically assign constructor parameters. To demonstrate this concept, let's look at the following code:

```

class Person {
    private firstName: string;
    private lastName: string;

    constructor(firstName: string, lastName: string) {
        this.firstName = firstName;
        this.lastName = lastName;
    }
}

```

Note that in the code, the constructor of the `Person` class accepts two parameters and then, within the body of the constructor, the values of each of these parameters is assigned to a private field within `Person` to maintain the availability of these values through the lifetime of this instance. This pattern is extremely common in most languages (including C#) and is something most developers do frequently.

The designers of the TypeScript language decided to simplify this pattern and provide a more elegant and succinct syntax for developers to achieve the same goal. In TypeScript, the `Person` declaration can be simplified to the following:

```
class Person {
    constructor(private firstName: string, private lastName: string) {
    }
}
```

In the shorter version of the `Person` declaration, the constructor parameters are prefaced with either a `public`, `private`, or `protected` modifier. No other change is needed. If the TypeScript compiler detects a constructor parameter with one of these modifiers it will automatically generate a similarly modified field within the class and assign this field to the value of the parameter passed in. Essentially this makes the code generated from the second example look identical to the code generated in the first, but with much less work.

This is a subtle, but very useful, feature that at first looks a bit odd to developers new to TypeScript. You will see it used very heavily in most TypeScript code bases, especially with frameworks like Angular 2, which rely heavily on dependency injection via constructor parameters.

Interfaces

Interfaces are also very easy to utilize in TypeScript. The following code snippet demonstrates a simple interface with one member and a class that implements the interface:

```
interface IModel {
    id: string;
}

class Person implements IModel {
    id: string;
}
```

TypeScript uses the `implements` keyword to indicate that a class implements an interface. Like C# (and numerous other languages), TypeScript supports a single inheritance model where any class can only inherit functionality from a single base class but any class can implement as many interfaces as it wants.

Interfaces in TypeScript may seem very simple and consistent with other languages you may know but, since TypeScript ultimately compiles down to pure JavaScript, the concept of interfaces (including when and how they are enforced) is implemented slightly differently.

First, TypeScript supports a syntax for specifying interfaces directly as datatypes or method parameters. This concept can be seen in this code:

```
var _o: { nameFirst: string, nameLast: string };
_o = new Person();
```

Here, we see the declaration of a variable called `_o` and, for its datatype, we specify a simple interface. We essentially tell the compiler that `_o` can be assigned to any object that has two string properties called `nameFirst` and `nameLast`. We have not declared the interface prior to this line and its definition is just specified inline with the variable declaration. When we instantiate a new `Person` object and assign it to the variable `_o` the compiler then verifies that the new object meets the interface definition specified when `_o` was declared. This is a much different process than in C#, where interfaces are only considered to be implemented if the developer explicitly states they are implemented in the class declaration (or in the declaration of a base class). In TypeScript, simply having the same members as an interface is enough. The compiler will ensure you implement an interface when you try to assign a variable to a specific interface or pass a specific object to a method expecting a parameter with an interface type.

The syntax for declaring interfaces inline is available anywhere you can declare a datatype. We see this here, when accepting a parameter to a function:

```
interface IModel {
  id: string;
}
function logData(model: { id: string }) {
}
function logData2(model: IModel) {
}
```

You can see that the `logData()` and `logData2()` methods are essentially identical. Both accept any object that has (at least) a single object containing a property of type `string` and called `id`. The first method specifies this interface requirement in its signature when declaring the `model` parameter. The second method utilizes a specific interface (`IModel`) declared elsewhere as the datatype for its `model` parameter. Since these interfaces are identical, both method signatures are considered identical by TypeScript. As a developer, you are free to choose which interface definition syntax fits any specific scenario in your code.

Generics

The next concept worth reviewing when discussing TypeScript is the ability for developers to utilize generics in their TypeScript code. A full discussion of generics is beyond the intended scope of this chapter, but the following examples will provide a quick introduction as to the syntax and concepts. Developers familiar with utilizing generics in languages like `C#` should find the implementation of generics in TypeScript to be very familiar.

The following example of a `BaseService` class demonstrates the simplest use of generics when incorporating them into your own classes:

```
class BaseService<T> {
  searchItems(filter: string): T[] {
    return new Array<T>();
  }
  saveItem(item: T) : T
  {
    return item;
  }
}
```

The `BaseService` class specifies a single generic parameter named `T`. `T` can then be utilized throughout the class as a datatype placeholder, which will be specified at compile-time based on the usage when a generic class is instantiated. For example, the following code demonstrates how the generic class `BaseService` can be utilized as a type-specific way through inheritance.

```
class Person {
  nameFirst: string;
  nameLast: string;
}
class PersonService extends BaseService<Person> {
}
```


The `PersonService` class inherits from `BaseService` and specifies the generic type as `Person`. This means that, for developers utilizing `PersonService` in their code, the methods utilizing the previously “generic” type `T` will now be strongly typed to type `Person`. In this way, we can build classes that define a specific interface but treat their core datatypes interchangeably.

As with C#, TypeScript classes can specify multiple generic parameters at a class or method level. These parameters are separated by a comma. There are scenarios where a generic type cannot be 100% generic and some type of constraints need to be applied. This is achieved in TypeScript through another use of the `extends` keyword, as demonstrated in the following class declaration:

```
class BaseService<T extends BaseModel> {
}

```

In the implementation of `BaseService`, the generic parameter `T` can be assigned to any class that derives from `BaseModel`. This provides a large amount of flexibility to developers building generic implementations of core services, as they can now depend on the fact that the types they are provided have specific capabilities.

Like the implementation of generics in C#, there is more to the syntax and implementation than we can cover in this introductory chapter. It is important to note that TypeScript fully supports generics and most of the concepts and syntax you may be familiar with from C# have been brought into the TypeScript eco-system.

Compiler Settings

Like any other advanced language there are a significant number of configuration options you may want to tweak when developing a TypeScript application. The TypeScript compiler itself (a tool called `TSC.exe`) can accept these settings as parameters passed directly via the command line or, more commonly, development teams include a specially formatted JSON file called `tsconfig.json` in their TypeScript projects.

The `tsconfig.json` file has a very specific JSON structure and a defined set of options and configuration settings. The following JSON provides an example of a basic setup for a `tsconfig.json` file:

```
{
  "compilerOptions": {
    "noImplicitAny": false,
    "strictNullChecks": true,
    "noEmitOnError": true,
    "removeComments": false,
    "sourceMap": true,
    "target": "es5",
    "module": "commonjs",
    "outDir": "../output"
  },
  "compileOnSave": true,
  "exclude": [
    "node_modules",
    "wwwroot"
  ]
}
```

Some of the major options frequently configured for a TypeScript project include:

- The JavaScript platform to target with all generated code. This is set to ECMAScript 5 or es5 in the previous example.
- The directory in which to output all generated JavaScript code. The sample sets this to a relative path pointing at a directory called `output`. If this setting is not specified, all generated code will appear directly beside the input TypeScript files.
- Directories to explicitly include or exclude. The example does not specify any files to specifically include, so the assumption is to include every `*.ts` file in the same directory or lower than the `tsconfig` file itself. We do specify specific folders to exclude including the NPM packages folder (`node_modules`) and a folder called `wwwroot`.
- The module system to use when referencing components from one file in another file. Using modules with TypeScript is discussed in the next section.

A full list of compiler options available in the `tsconfig` file can be found on Microsoft's site: <https://www.typescriptlang.org/docs/handbook/compiler-options.html>.

Projects typically have a single `tsconfig.json` file at the root of the folder or project containing their TypeScript code. It is possible to have multiple `tsconfig` files and inherit specific settings from a root project.

Modules

Structuring a large TypeScript (or JavaScript) application can be tricky and there are a lot of different factors to consider. To follow industry best practices and take advantage of the OO concepts we described, it is highly recommended that development teams break their code into individual files per type or interface and, when code in one file depends on classes or other structures in another file, the dependency must be specified accordingly so that the compiler knows to include the other file.

Complex JavaScript projects have begun to depend heavily on external “module” loaders to help simplify these dependencies and to include required code efficiently and on demand. Two of the most popular JavaScript module loaders are SystemJS and CommonJS. As briefly discussed, the TypeScript compiler can be configured to easily output code to target either of these systems (in addition to several others).

TypeScript provides two primary keywords to support its integration with module loaders: `export` and `import`. Let's look at a brief scenario and discuss how these keywords simplify your usage of modules. First, imagine a project with a class called `Person` that is implemented in a file called `Person.ts` as the following:

```
export class Person {
  id: string;
  nameFirst: string;
  nameLast: string;
}
```

Note in this implementation of the `Person` class we have added the `export` keyword onto the class definition. This keyword tells TypeScript that we want this class exposed to the module system, as we want to use it from other files.

Given this `Person` implementation, we may decide to create a wrapper around a remote API to simplify our interaction with the server with regards to `Person` objects. This may turn into a class called `ApiService` that is implemented in its own file (`ApiService.ts`) that looks like this:

```
import { Person } from "./Person";

export class ApiService {
  getPeople(filter: string): Person[] {
    return new Array<Person>();
  }
}
```

In the `ApiService` class, we start with a line of code using the `import` keyword to indicate we are planning on using the previously exported class `Person`. There are multiple ways to form an `import` statement in TypeScript and the one specified here indicates we want to import a single class (i.e., `Person`) from the file `./Person`. Note that the filename is a relative path and does not specify an extension. The module loader will search for a file with either a JS or a TS extension. If we had not previously included the `export` keyword on the `Person` class, the compiler would give us an error explaining that the file we specified was not a module as it didn't export what we were looking for.

If we had not included the `import` line at the beginning of the `ApiService` class, the compiler would not recognize the `Person` class in our implementation of `ApiService` and we would get several errors.

Exporting classes appropriately and including the necessary `import` statements at the top of any TypeScript classes is simple enough and one of the most common steps in creating a new TypeScript class. The concept of explicitly stating our dependencies via `import` allows the TypeScript compiler and the specified module system with the information it needs to efficiently load files when requested and to keep track of everything accordingly.

There are also several other great features of the TypeScript module integration. First, if a relative path is not specified as an `import` the module loader will look for the module as a package loaded via NPM into the `node_modules` folder. This makes it very easy to load external packages via NPM and integrate them into your code with a simple `import`. Next, since every TypeScript file explicitly declares its dependencies via one or more `import` statements, module bundlers such as WebPack can be given one or more "entry points" into your application and can then intelligently determine every dependency for your application and package everything into an efficient set of files or bundles for deployment.

In the upcoming chapters of this book, you will see this concept implemented frequently as we use the concept of TypeScript exports and imports to build applications using multiple frameworks (Angular 2 and React), all while using TypeScript and module loaders to integrate the components we need and to specify the relationships between our own components/files.

Implementing a Basic TypeScript Application

The best way to get familiar with the TypeScript language is to dive in and see how easy it is to add TypeScript and debug the resulting code. The example we use in this chapter is very simple and does not take advantage of a rich client side framework such as Angular, as we want to focus solely on TypeScript. The remaining chapters of this book will use TypeScript in much more advanced scenarios.

If you are using Visual Studio 2017 for your development you already have everything you need to incorporate TypeScript into your applications. If you are utilizing an older IDE such as Visual Studio 2015, you may have to install TypeScript separately. With Visual Studio this is done via the Tools > Extensions menu. Many build processes utilize a package manager such as NPM to install a specific version of TypeScript and then perform all compilation as part of a build process using tools such as Gulp or WebPack. Some of these concepts are discussed in Chapter 6.

Setting Up a Sample Project

Now let's create a blank web project to play around with TypeScript. Start by launching Visual Studio and selecting **File** ► **New** ► **Project**. In the left rail, select **Web** under **Visual C#**, select **ASP.NET Core Web Application**, and change the Name to `TypeScriptSample`, as shown in Figure 7-1. Click **OK**.

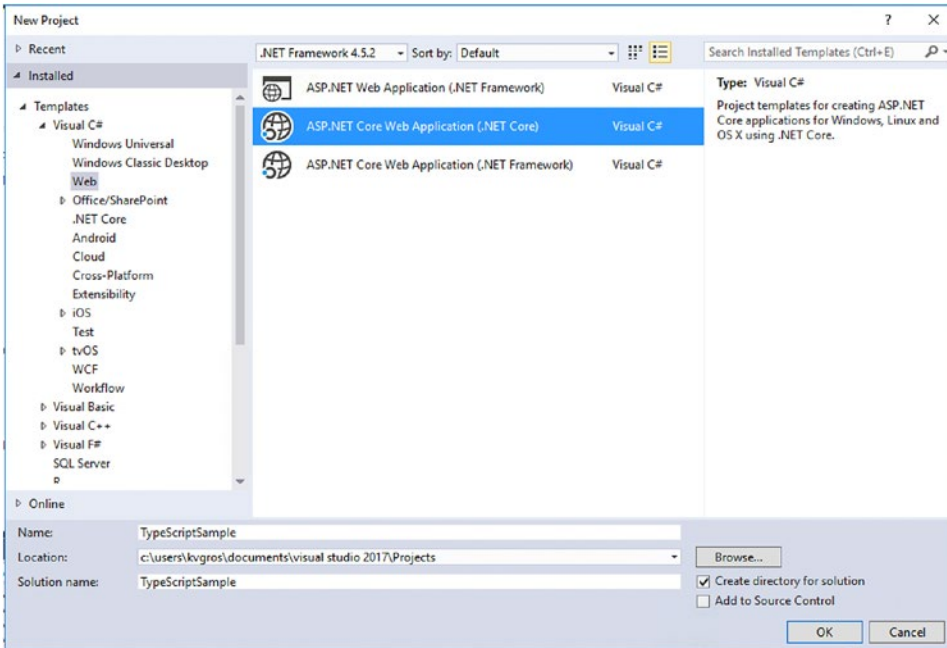


Figure 7-1. Creating a new project

On the next screen, select **Empty** under **ASP.NET Core Templates** and click on **OK**, as shown in Figure 7-2.

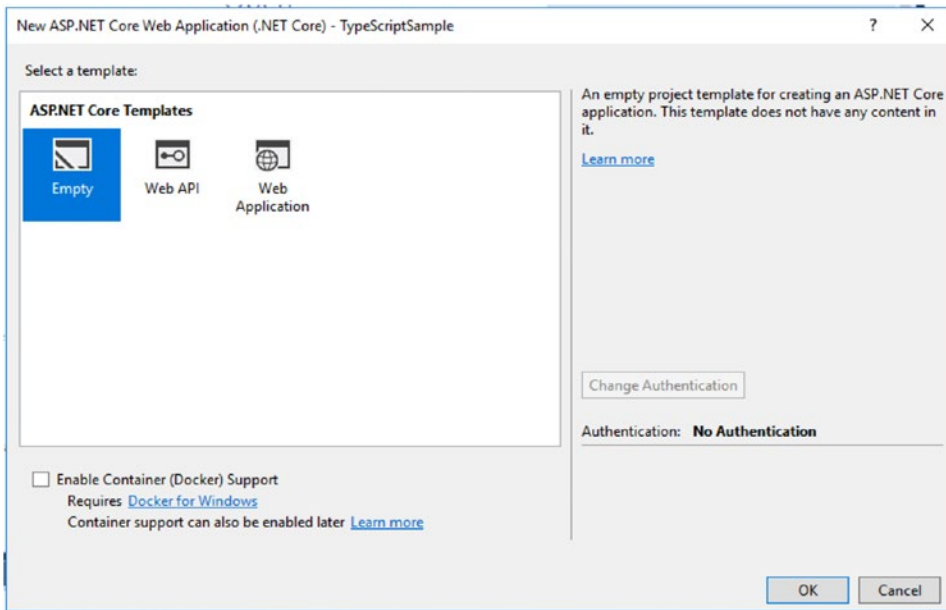


Figure 7-2. Template selection

Now you will have an empty ASP.NET Core project. This chapter will not dive into ASP.NET Core or the empty project structure right now. Those details are discussed in the WebAPI and MVC chapters earlier in the book. The project will need to be configured to display static files so it can run the TypeScript samples.

To configure the project to display static files, we need to add some middleware to the `Startup.cs` file. To open this file, expand the `src` ▶ `TypeScriptSample` area in the Solution Explorer. Now double-click on the `Startup.cs` file. By default, it will look like this.

```
public class Startup
{
public void ConfigureServices(IServiceCollection services)
    {
    }

public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory
loggerFactory)
{
    loggerFactory.AddConsole();

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

    app.Run(async (context) =>
    {
        await context.Response.WriteAsync("Hello World!");
    });
}
}
```

Notice this code just writes "Hello World!" to the page. Not very useful. In order to use static HTML files, we will need to install the Microsoft.AspNet.StaticFiles NuGet package. Right-click on the TypeScriptSample project in the Solution Explorer and click Manage NuGet Packages... Next select Browse and type `AspNet.StaticFiles` in the search box, as shown in Figure 7-3.

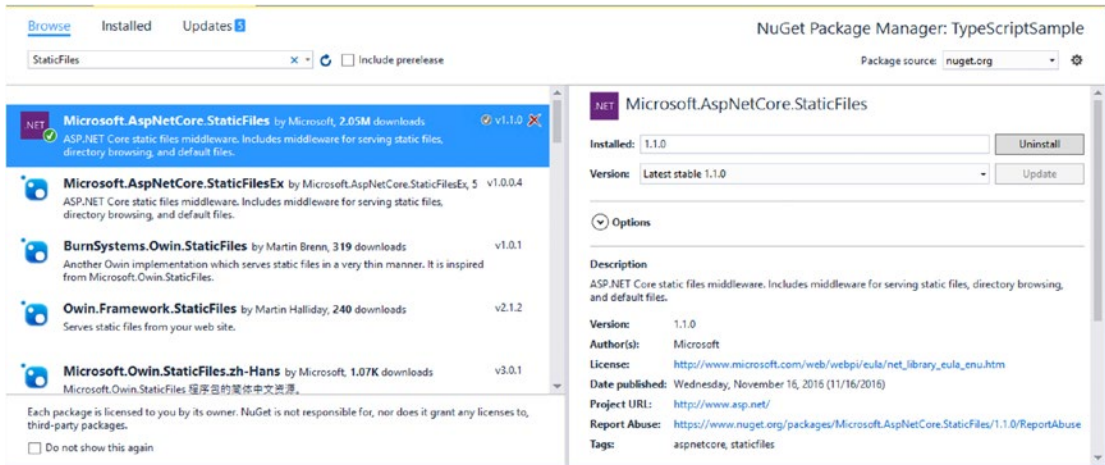


Figure 7-3. The NuGet Package Manager

Select `Microsoft.AspNet.Static files` and install them (accepting any dialogs that pop up). Now modify the code in `Startup.cs` to use default and static files. Remove the Hello World code and replace it with `app.UseDefaultFiles` and `app.UseStaticFiles`, as shown in bold:

```
public void Configure(IApplicationBuilder app, IHostingEnvironment env, ILoggerFactory
loggerFactory)
{
    loggerFactory.AddConsole();

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

    app.UseStaticFiles();
    app.UseDefaultFiles();
}
}
```

Now we will set up the structure of the project in the `wwwroot` folder. First, we add a static HTML file to project by right-clicking on the `wwwroot` folder and clicking Add ► New Item. In the left rail, select Web under ASP.NET Core, select HTML Page, and change the name to `Index`, as shown in Figure 7-4.

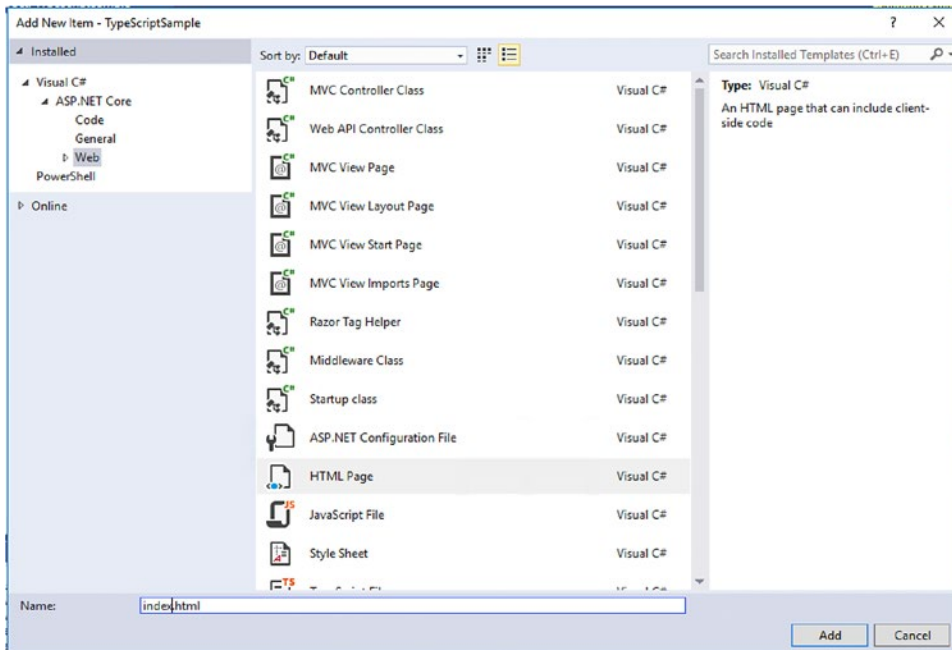


Figure 7-4. Adding a new HTML page

Now create a folder to put the JavaScript that is output from the TypeScript files. Add that by right-clicking on the `wwwroot` folder and selecting **Add ► New Folder**. Name the new folder `js`. We will configure our TypeScript compiler to generate all JavaScript into this folder.

Working with TypeScript Files

Since TypeScript files are compiled to JavaScript files, they are not typically deployed with the site. Instead the JavaScript that is output from compiling (or, more accurately, transpiling) the `.ts` files will be deployed with the site. When debugging locally, source mapping is generated that map the `.js` files back to the `.ts` files. This allows modern browsers such as Google Chrome to properly debug TypeScript.

In the sample project's case, we will maintain all our TypeScript files outside of the `wwwroot` folder in a folder called `src` that is in the root of the project. To create this folder, right-click on the TypeScriptSample project and select **Add ► New Folder**. Name the folder `src`. Make sure this folder is not in the `wwwroot` folder and will not be deployed with the site.

The `src` folder is where all the TypeScript files will be added. The `wwwroot\js` folder is where the compiled JavaScript files will be located. So how does the compiler know where the TypeScript files are and what folder to compile the JavaScript to? As described, we will specify this in the `tsconfig.json` file. Add this file to the project by right-clicking on the `src` folder and clicking **Add ► New Item**. In the left rail, select **Web** under **ASP.NET Core**, select **TypeScript JSON Configuration File**, and leave the name as `tsconfig.json`, as shown in Figure 7-5.

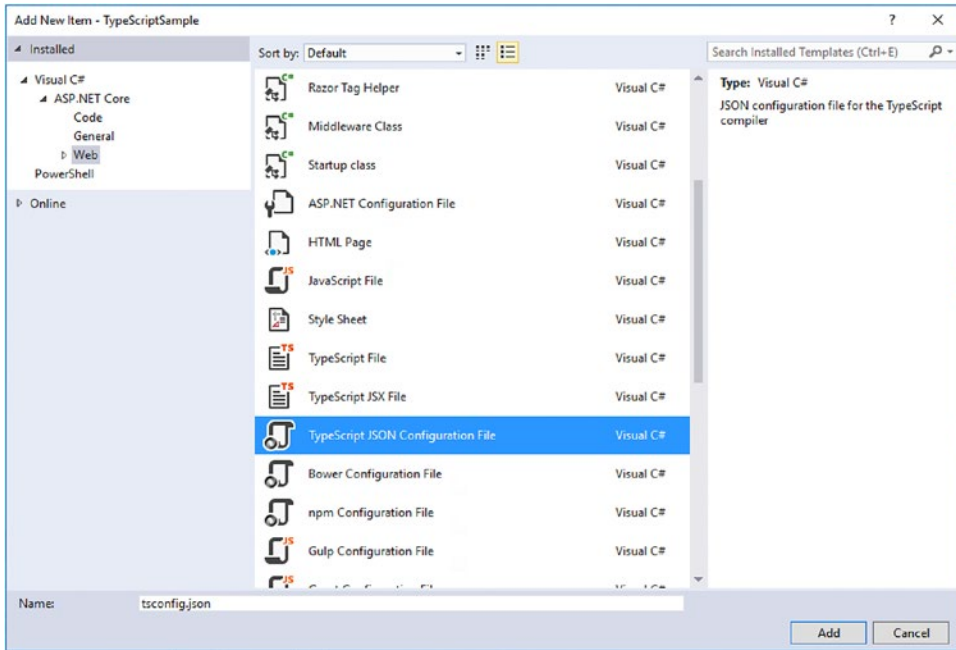


Figure 7-5. Adding a new `tsconfig.json` file

This JSON file contains the compiler options for TypeScript. The basics of this file were discussed early and there are many options not covered in this chapter. For a complete list of options, browse to <http://www.typescriptlang.org/docs/handbook/tsconfig.json.html>. Here is what the file looks like by default:

```
{
  "compilerOptions": {
    "noImplicitAny": false,
    "noEmitOnError": true,
    "removeComments": false,
    "sourceMap": true,
    "target": "es5"
  },
  "exclude": [
    "node_modules",
    "wwwroot"
  ]
}
```

Most of these defaults are all valid for the sample project. It does need a couple more options. Table 7-1 lists some of the options that we need to configure for this sample.

Table 7-1. TypeScript Compiler Options

Option	Meaning
compileOnSave	Tells the IDE to generate the JavaScript file when a TypeScript file is saved.
inlineSourceMap	Include the TypeScript source map with the JavaScript files. For debugging only.
inlineSources	Create one JavaScript file with both the TypeScript and the JavaScript. For debugging only.
outDir	The directory where the compiled JavaScript files are saved.
module	The module loading system to expect when generating JavaScript. For our example, we will use “system” to specify SystemJS.

For purposes of this example, let’s replace the default `tsconfig.json` file with the settings configured in this sample:

```
{
  "compileOnSave": true,
  "compilerOptions": {
    "noImplicitAny": false,
    "noEmitOnError": true,
    "removeComments": false,
    "inlineSourceMap": true,
    "inlineSources": true,
    "module": "system",
    "moduleResolution": "Node",
    "target": "es5",
    "outDir": "../wwwroot/js"
  },
  "exclude": [
    "node_modules",
    "wwwroot"
  ]
}
```

Notice it is not listing all the TypeScript files or providing the directory of the TypeScript files. By default, the directory containing the `tsconfig.json` file is considered the root directory for our TypeScript files. Since we added the `tsconfig.json` file to our `src` directory, that is considered the root directory.

Finally, for purposes of this example we specify that the TypeScript compiler should utilize the SystemJS module loader. We will see how this can be used for simple applications when we add our bootstrap code to the `Index.html` file.

NPM Packages

To demonstrate a common step in any modern web development project, we are going to utilize Node’s Package Manager (NPM) to install some items into our project. This concept will be used heavily in later chapters to bring in our client side frameworks. For this example, we will use NPM to install the current TypeScript compiler and some TypeScript “typings” for iQuery, which we will use in our examples. The concept behind “typings” will be discussed when we take advantage of them and start utilizing jQuery.

Visual Studio 2017 has native support for NPM and we can begin to use NPM in our project by right-clicking on the project and selecting Add ► New Item. In the left rail, select Web under ASP.NET Core, select NPM Configuration File, and leave the name as package.json, as shown in Figure 7-6.

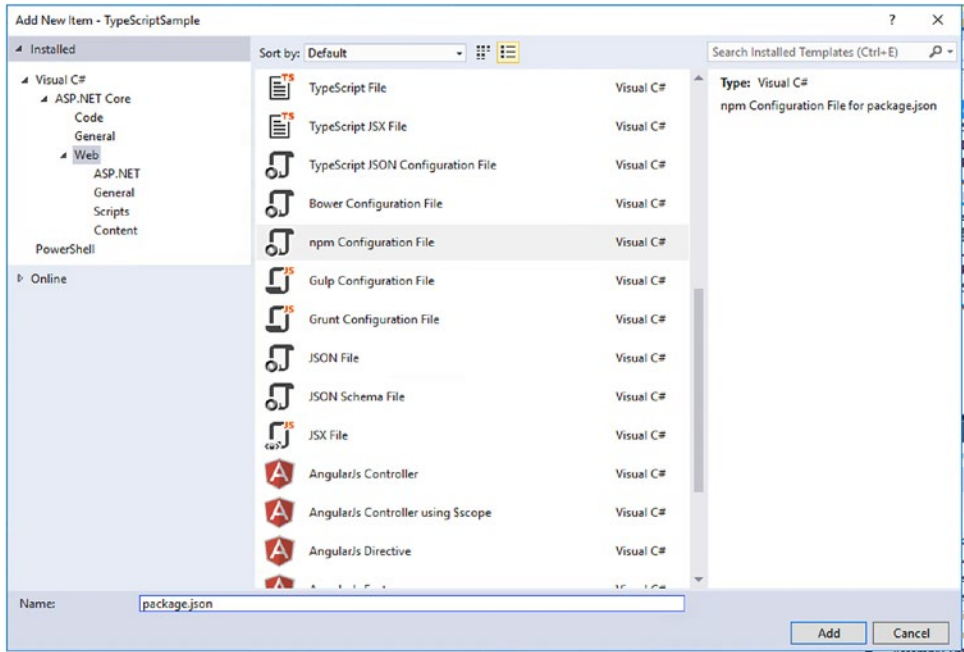


Figure 7-6. Adding a new NPM configuration file

Once the package.json has been added to the root of your project, you can use Visual Studio's editor to modify the default file contents to include two devDependencies by adding the lines highlighted in bold:

```
{
  "version": "1.0.0",
  "name": "asp.net",
  "private": true,
  "dependencies": {
  },
  "devDependencies": {
    "typescript": "2.1.4",
    "@types/jquery": "1.10.31"
  }
}
```

As mentioned, the full power of NPM will be described and demonstrated in other chapters in this book but, for now, we are simply utilizing it to ensure that TypeScript and typings for jQuery are added to our project.

Once we have modified the file to include these lines (and saved), Visual Studio will automatically detect the changes and execute NPM behind the scenes to ensure the specified packages are installed in your project. If you are new to NPM, you will find all installed packages in the root of your project in a folder

called `node_modules`. This folder is automatically hidden in your Visual Studio solution explorer so, to see it, you must navigate into your project folder via Windows Explorer. Visual Studio does provide a nice extension to its Solution Explorer to let developers know visually what packages are currently installed. This can be seen under the Dependencies node, as shown in Figure 7-7.

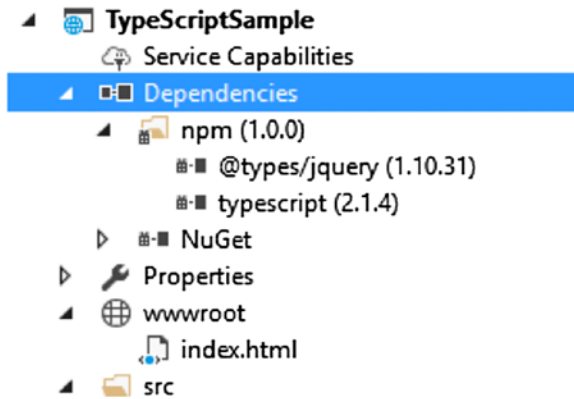


Figure 7-7. Solution Explorer Dependencies node

Note that the Dependencies node in Solution Explorer is used to show dependencies installed via NPM and those installed via NuGet.

Adding TypeScript

At this point we have an empty project set up with some plumbing in place in preparation of our TypeScript demonstration. At this point, these samples are going to keep it simple to demonstrate the syntax of TypeScript without needing a ton of explanation of the problem that is being solved. They will not connect to any APIs or save data to a data source. Later chapters in this book demonstrate the usage of the TypeScript language in more advanced scenarios as we evaluate the more robust client side frameworks.

The premise of the following samples is to provide a list of products and prices. The prices are determined by what type of customer is logged in. The customer types are Anonymous, Bronze, Silver, and Gold. This simple structure will be used to explain the benefits of TypeScript.

To begin to see TypeScript in action, right-click on the `src` folder in the `TypeScriptSample` project and click **Add** ► **New Item**. In the left rail, select **Web** under **Web**, select **TypeScript File**, and name it `ICustomer.ts`, as shown in Figure 7-8.

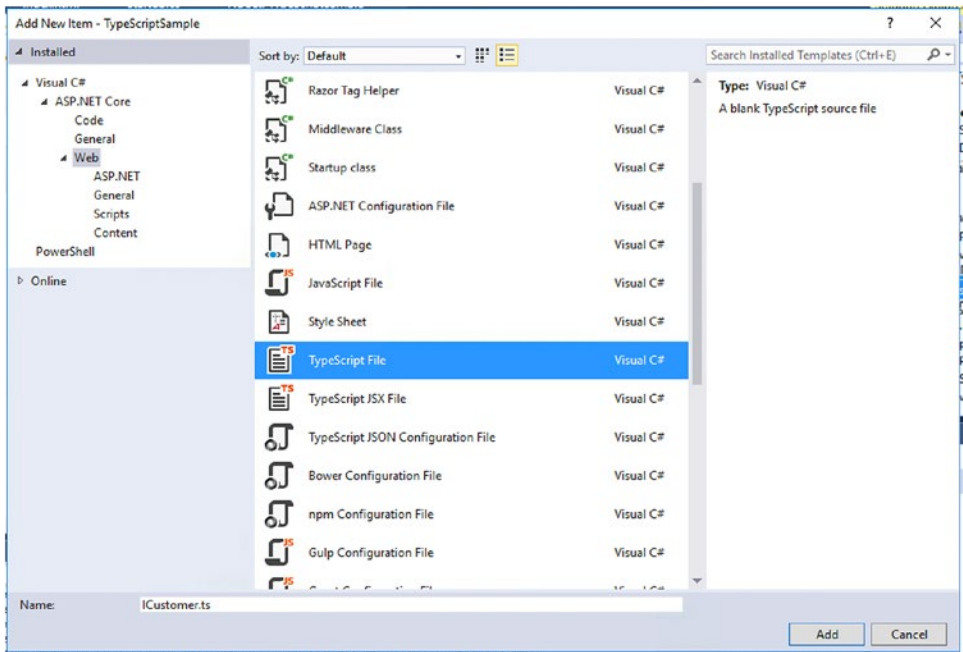


Figure 7-8. Adding a TypeScript file

Within the new file, we will create our first TypeScript interface with the following code:

```
export interface ICustomer {
    name: string;
    numberOfYearsCustomer: number;
    discountPercent(): number;
}
```

Note the use of the `export` keyword in the `ICustomer` implementation. This indicates that we will be using this interface in other files and that the module loader should prepare it accordingly.

The `ICustomer` interface declares two variables and one method. One `string` variable, one `number` variable, and one method that returns a `number`. As mentioned, when discussing datatypes, simply adding “: `string`” to the name variable tells the compiler that it is a `string`. Now if we try to assign a `number` to the name variable, we will get a compilation error. There will be an example of that in the next section.

What does the JavaScript look for this file once it is compiled? Well, remember that TypeScript defines a developer’s intent. JavaScript does not provide the concept of an interface. So, once you save this file you will see a `ICustomer.js` file in the `TypeScriptSample\wwwroot\js` folder. If you open this file, you will see that it is empty. TypeScript will use the `ICustomer` interface for compiling other parts of the TypeScript, but there is no JavaScript to send to the client.

Classes and Inheritance

Now that the `ICustomer` interface is complete, there needs to be one or more classes to implement it. For our example, we are going to create a `BaseCustomer` class and then four concrete implementations of the `ICustomer` interface that inherit base functionality from `BaseCustomer`.

To do this, right-click on the `src` folder in the `TypeScriptSample` project and click **Add ► New Item**. In the left rail, select **Web** under **ASP.NET Core**, select **TypeScript File**, and name it `CustomerBase.ts`.

Add the following abstract class to the `CustomerBase.ts`.

```
import { ICustomer } from "./ICustomer";

export abstract class CustomerBase implements ICustomer {
    name = "Customer Base";
    numberOfYearsCustomer = 0;
    discountPercent() {
        return .01 * this.numberOfYearsCustomer;
    }
}
```

This class implements the `ICustomer` Interface. It also uses the `abstract` keyword to indicate that it cannot be instantiated directly. It is again important to note that this is only defining the coder's intent. The JavaScript that is output from the compiler will not be abstract at runtime. It is for compilation purposes only.

Another key piece of the `CustomerBase` class is the use of the `import` keyword to indicate that it requires use of the `ICustomer` interface defined in another file.

Once this file is saved, there is some JavaScript to look at (finally!). Open the `TypeScriptSample\wwwroot\js\CustomerBase.js` file. It should look like this:

```
"use strict";
var CustomerBase = (function () {
    function CustomerBase() {
        this.name = "Customer Base";
        this.numberOfYearsCustomer = 0;
    }
    CustomerBase.prototype.discountPercent = function () {
        return .01 * this.numberOfYearsCustomer;
    };
    return CustomerBase;
})();
```

This code might look foreign to you if you are not familiar with JavaScript. It is using the constructor, prototype, module and revealing module JavaScript patterns. This code is readable and can be followed, but the TypeScript code is much easier for a most developers to read and write. TypeScript hides some of the complexities that are involved in writing good, maintainable JavaScript.

■ **Note** Do not modify this JavaScript code directly. Like all code generation\compilation tools, it will overwrite this file once the TypeScript file is saved again. Any changes made to the JavaScript file will be lost.

When taking a closer look at this code, you can see that it does not contain any of the type annotations nor references to the `ICustomer` interface. In fact, it is also not validating or enforcing those types. This proves that many of the features of TypeScript are only important for compilation.

Now there is a customer base class, we will continue by building four concrete customer implementations. Right-click on the `src` folder in the `TypeScriptSample` project and click **Add ► New Item**. In the left rail, select **Web** under **ASP.NET Core**, select **TypeScript File**, and name it `CustomerAnonymous.ts`, as shown in Figure 7-9.

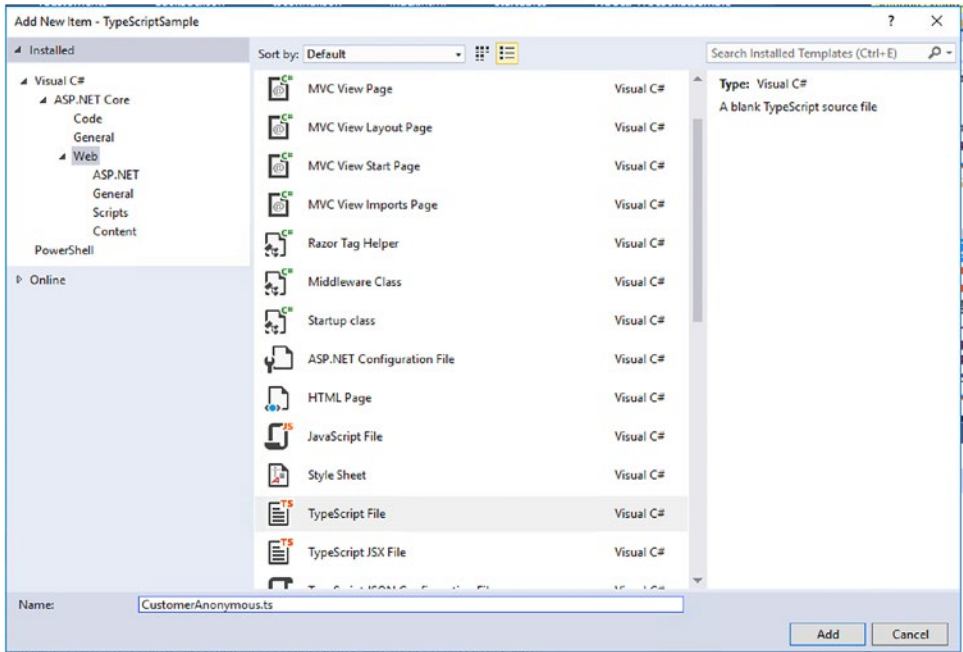


Figure 7-9. Adding a new *CustomerAnonymous.ts* file

Add the following abstract class to *CustomerAnonymous.ts*.

```
import { CustomerBase } from "./CustomerBase";

export class CustomerAnonymous extends CustomerBase {
    name = "Valued Customer";
}
```

This class simply extends the customer base class and changes the name value. If the sample was retrieving a customer from an API call, it might instantiate an object of this class and set the name based on the customer it retrieved. There is now a customer interface, customer base class, and an anonymous customer class. The sample project will also need to include *CustomerBronze.ts*, *CustomerSilver.ts*, and *CustomerGold.ts* files. Create these files in the *src* directory the same way you created the *CustomerAnonymous.ts* file earlier. The contents of each file are shown here:

CustomerBronze.ts

```
import { CustomerBase } from "./CustomerBase";

export class CustomerBronze extends CustomerBase {
    name = "Bronze Customer";
    numberOfYearsCustomer = 5;
}
```

CustomerSilver.ts

```
import { CustomerBase } from "./CustomerBase";

export class CustomerSilver extends CustomerBase {
  name = "Silver Customer";
  numberOfYearsCustomer = 10;
}
```

CustomerGold.ts

```
import { CustomerBase } from "./CustomerBase";

export class CustomerGold extends CustomerBase {
  name = "Gold Customer";
  numberOfYearsCustomer = 15;
  discountPercent() {
    return .20;
  }
}
```

The bronze and silver customers are simply extending the base customer and resetting some of the base properties. Nothing ground shattering. The gold customer is resetting the properties, but it is also overriding the `discountPercent` method. The base customer class calculates the discount percentage based on the number of years the customer has been a customer. Here is what that function looks like:

```
discountPercent() {
  return .01 * this.numberOfYearsCustomer;
}
```

Gold customers will get a 20 percent discount no matter how long they have been a customer. This is very common among object oriented coding patterns and provides for the concept of polymorphism. The behavior of the customer gold class is different from all the other customers.

Note that each of the concrete customer types (Anonymous, Bronze, Silver, and Gold) import the base class (`CustomerBase`) and inherit its core functionality. Each of them also implements the `ICustomer` interface through this base class, although they do not specify it explicitly.

The sample project should now look something like [Figure 7-10](#).

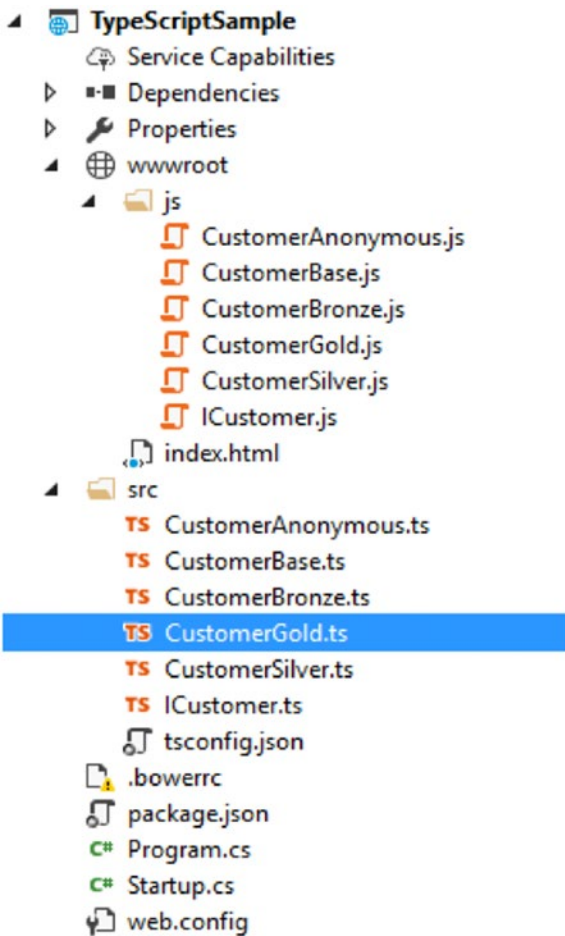


Figure 7-10. Current solution

Adding a Product List

The next step in hooking up our TypeScript sample is to add a few classes related to managing a list of products and then to hook everything up via a View Model class. Once we have all the TypeScript in place we will add code to our `Index.html` file to bootstrap everything and then debug through the process of getting everything running.

Let's start by adding a TypeScript file that will represent our "View Model" or the core code to be utilized to manage our user interface. To do this, right-click on the `src` folder in the `TypeScriptSample` project and click `Add > New Item`. In the left rail, select `Web` under `ASP.NET Core`, select `TypeScript File`, and name it `IndexVM.ts`, as shown in [Figure 7-11](#).

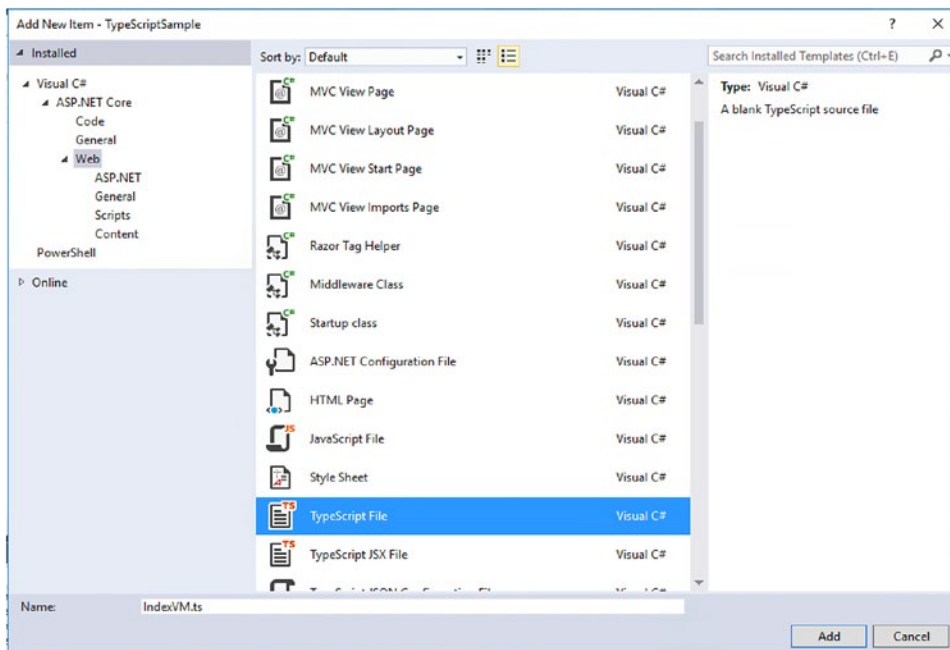


Figure 7-11. Adding a new *IndexVM.ts* file

Now, create the `indexVM` class inside of the `src` folder. In this file, we need to instantiate a global variable of it so we can access it from the page:

```
export class IndexVm {
}

export let indexVmInstance = new IndexVm();
```

The `IndexVM.ts` file will need to run some logic to instantiate the customer classes when the page loads. It will also need to access some of the HTML elements on the page and set their values. What better way to do that than to use jQuery? jQuery has fallen out of favor somewhat but it is still a great way to do some simple manipulation of the page. jQuery is a highly adopted third-party JavaScript library that has no direct relationship to the TypeScript language. However, like all JavaScript files, it can be added to the page via a script tag. This will make it available to all subsequent JavaScript code. So how do we use it, or any third-party JavaScript framework, with TypeScript and not get compilation errors? There are several ways this can be accomplished and, if you have been paying attention, we've already laid some of the groundwork for the use of jQuery in our project.

First, to use a third-party library like jQuery we can simply load the library from either a local or a remote (i.e. CDN) source and we can use it directly from our TypeScript. With jQuery, we know that the `$` symbol serves as the root of all functionality. To allow us to use the jQuery `$` without causing TypeScript compile errors, we need to include what is called an *ambient declaration*. If we add the following line of code to the top of our file (or somewhere else that is always loaded with our project):

```
declare var $: any;
```

Using the TypeScript `declare` keyword we essentially tell the TypeScript compiler that we want to use the `$` symbol and that it should be typed as `any`. This allows us to use the `$` and forces the TypeScript compiler to skip any type checking when we access properties or functions on this symbol. We do not get any IDE IntelliSense or compile-time validation on our usage of the `$` using this pattern. If we accidentally have a typo or otherwise use it incorrectly we will not know until the code executes and the browser logs an error.

Using the `declare` keyword we can globally declare any keyword to be available and safe for us to use. This is a quick and easy way to use third-party JavaScript libraries when we know the name of their global classes.

What do we do if we want to use a library like jQuery and enjoy the type safety and IntelliSense provided by TypeScript? For this we need to either find a TypeScript “typing” file or write our own. In the ambient declaration line, we specified that `$` should be treated as type `any`. If we had a TypeScript interface (or set of interfaces) that defined all the datatypes and method signatures for the `$` (as implemented in jQuery’s own JavaScript file), we could have used that and the TypeScript compiler would ensure that we use the `$` correctly and that all calls are passed on to the actual jQuery implementation. This would be the best of all worlds. In the TypeScript world, this solution is achieved using “typing” files or libraries. To demonstrate this concept let’s look at what we have already set up for jQuery.

First, when we configured NPM to load packages into our project, one of the two packages we specified in the `package.json` was the following:

```
"@types/jquery": "1.10.31"
```

This package contains a library of community-built TypeScript interfaces to match the signatures of the jQuery library. The file loaded in this package are automatically downloaded into the project in the `node_modules/@types/jquery` folder. TypeScript (and most IDEs) automatically reference type definitions found under the `node_modules/@types` folder. Using the `tsconfig.json` file, you can adjust this setting to explicitly call out typings to include or to exclude. In our sample, we do not change any settings and we accept the default behavior of including everything.

The standard naming convention for TypeScript “typing” files is to use a `*.d.ts` extension. This helps differentiate files that contain no more than type interfaces for third-party JavaScript libraries from those containing actual app code. If you were to use Windows Explorer to browse to the `node_modules/@types/jquery` folder, you would find all jQuery interfaces included in a single file called `jquery.d.ts`.

Now that we have ensured our `package.json` includes the jQuery types, we can check out the benefits of having these files. If you open the `IndexVM.ts` file and enter a `$` near the end of the file, you will see a very robust IntelliSense implementation that dictates what capabilities are included with jQuery. This can be seen in Figure 7-12.

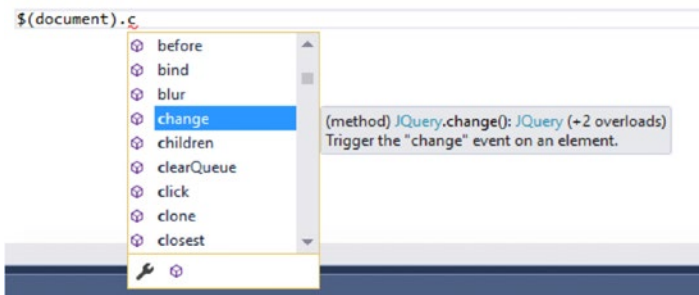


Figure 7-12. IntelliSense with jQuery typings

If you would like to see the sheer number of TypeScript typing files for third-party JavaScript libraries, you can check them out at the site <http://definitelytyped.org/>. The community has started a public GitHub repo that currently provides typing definitions for over 1,700 JavaScript libraries. This makes it very easy to include them in your TypeScript projects and enjoy all the type safety and information they provide.

It is now time to begin building a simple UI to execute the TypeScript. Open the `Index.html` page located in the `wwwroot` folder of the `TypeScriptSample` project. The default content for this file contains some basic HTML like the following:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title></title>
</head>
<body>

</body>
</html>
```

First, we need to reference the jQuery since it will be used by the sample application. We will reference this using a Content Delivery Network (CDN). The CDN takes the place of downloading and providing the jQuery files inside of the sample project. It is best practice to reference scripts at the bottom of the body tag for performance reasons. Add a reference to the jQuery CDN to the body tag, shown in the following bolded code snippet.

```
<body>

  <script src="https://code.jquery.com/jquery-2.2.2.min.js" integrity="sha256-36cp2Co+
+62rEAAYHLmRCPIych47CvdM+uTBJwSzWjI=" crossorigin="anonymous"></script>

</body>
```

Generally, you would also reference each of the individual JavaScript files you have in your own project to this same area of the HTML. In our sample, we are going to focus on the SystemJS module loader and how we can use it to simplify loading complex TypeScript projects. For this we need to include a reference to the `system.js` library itself and we will again use a CDN. This can be achieved by adding the script tag highlighted in bold:

```
<body>

  <script src="https://cdnjs.cloudflare.com/ajax/libs/systemjs/0.19.41/system.js">
  </script>
  <script src="https://code.jquery.com/jquery-2.2.2.min.js"
  integrity="sha256-36cp2Co+62rEAAYHLmRCPIych47CvdM+uTBJwSzWjI="
  crossorigin="anonymous"></script>

</body>
```

■ **Note** The order of these `script` tags is important. JavaScript is an interpreted language and will be load sequentially. When a `.js` file references a class, variable, or function it must already be defined by a previous `.js` file.

Next we need to add a few lines of code to configure SystemJS and to tell it to open the “entry point” of our application. The entry point for our sample will be the `indexVM` file. The setup for SystemJS is specified in this code snippet:

```
<body>

  <script src="https://cdnjs.cloudflare.com/ajax/libs/systemjs/0.19.41/system.js">
  </script>
  <script src="https://code.jquery.com/jquery-2.2.2.min.js"
    integrity="sha256-36cp2Co+/62rEAAAYHLMRCPIych47CvdM+uTBjwSzwjI="
    crossorigin="anonymous"></script>
  <script type="text/javascript">
    System.config({
      defaultJSExtensions: true
    });
    System.import('./js/indexvm.js');
  </script>
</body>
```

The `System` class is the root of SystemJS and, on the first line we configure the module loader to look for modules with the extension of `*.js`. Remember our TypeScript is compiled at compile time and saved to the `wwwroot/js` folder. Each file generated by the TypeScript transpiler will have the same name as the source TypeScript file but with a JS extension. This line ensures that our modules are found accordingly (since the `import` statements in our TypeScript files do not specify an extension).

The next line of code calls `System.import` and specifies the file that will be the entry point to the application: `indexvm.js`. As discussed earlier, each TypeScript file we have written will ultimately declare its own dependencies via the `import` statements (usually at the top of the file). SystemJS can use these dependency declarations to load any required files on-demand. For this reason, we are not required to add `script` tags for any of our individual TypeScript/JavaScript files (including the `indexVM`) and instead we tell SystemJS to start with our entry point and let everything else load on demand. In more robust systems and build processes such as those that use WebPack, we will have different module and bundling considerations but, for this sample, we just kick off `indexVM.js` and everything else gets loaded accordingly. As more files are added and our complexity grows, we may never need to modify the initial scripts that bootstrap/load our application.

Now that our `script` tags are set up correctly and our SystemJS configuration has been added, we can add a few more tags to the `index.html` file. Add some HTML elements to the page that will hold the customer’s name, shown in the following bolded code snippet, some buttons to change the customer type, and a placeholder we will use to display a list of products. These elements are shown here in bold:

```
<body>
  <div>
    <button id="loginBronze">Login as Bronze</button>
    <button id="loginSilver">Login as Silver</button>
    <button id="loginGold">Login as Gold</button>
  </div>
```

```

<h2>Product List</h2>
<h4 id="customerName">Welcome</h4>

<table id="productsTable"></table>

<script src="https://cdnjs.cloudflare.com/ajax/libs/systemjs/0.19.41/system.js">
</script>
<script src="https://code.jquery.com/jquery-2.2.2.min.js" integrity="sha256-36cp2Co+/62r
EAAyHLMRCPIych47CvdM+uTBjwSzwjI=" crossorigin="anonymous"></script>
<script type="text/javascript">
    System.config({
        defaultJSExtensions: true
    });
    System.import('./js/indexvm.js');
</script>

</body>

```

With the addition of the base `Index.html` implementation, we can now start adding code to our `src/IndexVM.ts` file and begin to see the results of the sample project thus far.

The first step in adding code to our `IndexVM` file will be to include the necessary `import` statements that will allow us to use the classes and interfaces we have already implemented in other files. You can do this by adding the following code to the top of the `IndexVM.ts` file:

```

import { CustomerBase } from "./CustomerBase";
import { ICustomer } from "./ICustomer";
import { CustomerBronze } from "./CustomerBronze";
import { CustomerGold } from "./CustomerGold";
import { CustomerSilver } from "./CustomerSilver";
import { CustomerAnonymous } from "./CustomerAnonymous";

```

Having these imports at the top of the file tells the TypeScript compiler where to find the implementation details for any of the specified classes or interfaces we may use in this file. They also tell SystemJS that code in this file depends on exported capabilities in those other files so that SystemJS knows to load them.

Let's continue by adding some code to our `IndexVM` to store a private instance of the current user and a method to use jQuery to display the current user information on the screen.

```

export class IndexVm {

    private currentCustomer: ICustomer = new CustomerAnonymous();

    setWelcomeMsg() {
        var msg = `Welcome ${this.currentCustomer.name}`;
        $("#customerName")[0].innerText = msg;
    }
}

```

Notice that the type for the `currentCustomer` variable is `ICustomer`. This allows the application to set this variable to any of the customers that implement this interface. Later the sample will use that to switch between the anonymous, bronze, silver, and gold customers.

The final step is call the `setWelcomeMsg` method when the page is loaded, shown in the bolded code. This code should be at the end of the current `IndexVm` file (but not inside the `IndexVm` class).

```
export let indexVmInstance = new IndexVm();

$(document).ready(() => {
  indexVmInstance.setWelcomeMsg();
});
```

Now we are ready to run the application and see a basic version of the code running. Press F5 to run the application. It should look like Figure 7-13.

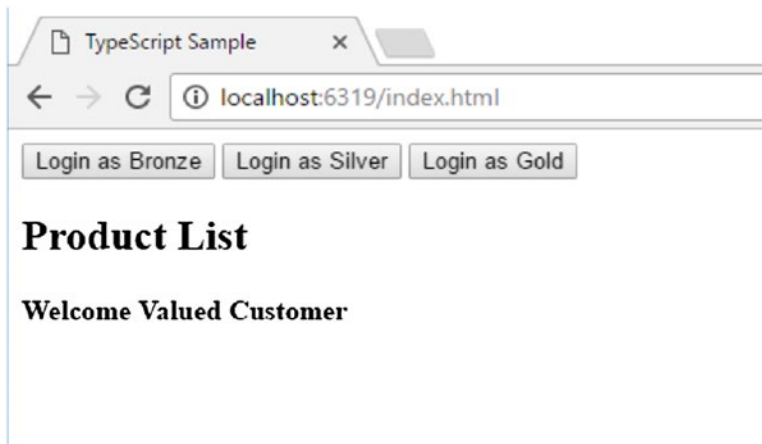


Figure 7-13. Initial test of the application

Pretty exciting! The application is displaying the name of the anonymous customer. That is the default value of the `currentCustomer` variable in the `IndexVm` class.

Debugging TypeScript

Debugging TypeScript can be done using any of the developer tools provided by modern browsers. Remember earlier in the chapter we set up the `tsconfig.json` file to include the source maps with the `.js` files. TypeScript debugging is possible because of these source maps. This section will use the Chrome browser to demonstrate debugging. You can use whichever browser you prefer if it has developer tools that support TypeScript debugging.

First, press F5 inside of Visual Studio to run the application. Once the application opens in the browser, press F12 to launch the developer tools. Next click the Sources tab in the top menu of the developer tools. Notice that both the `js` and the `src` folders are present, shown in Figure 7-14.

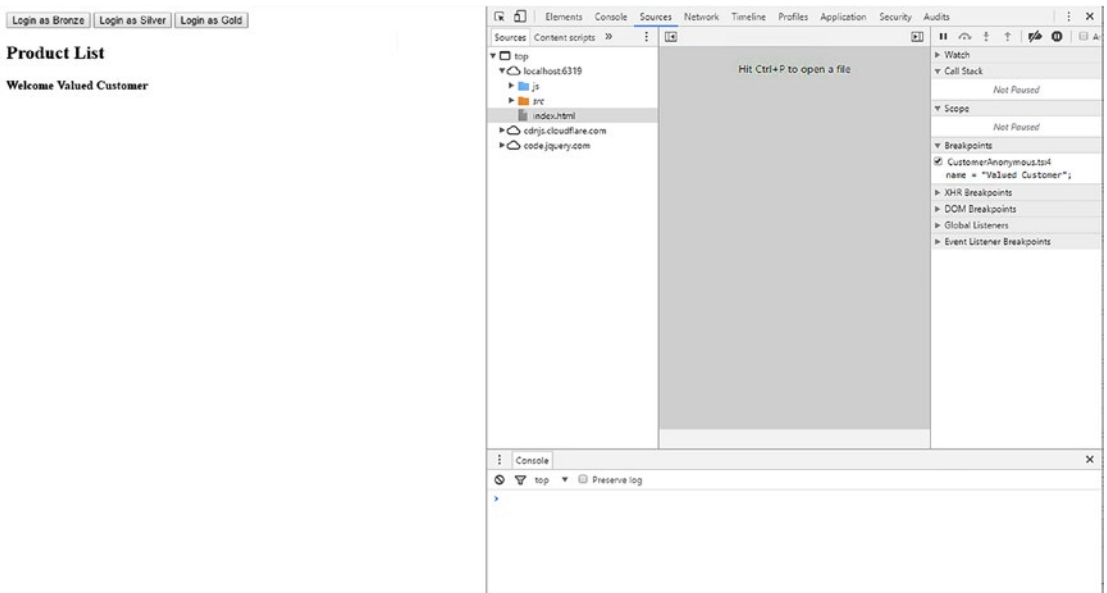


Figure 7-14. TypeScript sources

Click the `IndexVM.ts` file under the `src` folder to open it. This file looks just like the TypeScript file in the solution. Add a breakpoint to the `msg` variable declaration of the `setWelcomeMsg` function on line 8, shown in Figure 7-15.

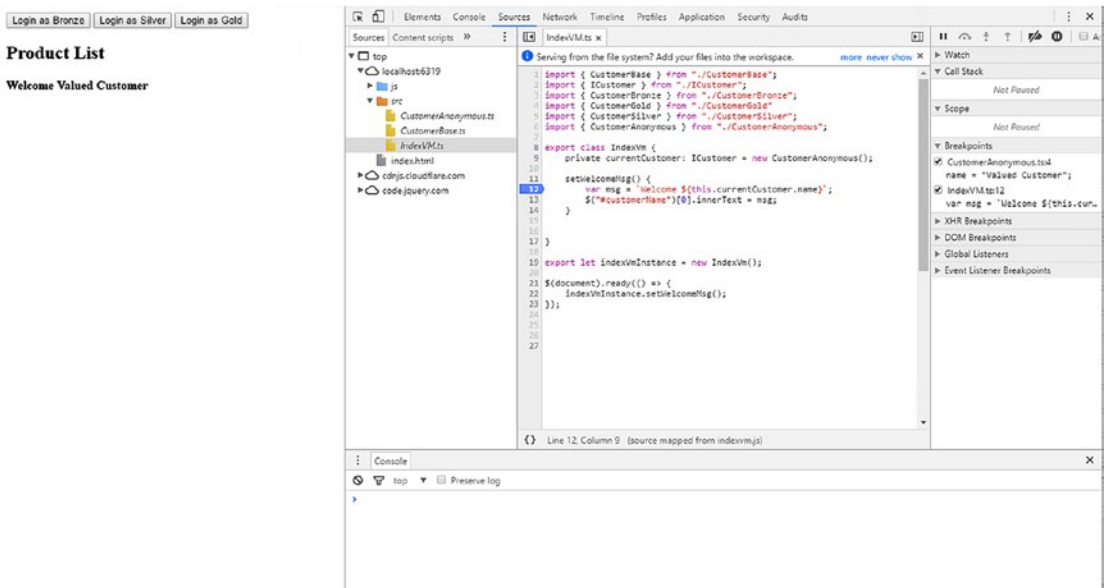


Figure 7-15. Breakpoint set

Refresh the page and you will see that your breakpoint in TypeScript is hit. If you hover on the `currentCustomer` variable being returned, you will see the values in that variable, shown in Figure 7-16.

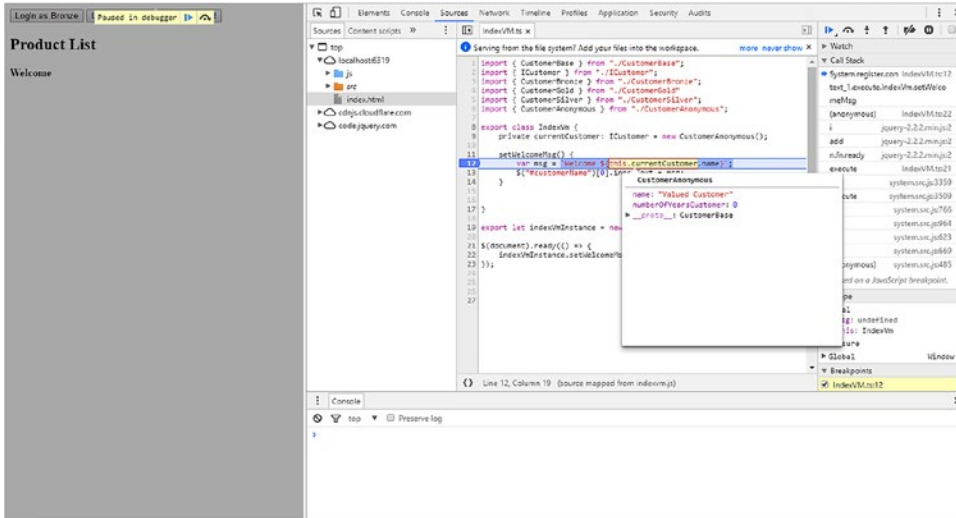


Figure 7-16. Breakpoint hit

Displaying the name of the anonymous customer is a good start. Now the sample will add the functionality of switching between the different customers and displaying a product list with prices for each.

We will continue with adding the ability to switch between customers. The sample is designed to show how TypeScript works not to provide an elaborate login and security system. Thus, it will simply add a button to the page that simulates a login for each customer type. We have already added the HTML for the necessary buttons when we made our last changes to the `index.html` file. Now we will continue by hooking these buttons up and letting the user change customer types in the `IndexVm` class.

Open the `IndexVm.ts` file in the `src` folder under the TypeScriptSample project. Add a method for each customer type that reflects the login logic for that customer, as shown in the following bolded code snippet.

```
export class IndexVm {
  private currentCustomer: ICustomer = new CustomerAnonymous();

  loginBronze() {
    this.currentCustomer = new CustomerBronze();
    this.setWelcomeMsg();
  }

  loginSilver() {
    this.currentCustomer = new CustomerSilver();
    this.setWelcomeMsg();
  }

  loginGold() {
    this.currentCustomer = new CustomerGold();
    this.setWelcomeMsg();
  }
}
```



```

    setWelcomeMsg() {
      var msg = `Welcome ${this.currentCustomer.name}`;
      $("#customerName")[0].innerText = msg;
    }
  }
}

```

To enable users to click the buttons defined in the HTML, we will use jQuery to react to user interaction. Later in the book we will demonstrate how various UI frameworks like Angular and React can simplify this type of interaction but jQuery will serve our purposes for now. The code for handling the button click events is highlighted in bold:

```

export let indexVmInstance = new IndexVm();

$(document).ready(() => {
  indexVmInstance.setWelcomeMsg();
});

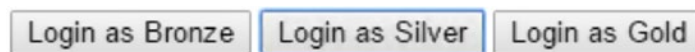
$('#loginBronze').click(function () {
  indexVmInstance.loginBronze();
});

$('#loginSilver').click(function () {
  indexVmInstance.loginSilver();
});

$('#loginGold').click(function () {
  indexVmInstance.loginGold();
});

```

Press F5 inside of Visual Studio to run the application. Click the login button for each customer type. Notice the welcome message changes for each customer type. The `setWelcomeMsg` method provides different results depending on what the `currentCustomer` is set to. This is possible because of the `ICustomer` interface that each customer type implements through the `CustomerBase` class. Figure 7-17 shows the results after clicking the Login as Silver button.



Product List

Welcome Silver Customer

Figure 7-17. Silver customer type logged in

What would happen if we try to set the `currentCustomer` variable in the `IndexVM.ts` file to a string? It produces a compilation error because the `currentCustomer` variable has a type of `ICustomer`. In straight JavaScript, you would not know of this bug until you ran the application and clicked the button. It is much nicer to catch these types of mistakes at compile time instead of at runtime.

This is an example of type safety in TypeScript. By telling the TypeScript compiler that the `currentCustomer` variable is of type `ICustomer`, we could detect errors before runtime. This becomes very handy once the application grows and you have hundreds of lines of code being shared across many TypeScript files.

Finishing the Application by Adding a Product List

By now the sample application has demonstrated many key features of TypeScript. Finish it off by taking advantage of more of these features to display some products and their prices. To do this we will need to add two more TypeScript files to our project under the `src` folder. These files will be called `IProduct.ts` and `Product.ts` and the code for each of these files follows:

`IProduct.ts`

```
import { ICustomer } from "./ICustomer";

export interface IProduct {
  name: string;
  currentPrice(customer: ICustomer);
}
```

`Product.ts`

```
import { IProduct } from "./IProduct";
import { ICustomer } from "./ICustomer";

export class Product implements IProduct {

  name = "";
  private basePrice = 0.0;

  constructor(nameValue: string, basePriceValue: number) {
    this.name = nameValue;
    this.basePrice = basePriceValue;
  }

  currentPrice(customer: ICustomer) {
    let discount = this.basePrice * customer.discountPercent();
    return this.basePrice - discount;
  }
}
```

This product class implements the `IProduct` interface. The initial state of a `Product` instance is configured in the class constructor. It has a private variable `basePrice` that can only be set by the constructor. The `currentPrice` method uses the discount from the customer variable and the private `basePrice` variable to calculate and return a price.

The product interface and class are now complete. Next, we need to modify the `Index.html` and `IndexVM.ts` files to load some products. To achieve this, we will again use jQuery to modify an HTML element, which we have already added to the `Index.html` files. The list of products for a particular customer will be displayed in the `<table>` element in bold.

```

<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <title>TypeScript Sample</title>
</head>
<body>
  <div>
    <button id="loginBronze">Login as Bronze</button>
    <button id="loginSilver">Login as Silver</button>
    <button id="loginGold">Login as Gold</button>
  </div>

  <h2>Product List</h2>
  <h4 id="customerName">Welcome</h4>

  <table id="productsTable"></table>

  <script src="https://cdnjs.cloudflare.com/ajax/libs/systemjs/0.19.41/system.js">
</script>
  <script src="https://code.jquery.com/jquery-2.2.2.min.js" integrity="sha256-36cp2Co+
/62rEAAyHLMRCPIych47CvdM+uTBJwSzwjI=" crossorigin="anonymous"></script>
  <script type="text/javascript">
    System.config({
      defaultJSExtensions: true
    });
    System.import('./js/indexvm.js');
  </script>

</body>
</html>

```

To build and display the table, open the `IndexVM.ts` file located in the `src` folder of the `TypeScriptSample` project. This code will create a list of products and load them on the page. Inside the `IndexVm` class, add the appropriate `import` statements to the top of the file. Next, create a private array variable to hold a list of products. Also, create a private method to load products and a public method to display that list. Finally, call the method to display the list when the page loads and when a new customer type is logged on. These additions are shown in the following bolded code snippet.

```

import { IProduct } from "./IProduct"
import { Product } from "./Product"

export class IndexVm {

  private currentCustomer: ICustomer = new CustomerAnonymous();

  private productList: Array<IProduct> = new Array(0);

  loginBronze() {
    this.currentCustomer = new CustomerBronze();
    this.setWelcomeMsg();
    this.displayProducts();
  }

```

```

loginSilver() {
  this.currentCustomer = new CustomerSilver();
  this.setWelcomeMsg();
  this.displayProducts();
}

loginGold() {
  this.currentCustomer = new CustomerGold();
  this.setWelcomeMsg();
  this.displayProducts();
}

displayProducts() {
  this.loadProducts();

  let htmlToDisplay: string = "<th>Product Name</th><th>Price</th>";

  this.productList.forEach(product => {
    htmlToDisplay += `<tr><td>${product.name}</td><td>${product.currentPrice(this.
currentCustomer)</td></tr>`;
  });

  $("#productsTable")[0].innerHTML = htmlToDisplay;
}

private loadProducts() {
  this.productList.length = 0;

  this.productList.push(new Product("Product 1", 100.00));
  this.productList.push(new Product("Product 2", 200.00));
  this.productList.push(new Product("Product 3", 300.00));
  this.productList.push(new Product("Product 4", 400.00));
  this.productList.push(new Product("Product 5", 500.00));
  this.productList.push(new Product("Product 6", 600.00));
  this.productList.push(new Product("Product 7", 700.00));
  this.productList.push(new Product("Product 8", 800.00));
  this.productList.push(new Product("Product 9", 900.00));
  this.productList.push(new Product("Product 10", 1000.00));
}

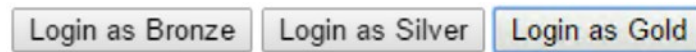
setWelcomeMsg() {
  var msg = `Welcome ${this.currentCustomer.name}`;
  $("#customerName")[0].innerText = msg;
}
}

```

For this sample, the `loadProducts` method is simply loading a list with some hardcoded products. In the real world, these products could be loaded from an API call or from a local resource. The purpose of the sample is to show the constructor for these products being used to create and add them to an array. The `displayProducts` method uses a `foreach` loop to run through the list of products and generate the row and

columns needed to the display the products. Then it uses jQuery to get the table element from the page and add in the HTML. The rest of the changes are simple calls to the display products method to load the products with the appropriate prices.

Press F5 inside of Visual Studio to run the application. Notice that, by default, the anonymous customer gets a list of products that have no discount. Click the login button for each customer type. Notice the price changes for each type. Again, this is a demonstration of polymorphism. The `currentPrice` method of the `Product` class provides different results depending on what customer is provided. Remember earlier in the chapter we changed the behavior for gold customers to hard code their discount to 20 percent. Figure 7-18 shows the results after clicking the Login as Gold button.



Product List

Welcome Gold Customer

Product Name	Price
Product 1	80
Product 2	160
Product 3	240
Product 4	320
Product 5	400
Product 6	480
Product 7	560
Product 8	640
Product 9	720
Product 10	800

Figure 7-18. Gold customer type logged in

■ **Source Code** The `TypeScriptSample` solution can be found in the `Chapter_07` subdirectory of the download files.

Summary

This chapter was an introduction to the basic syntax and setup of TypeScript. While TypeScript is one of many JavaScript transpilers it has enjoyed a massive rise in popularity in a very short time. TypeScript adds many robust features to the core JavaScript language and allows developers to take advantage of these

features while still being able to deploy raw JavaScript to environments that do not have modern browsers. It is key to remember that TypeScript “adds features” to JavaScript. The TypeScript syntax builds on top of the JavaScript syntax and all code is ultimately transpiled down to raw JavaScript, targeting whatever version of JavaScript you would like. This also makes it easy to use TypeScript while still incorporating third-party JavaScript libraries.

There are many more capabilities that the TypeScript language provides than were discussed in this brief introductory chapter. Microsoft’s online documentation and tutorials are a great place to learn more and to broaden your knowledge of the TypeScript languages. These can be found here: <https://www.typescriptlang.org/docs/>. TypeScript is already evolving and, at the time of this writing, on version 2.1.4. The ecosystem surrounding TypeScript is very active and new products and libraries are constantly being released to allow developers to better take advantage of this language.

The remaining chapters of this book utilize TypeScript to demonstrate several client side UI frameworks. Within these chapters you will learn even more about TypeScript by “seeing it in action” in the implementation of user interfaces much more advanced than the one utilized here.

CHAPTER 8



Angular 2

There are a number of different resources, including the extensive documentation by the Angular 2 team, that can be used to guide you through creating an Angular 2 application. The purpose of this chapter is not to be a comprehensive guide to Angular 2, but to focus on the ASP.NET Core and common comparable differences with Angular 2 and other Single Page App frameworks covered in the later chapters.

In order to do this, this chapter will take you through the process of creating the SpyStore application that has been used previously, using Angular 2. This chapter will highlight core components of Angular 2 throughout the process of creating the SpyStore application. It will try to point out other features with Angular 2 that may not be covered in an effort to point the reader to areas of potential future research. Throughout this chapter Angular 2 may be shortened to just Angular, if Angular 1 is intended, then it will be explicitly stated as such. In addition, Angular 2 is an open source product that is subject to change. All the information is based on the product as it stands at the time of publishing.

Creating a New Visual Studio Core Project

There are a number of ways to create an Angular 2 application. Given the ASP.NET Core and Visual Studio focus of this book, this chapter will focus specifically on creating a new Angular 2 application using Visual Studio. To get started, open Visual Studio and create a new project. In the New Project dialog, shown in Figure 8-1, select a new ASP.NET Core Web Application (.NET Core), change the name to SpyStore.Angular2, and click OK.

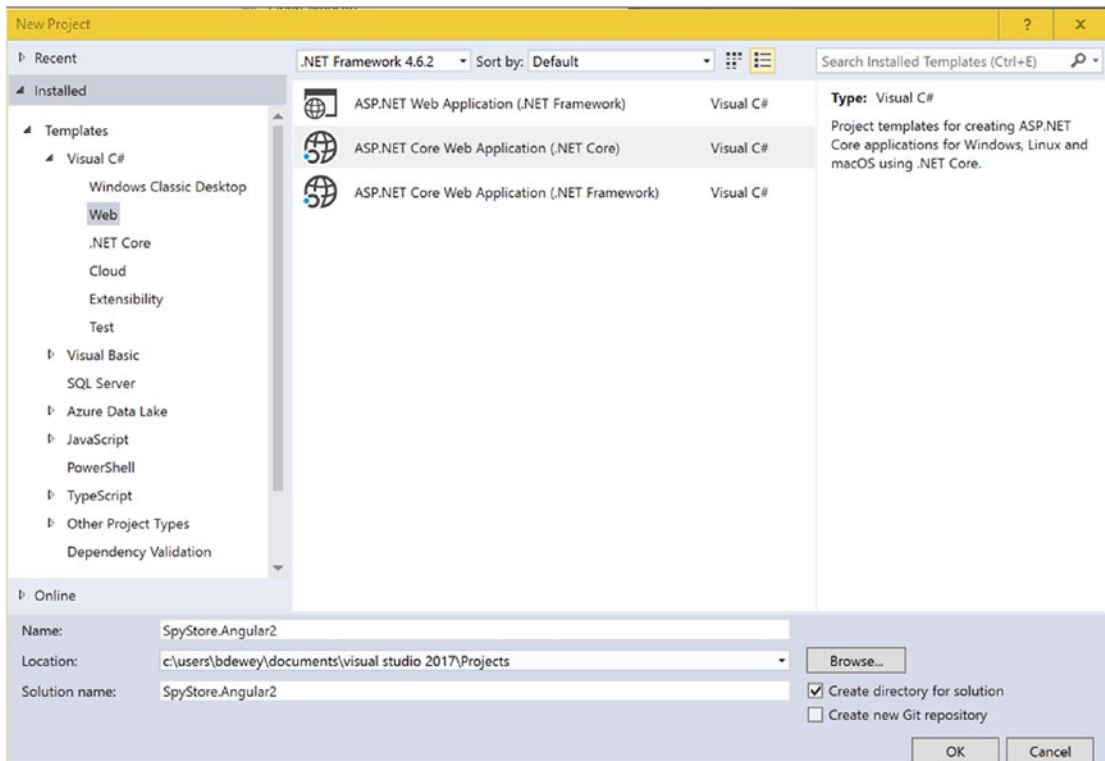


Figure 8-1. Dialog for creating a new project

After confirming your selection, the New ASP.NET Core Web Application dialog will appear, as shown in Figure 8-2. From this dialog, select an Empty project, which should be the default selection, and click OK. This will create a new project and automatically kick off the package restoration process. From here, you have a plain ASP.NET Core application, which you can launch, but nothing specific to Angular 2 has been added to the project yet.

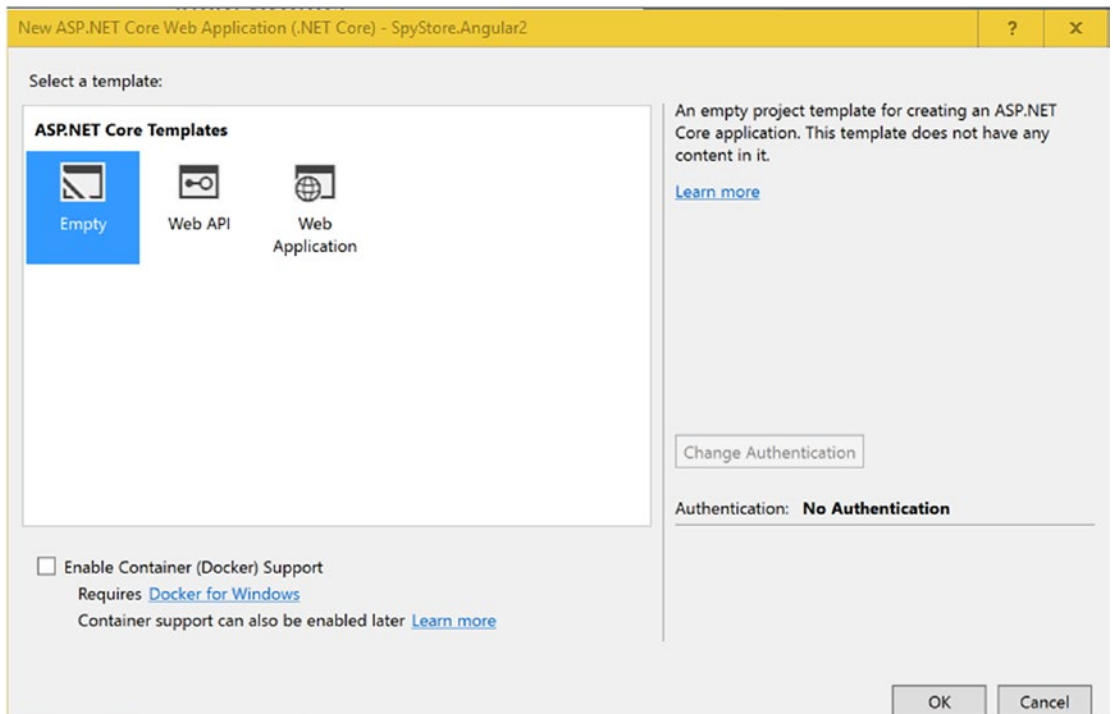


Figure 8-2. Second page of the dialog for creating a new project

■ **Note** There are a number of ways to get up and running with Angular 2. With the release of Angular 2, the team created a new Angular 2 command line interface (CLI) tool. This tool can be used to create the initial project as well as many of the common components, directives, and services used by Angular 2. For more information on the Angular 2 CLI, see the documentation online at <https://cli.angular.io/>.

Project Files

When the project is initially created, there are two project files and a `web.config`. Given that Angular 2 is a single page application framework and uses its own internal routing, there are a few tweaks that need to be made to these base files in order to get started. These tweaks revolve around setting up the ASP.NET Core middleware to act as a static file server. As with all middleware components in ASP.NET Core, first you need to add the associated package as a NuGet reference; in this case add the `Microsoft.AspNetCore.StaticFiles` package. Figure 8-3 shows adding the NuGet package to your project for static files, which allows for serving static HTML files for the app.

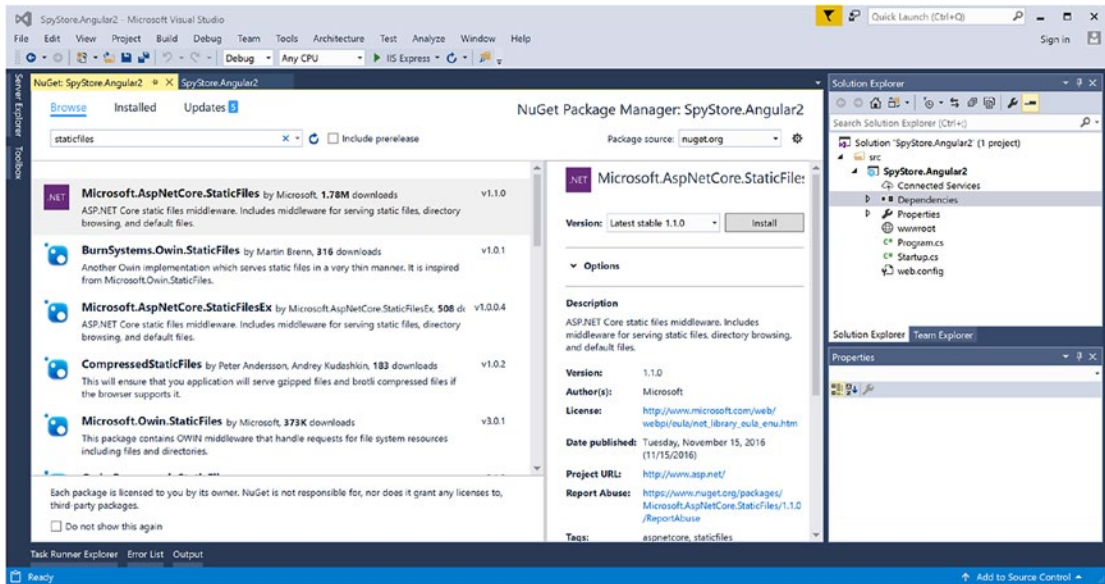


Figure 8-3. Adding the `StaticFiles` NuGet package

After adding the NuGet package, the dependency will be added to the project for use in the next step.

Setting Up the Startup Class

Now that the `StaticFiles` dependency has been added to the project, it can be used by the application, before that can happen it has to be added to the ASP.NET Core pipeline, which is set up in the `Startup.cs` file. Inside the `Startup.cs` file, locate the `configure` method. In the `configure` method, you will find the following code.

```
app.Run(async (context) =>
{
    await context.Response.WriteAsync("Hello World!");
});
```

Replace this code with the middleware code for serving static files.

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

In the same file there should also be some code, which may currently be commented out, for using default files. This enables default `index.html` files to be served when a directory is requested. Either uncomment out that code or add the following line to the Startup class.

```
app.UseDefaultFiles();
```

Adding this middleware alone allows for the files, such as dependent JavaScript, application JavaScript, and HTML views to be served up. Unfortunately, this doesn't address the routing concerns. Given that Angular 2 uses HTML5 based URL routing, as opposed to hash (#) routing from previous incarnations, the server needs to be set up to ignore those route parameters in the URL, serve up the root `index.html` file, allow Angular to parse the route, and direct the user to the appropriate page. A simple way to resolve this is to create a custom middleware function that checks for missing files with missing file extensions and route them all to the root page. The following code shows the custom middleware function to achieve this. The function can be useful for any type of single page application written in ASP.NET.

```
// include using System.IO; at the top of the file.
//
// Route all unknown requests to app root
app.Use(async (context, next) =>
{
    await next();

    // If there's no available file and the request doesn't contain an
    // extension, then rewrite request to use app root
    if (context.Response.StatusCode == 404 && !Path.HasExtension(context.Request.Path.Value))
    {
        context.Request.Path = "/index.html";
        await next();
    }
});
```

Now that the web site has been customized to support Angular 2 and can serve static files, we can start adding the necessary Angular 2 dependencies. NPM will be used to do this in the next section.

NPM Install

For this project, we need to install a number of different project dependencies using NPM. Table 8-1 lists the dependencies that are needed for the application.

Table 8-1. *NPM Packages Needed for the SpyStore Angular 2 Application*

Package	Save Mode
core-js	--save
ng2-bootstrap	--save
reflect-metadata	--save
Rxjs	--save
systemjs	--save
zone.js	--save
@angular/core	--save
@angular/common	--save
@angular/compiler	--save
@angular/platform-browser	--save
@angular/platform-browser-dynamic	--save
@angular/http	--save
@angular/router	--save
@angular/forms	--save
gulp	--save-dev

As covered previously in the NPM chapter, before you can install these NPM packages, you need to initialize the project by running the NPM initializer from the command line. To do this, open the command line to the project directory and run the following command.

```
npm init
```

For each of the dependencies listed in Table 8-1, install them using the `npm install` command.

```
npm install <package> [--save|--save-dev]
```

Once all the node packages are installed, you can proceed to getting the application set up to use those packages.

Gulp Setup

The use of Gulp for the purposes of this application are only used for copying files. The needs of other applications may vary. Given the styling and images are located in a specific bootstrap theme project and the fact that the `node_modules` are by default stored outside of the `wwwroot` folder, the Gulp script for this application focuses on copying files to the appropriate locations in the `wwwroot` folder. The Gulp script for the Angular 2 application can be found in the example code for the project and consists of one root copy task and one subtask copy:node.

```
var gulp = require('gulp');

gulp.task('copy', ['copy:node']);

gulp.task('copy:node', function () {
  return gulp.src([
    'node_modules/@angular/**/bundles/*',
    'node_modules/core-js/client/shim.min.js',
    'node_modules/zone.js/dist/zone.js',
    'node_modules/reflect-metadata/Reflect.js',
    'node_modules/systemjs/dist/system.src.js',
    'node_modules/rx**/**',
    'node_modules/core-js**/**',
    'node_modules/ng2-*/dropdown/**/*.*ts',
    'node_modules/ng2-*/dropdown/**/*.*js',
  ]).pipe(gulp.dest('./wwwroot/lib/'));
});
```

After adding the `gulpfile.js` file to your project, you can run the following command on the command line.

```
gulp copy
```

After running the copy task, all of the Angular files and styling code will be present in the `wwwroot` folder of your application and available for the next section, where you create the application components and the `index.html` file.

■ **Note** If you look at the at the Angular 2 documentation at <https://angular.io/docs/ts/latest/quickstart.html>, it will use Node for running and building the TypeScript files. Due to the ASP.NET Core and Visual Studio focus of this book, this project will run through Visual Studio and will use Visual Studio to build the TypeScript files.

Typescript Setup

At this point, the ASP.NET Core application to serve the files has been set up and the third-party dependency files for the site have been loaded. The last thing that needs to be done before writing the Angular 2 code is to set up the project to enable TypeScript, which will be used for this application.

To set up TypeScript, create a new folder on the root of your project called `scripts`, which is where the TypeScript files will be placed. After that, create a new file on the `scripts` folder of the project called `tsconfig.json` and add the following code.

```
{
  "compilerOptions": {
    "target": "es6",
    "module": "commonjs",
    "moduleResolution": "node",
    "sourceMap": true,
    "emitDecoratorMetadata": true,
    "experimentalDecorators": true,
    "noEmitOnError": true,
    "noImplicitAny": false,
    "outDir": "../wwwroot/js/app/",
    "removeComments": false
  },
  "exclude": [
    "../node_modules",
    "../wwwroot"
  ]
}
```

■ **Note** Angular 2 is built with TypeScript and therefore it is common to build applications using TypeScript as well. There is documentation on the Angular site about building applications with other languages. The SpyStore application will use TypeScript and the remainder of this chapter will focus on TypeScript for the examples. Refer back to the previous chapter on TypeScript as needed.

With this file added, Visual Studio will automatically recognize the TypeScript file and build it as part of the project build process. This concludes the Visual Studio setup for the Angular 2 project. The next section reviews setting up the basic Angular 2 files and module loaders to get the scripts on the page.

Main SpyStore App Component Setup

Now that the basics of the project are set up, it is time to start building the SpyStore Angular 2 application. The following list outlines the steps needed to get a basic project set up.

- Create a root page
- Create app module and root app component
- Create root app view

The following section walks through the steps to create a basic Angular 2 application that will be expanded upon to create the SpyStore application.

Creating the Root index.html Page

Angular 2 applications start as a basic HTML page with a root component. In Visual Studio, right-click on the `wwwroot` folder and create a new `index.html` file, as shown in Figure 8-4.

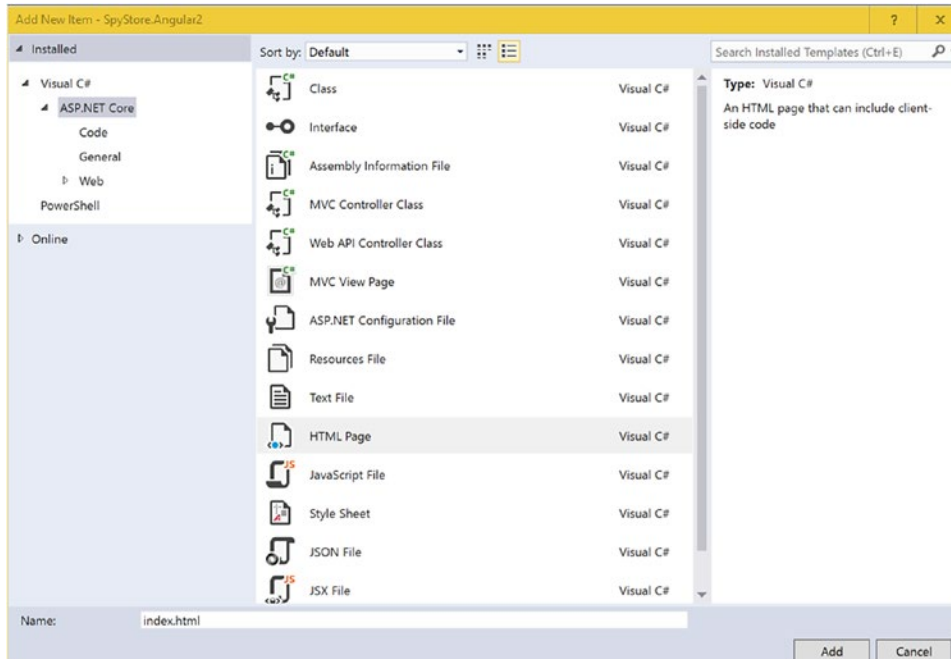


Figure 8-4. Dialog for creating a new `index.html` file

This creates a very basic HTML file. In order to be used as an Angular 2 application, the following modification need to be made.

First, locate the end head tag (`</head>`). On the line directly before that, add `<base href="/" />`. This tells the HTML parser that no matter what the current page is, it should use the root `'/'` directory for all relative links.

Secondly, in the main body of the application, create a new root component element called `<spystore-app>Loading...</spystore-app>`. Notice the contents of this element include the word `Loading`. This displays on the screen before the Angular 2 application is loaded and can be anything you want. It will be overridden when the application loads.

The final step to getting the Angular 2 application up and running is to add the third-party tools script tag references. Simply adding the `<script />` tags manually would do the trick, but recently module loaders have become popular for this task and they eliminate the need for maintaining the script tags with every new file. Module loaders were covered in more depth in the previous chapter. This version of the `SpyStore` app uses `SystemJS` as its module loader of choice. To set up `SystemJS`, add the following tags to the bottom of your page just before the end body tag.

```
<!-- 1. Load libraries -->
<script src="/lib/shim.min.js"></script>
<script src="/lib/zone.js"></script>
<script src="/lib/Reflect.js"></script>
<script src="/lib/system.src.js"></script>
```

```
<!-- 2. Configure SystemJS -->
```

```
<script>
  /**
   * System configuration for Angular 2 samples
   * Adjust as necessary for your application needs.
   */
  (function (global) {
    System.config({
      paths: {
        // paths serve as alias
        'npm:': 'node_modules/'
      },
      // map tells the System loader where to look for things
      map: {
        // our app is within the app folder
        app: '/js/app',
        // angular bundles
        '@angular/core': '/lib/core/bundles/core.umd.js',
        '@angular/common': '/lib/common/bundles/common.umd.js',
        '@angular/compiler': '/lib/compiler/bundles/compiler.umd.js',
        '@angular/platform-browser': '/lib/platform-browser/bundles/
platform-browser.umd.js',
        '@angular/platform-browser-dynamic': '/lib/platform-browser-dynamic/
bundles/platform-browser-dynamic.umd.js',
        '@angular/http': '/lib/http/bundles/http.umd.js',
        '@angular/router': '/lib/router/bundles/router.umd.js',
        '@angular/forms': '/lib/forms/bundles/forms.umd.js',
        // other libraries
        'rxjs': '/lib/rxjs',
        'ng2-bootstrap/dropdown': '/lib/ng2-bootstrap/dropdown/index.js',
        'ng2-bootstrap': '/lib/ng2-bootstrap'
      },
      // packages tells the System loader how to load when no filename and/or no extension
      packages: {
        js: {
          defaultExtension: 'js'
        },
        app: {
          //boot: './boot.js',
          defaultExtension: 'js'
        },
        rxjs: {
          defaultExtension: 'js'
        },
        'ng2-bootstrap': {
          defaultExtension: 'js'
        }
      }
    });
  })(global);
</script>
```



```

        'angular2-in-memory-web-api': {
            main: './index.js',
            defaultExtension: 'js'
        }
    }
});
})(this);
System.import('js/app/boot').catch(function (err) { console.error(err); });
</script>

```

In addition to the Angular 2 setup code, this is a good time to add any style sheets that exist for the application. For the SpyStore application, there is a single CSS file that is copied over as part of the Gulp task. In order to add this, create a new `<link />` in the `<head>` of the page.

```
<link href="/css/spystore-bootstrap.css" rel="stylesheet" />
```

Running the application now would result in a basic application showing a loading message using the style for the SpyStore application. The next step is to create the app module and root component.

Creating the Root App Component

The `<spystore-app>` element that was added to the `index.html` page represents the root component. To create the implementation for the root component, you need to declare the root component in TypeScript. To do this, create a new file in the scripts folder called `app.component.ts`. In this file, include the following code.

```

import { Component } from "@angular/core";

@Component({
  selector: "spystore-app",
  templateUrl: "/app/app.html",
})
export class AppComponent {

  constructor() {
  }
}

```

This code includes a basic `AppComponent` class; the key to making this work is the `@Component` decorator, which defines two important details about the component. The first is the selector, which is a standard CSS selector notation. In this case, the `spystore-app` represents any tag with that name. The second is the `templateUrl` for the file, which represents the HTML for the view.

The `app.html` template for the root component is included as part of the example code and can be found under `End-Part1/app/app.html`. Include this file in your project in the location specified for the `templateUrl` for the component.

Creating the App Module

The application root component is now set up. Unfortunately, it doesn't actually run anything yet because it hasn't been wired up as an actual Angular module. To do this, create a new file in the `scripts` folder called `app.module.ts`. Inside that file, include the following code.

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';

@NgModule({
  imports: [
    BrowserModule
  ],
  declarations: [
    AppComponent
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {
}
```

The `AppModule` class is completely empty. The important part again is the decorator. This `@NgModule` decorator includes a number of options, as stated at <https://angular.io/docs/ts/latest/guide/ngmodule.html>. The basic options include `imports`, which declare the `BrowserModule` and enable the Angular application to run in a browser with HTML, `declarations`, which tell the app what is required for the application to function, and a `bootstrap` property, which tells the app which component is the root component.

Creating the Angular Bootstrap

Now that there is a root component and a module defined for the app, the last thing is to bootstrap the app. To do this, create a new `boot.ts` file in the `scripts` directory. This file includes the following code, which bootstraps the application.

```
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { AppModule } from './app.module';

const platform = platformBrowserDynamic();
platform.bootstrapModule(AppModule);
```

At this point running the app will launch a basic Angular 2 application with a root component. In order to proceed it is best to review some core concepts of Angular 2. The next section highlights the concepts that are used throughout the remainder of the setup of this application.

■ **Note** This book focuses on the concepts of Angular 2 that are relevant to the creation of the `SpyStore` application. For a complete tutorial of Angular 2, see the documentation at <http://angular.io>.

Core Concepts

Angular as a single page app framework has a number of core concepts that need to be understood before building out an application. Table 8-2 provides an overview of the concepts that are covered in this section and are used throughout the remainder of this chapter to build out the SpyStore Angular 2 application.

Table 8-2. Core Concepts for Angular 2 Development

Core Concept	Concept Overview
Application Initialization	Also known as bootstrapping, initializes the application by starting up the main module that is provided to the bootstrapper. This process also manages the dependency injection container.
Components	Components are a new part of Angular and are made up of a view and view logic similar to directives in previous versions of Angular.
Services	Services are a mechanism for abstracting and sharing code among multiple components. In the SpyStore application services are used primarily to access the SpyStore API. The lifetime of a service is managed by the dependency injection container.
Templates	Templating is the feature in Angular 2 that allows for displaying data and manipulating the HTML based on component logic. Angular 2 has a brand new templating syntax compared to Angular 1. There are mechanisms in place for handling inbound and outbound binding of objects and events to components.
Routing	Routing allows multiple pages to be loaded within an application. In the case of the SpyStore application, the main root component contains an outlet for which all other components can be loaded, thus creating typical navigation that you would see in traditional applications.

Application Initialization

Application initialization, as seen when the app was first set up, is based on two core functions—declaring a module and bootstrapping the application. First, let's take a look at declaring a module.

Modules are the building blocks of angular applications. They allow developers to combine services, directives, and components into logical groups and define them as a hierarchy where all the imports and dependencies roll up to a parent module. In order to define a module in Angular 2 using TypeScript, export an empty class and add the `@NgModule` decorator to that class, as shown here.

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';

@NgModule({
  imports: [ BrowserModule ],
  declarations: [ AppComponent ],
  bootstrap: [ AppComponent ]
})
export class AppModule {
}
```

The `NgModule` decorator accepts a single argument, which is the configuration for the module. The basic configuration for a new browser based application requires `imports`, which tell the module to include the `BrowserModule`, declarations, which tell the module that it is using the `AppComponent` component, and finally a bootstrap component, which in this case is the `AppComponent` as well.

Table 8-3 shows a list of all the options available for application initialization using the `NgModule` decorator.

Table 8-3. Options of the `ngModel` Decorator

Option	Description
<code>providers</code>	An array of injectable objects or services that can be included as part of the dependency injection container.
<code>declarations</code>	An array of directives, components, or pipes that will be used in the application.
<code>imports</code>	An array of modules whose exported declarations will be used in the application.
<code>exports</code>	An array of objects that can be used when this module is imported into other modules.
<code>entryComponents</code>	An array of components that should be compiled when the modules are declared.
<code>bootstrap</code>	An array of components that should be bootstrapped when the application loads.
<code>schemas</code>	An array of elements, or properties, that are not angular components and should be declared with the module.
<code>id</code>	The ID used for the module when loading from <code>getModuleFactory</code> .

For more information on application initialization, see the Angular documentation at <https://angular.io/docs/ts/latest/guide/appmodule.html> and, for additional information on the `NgModule` options, see <https://angular.io/docs/ts/latest/api/core/index/NgModule-interface.html>.

Components

Components are one of the core building blocks of any application in Angular. Components are made of two parts, a view and view logic. In the case of the `SpyStore` application, there is a root component, a component for each page, and a component for the shopping card record.

In addition, components can interact with other objects, such as service via dependency injection into the component class and via directives, which can be used by the view to interact with the DOM.

The following code is an example taken from the product details page where the view, or HTML, can bind itself to objects and properties on the view logic, or component class. The following is the component class for the product details page.

```
import {Component, OnInit} from "@angular/core";
import {ActivatedRoute, Params } from '@angular/router';
import {ProductService} from '../product.service';

@Component({
  templateUrl: "app/components/productDetail.html"
})
export class ProductDetailComponent implements OnInit {
  product: any;
```

```

constructor(
  private _route: ActivatedRoute,
  private _productService: ProductService) {
  this.product = {};
}

ngOnInit() {
  let id: number;
  this._route.params.forEach((params: Params) => {
    id = +params['id'];
  });
  this.project = this._productService.getProduct(id);
}
}

```

This component class defines a public property called `product`, which is mapped to the value returned from the product service. Then in the view, content and DOM elements can be bound to the values in that object. More on templating and binding will be discussed later in the “Templating” section. The following code highlights the view that corresponds to the product details component.

```

<h1 class="visible-xs">{{product.ModelName}}</h1>
<div class="row product-details-container">
  <div class="col-sm-6 product-images">
    <img [src]="'images/' + product.ProductImageLarge" />
    <div class="key-label">PRODUCT IMAGES</div>
  </div>
  <div class="col-sm-6">
    <h1 class="hidden-xs">{{product.ModelName}}</h1>
    <div class="price-label">PRICE:</div>
    <div class="price">{{product.CurrentPrice | currency:'USD':true:'1.2-2'}}</div>
  </div>
</div>

```

The component class and view combine to make a component. This concept repeats itself numerous times with an Angular application. Applications will typically map using a treeview of the components to establish what is called a component tree.

In the component class, there is a lifecycle event called `ngOnInit()`. This event is called by the framework when the component is initialized as part of the component lifecycle. For more information about lifecycle events, see <https://angular.io/docs/ts/latest/guide/lifecycle-hooks.html>.

Services

Services provide a nice way to abstract your code out of a single component and allow it to be reused across different components. Services are most commonly used for data access, but can be used for all types of shared logic or as a layer to keep code maintainable and more testable.

When using TypeScript in Angular, services are nothing more than classes. The class can have properties and methods just like any other class. The real benefit for services is that they can be injected, using a technique called *dependency injection* (DI), into other services or components. All the dependent services are controlled and maintained by the dependency injection framework, which looks at the module's providers, builds them up, and provides them to the components and services when needed.

```
export class MyDataService {
  getData(): any[] {
    return someArrayOfData;
  }
}
```

In order to make this service available to other components, it has to be registered in the module. To do this, just add the type to the list of providers shown here.

```
import { MyDataService } from './mydataservice';

@NgModule({
  imports: [ /*...*/ ],
  declarations: [ /*...*/ ],
  providers: [ MyDataService ],
  bootstrap: [ /*...*/ ]
})
export class AppModule {
}
```

As the services in an application expand, services may be dependent on other services, and therefore, services may have dependencies that need to be injected into their constructors as well. In Angular, if the class includes an injectable decorator, then it will be available for injection of its dependencies when created. For the purposes of future proofing your code and for consistency, it is recommended that developers include the `@Injectable` decorator in all their services, even if they don't have any dependencies. The following code shows an example of how to include the `@Injectable` decorator.

```
import {Injectable} from '@angular/core';

@Injectable()
export class MyDataService {
  /*...*/
}
```

By specifying the providers in the application module as defined here, the dependencies can be shared among all the services and components in that module. This can be very beneficial for sharing data and keeping the application performant, as data is not reloaded with every component. While this is often correct, there are some cases where you wouldn't want a common instance across all components and the application should define new instances of the service with each component. Angular accomplishes this by defining a providers property at the component level. If a service is defined as a provider at the component level instead of the module level, it will be created and destroyed as part of the component. The following code shows how to specify a service as a provider at the component level.

```
import { Component } from "@angular/core";
import { MyDataService } from './mydataservice';

@Component({
  providers: [ MyDataService ],
  templateUrl: "/app/components/products.html"
})
export class ProductsComponent {
  constructor() { }
}
```

For more information on Angular's dependency injection framework, see the documentation online at <https://angular.io/docs/ts/latest/guide/dependency-injection.html>.

Templating

Components were discussed earlier and are made up of two parts—the component logic and the view. In the case of our example the view is made up of HTML, which defines what will be rendered when the component is displayed. While plain HTML is great, it doesn't provide any real functionality or interact with the component logic in a meaningful way. This is where templating comes in for Angular.

A good place to start with templating is to understand the syntax while discussing the details of each one along the way.

Interpolation

The most basic way to interact with your component logic is to render a value from the component onto the view. A common way to do this is to use a feature called *interpolation*. For this example, assume that you have the following component class.

```
import {Component } from "@angular/core";

@Component( /* ... */)
export class ProductDetailComponent {
  productName: string = 'Escape Vehicle';
}
```

The corresponding view to display the product name for the component using interpolation would look like this code.

```
<h1>{{productName}}</h1>
```

Notice the double curly braces. This is the syntax for interpolation. Interpolation can work in a number of different ways as long as the template expression is a valid non-manipulative expression. In addition to rendering a simple property, some examples of valid template expressions include executing a method on a component and rendering their return value or calculating values together, such as `{{price * taxRate}}` to render the tax amount.

One might naturally think that Angular is executing these commands and rendering them, but it is actually converting these into property binding statements under the covers.

Property Binding

Property binding is available in two forms either using attributes or using a new syntax from Angular to trigger property binding. Here are examples of the same binding, but using property binding instead of interpolation.

```
<!-- Attribute Binding -->
<h1 bind-textContent="productName"></h1>

<!-- Property Binding in Angular -->
<h1 [textContent]="productName"></h1>
```

Event Binding

Property binding and interpolation are great for displaying data, but often times an application needs to accept user input in the form of event handlers. Angular handles events in a completely different way than typical JavaScript, where one would register event handlers. Angular uses a binding syntax similar to property binding shown previously, called *event binding*. In order to demonstrate this technique, assume the following component logic.

```
@Component( /* ... */)
export class ProductDetailComponent {
  addToCart() {
    // logic to add to cart
  }
}
```

The associated view to interact with this component logic using event binding would look like the following.

```
<button (click)="addToCart()">Add To Cart</button>
```

Notice that inside the button click event there is a call to the function from the component logic called `addToCart()`. Despite its similarities, this doesn't function like typical JavaScript where the button click event calls the `addToCart` function when the button is clicked. In reality the `(click)` syntax, with the parentheses, on the view is parsed by Angular and a one-way binding contract is established between the view target and the component logic.

This syntax is convenient and very readable for the developer. Similarly to property binding, this technique also works by using attribute binding. The attribute binding syntax that is equivalent to the event binding syntax would look like the following.

```
<button on-click="addToCart()">Add To Cart</button>
```

Two-Way Binding

The final binding syntax that applications typically encounter is a form of two-way data binding. This syntax involves combining the two techniques previously described. In order to demonstrate this, assume the component logic shown here.

```
@Component( /* ... */)
export class ProductDetailComponent {
  quantity: number = 1;
}
```

On the view there is a corresponding text input that is bound to this quantity field and looks like the following code.

```
<input [(ngModel)]="quantity" />
```


Notice the syntax for this quantity field is bound to a special `ngModel` property for Angular, but more importantly it is bound using a combination of the two binding syntax statements previously discussed and includes both the square brackets and the parentheses. The angular team in the documentation calls this binding a “banana in a box” syntax to help developers remember the order of the square brackets and parentheses.

This syntax tells Angular to bind the view to the value of the field as defined in the component logic, but also to automatically update the component field whenever the user changes the contents of the text input.

Not unlike the other binding examples discussed previously, two-way binding can also use attribute binding. The following code represents the attribute binding syntax for two-way binding.

```
<input bindon-ngmodel="quantity" />
```

■ **Note** While Angular doesn’t actually have two-way binding, this shorthand syntax can do a lot to provide a similar experience. For more information on the internals of this syntax, and how to write a component or directive that supports this syntax, see the following blog post at <http://blog.thoughttram.io/angular/2016/10/13/two-way-data-binding-in-angular-2.html>.

Structural Binding

The last form of binding is used by a few built-in Angular directives and involves structural binding, which converts view elements into nested templates. Two examples of these structural binding examples are `ngIf` and `ngFor`. To demonstrate these, assume the following component logic for the application.

```
@Component( /* ... */)
export class ProductComponent {
  products: any[];
}
```

The product view that corresponds to this component logic displays a repeated list of products based on the array backed by the `products` field.

```
<div *ngFor="let product of products">
  <h5>{{product.productName}}</h5>
  <!-- Other product information -->
</div>
```

Notice this form of binding includes an `*` as part of its syntax. This syntax is unique to these structural directives. In reality, the code is a shorthand for the following template.

```
<template ngFor let-product [ngForOf]="products">
  <div>
    <h5>{{product.productName}}</h5>
    <!-- Other product information -->
  </div>
</template>
```

There are a handful of other nuances with Angular templating, and it's best to read through the provided documentation to fully understand their impact. For more information on Angular templates, see the documentation online at <https://angular.io/docs/ts/latest/guide/template-syntax.html>.

Routing

Routing is the final critical component of Angular that you need to understand prior to building the SpyStore application in Angular. Routing provides a mechanism for navigating between different pages and views within Angular.

Figure 8-5 shows a basic diagram of how routing works. In the diagram notice there is a root component. Within this component, there is a router-outlet. This is a directive that is part of the router module and defines the location where the routed components will be rendered and replaced as the user navigates the application.

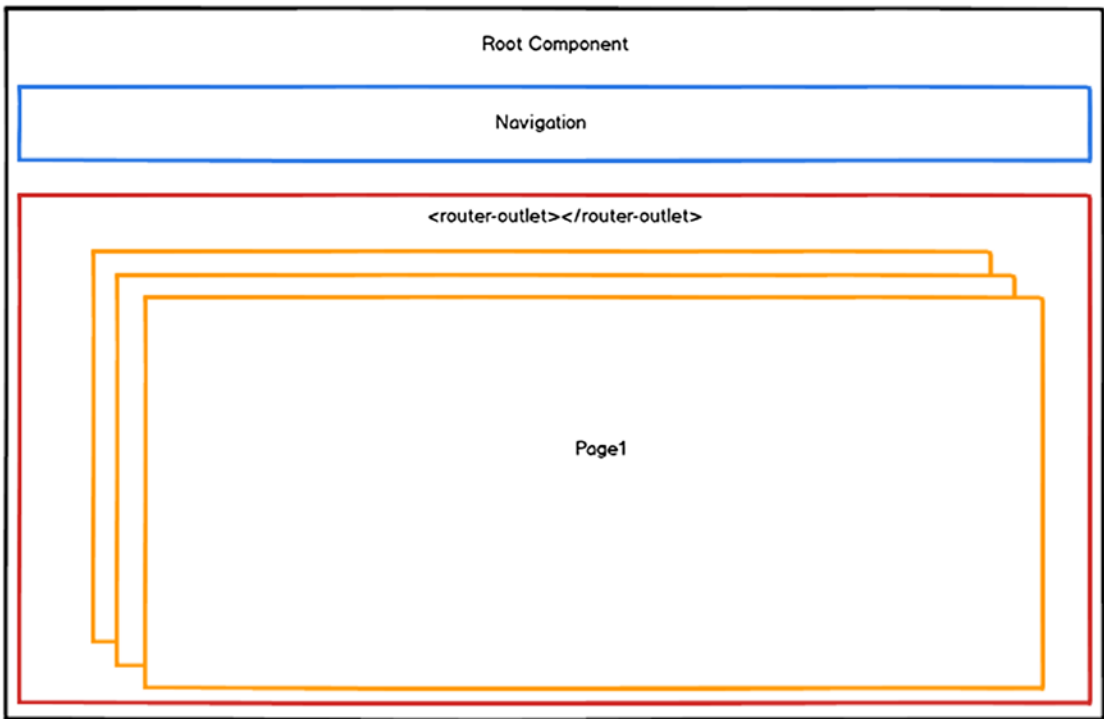


Figure 8-5. Routing example

Since routing is an optional aspect to Angular applications, the Angular team created a separate module for routing. The first step to adding routing to an application is to include the routing module and set it up as part of the application. If you are using NPM to install the Angular libraries, run the following command to install Angular routing.

```
npm install @angular/router --save
```

This includes the Angular router in `node_modules`. Once this is included it needs to be loaded as part of the application. Include the router bundle as a `script` tag in the `index.html` page or follow the documentation for the selected module loader.

The final step needed before the module can be used for routing is to add the `<base />` tag to the main page. This tag needs to be added to the last `<head>` tag and is typically set to the root address of the site, as stated in the following code.

```
<head>
  <!-- Other head elements -->
  <base href="/" />
</head>
```

After the module has been added to the application HTML, the needs to be set up in Angular. Assuming the application has a basic two-page structure, add the following route and `imports` to the app module.

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';

import { Page1Component } from "../components/page1.component";
import { Page2Component } from "../components/page2.component";

const routes: Routes = [
  {
    path: '',
    redirectTo: '/page1',
    pathMatch: 'full'
  },
  { path: 'page1', component: Page1Component},
  { path: 'page2', component: Page2Component}
];

@NgModule({
  imports: [
    /* .. */
    RouterModule.forRoot(routes)
  ],
  declarations: [
    /* .. */
    Page1Component,
    Page2Component
  ],
  bootstrap: [ /* .. */ ]
})
export class AppModule {
}
```

This code defines the routes and adds the router module all in one `import` command. The final step to get the router to function is to add the `<router-outlet>` to the root component view. The following code shows how to set up the router outlet and defines some basic navigation links to navigate between the different pages.

```
<nav>
  <a routerLink="/page1" routerLinkActive="active">Page 1</a>
  <a routerLink="/page2" routerLinkActive="active">Page 2</a>
</nav>
<div class="body">
  <router-outlet></router-outlet>
</div>
```

This example is a basic example of routing. Advanced concepts of routing, such as nested routes, are out of scope for this book. More information can be found online at the Angular documentation site using the link provided at the end of this section. The only remaining aspect of Angular routing that needs to be covered prior to building the SpyStore application is using route parameters. This example can be expanded to include a parameter for `page2` by changing the route to the following.

```
{ path: 'page2/:name', component: Page2Component}
```

The `:name` syntax tells the router to expect a parameter named `name`. In order to send a name to the page, the navigation links can be changed to the following.

```
<a routerLink="['page2', 'Bob']" routerLinkActive="active">Page 2 with Bob</a>
<a routerLink="['page2', 'Jim']" routerLinkActive="active">Page 2 with Jim</a>
```

By default, the route parameters are set up as observables, which is useful when the component is reused and allows the parameter to be changed over time. More information about component reuse can be found in the online documentation. A simpler example is to not reuse the component and use the snapshot mechanism to access the parameter. Assuming the `Page2Component` is not being reused, the following code could be used to access the parameter.

```
import {Component, OnInit} from "@angular/core";
import {ActivatedRoute} from '@angular/router';

@Component{/* ... */)
export class Page2Component implements OnInit {
  constructor(private _route: ActivatedRoute) { }

  ngOnInit() {
    let name = this._route.snapshot.params['name'];
  }
}
```

As noted previously, the new router in Angular 2 is extensive and is too broad to cover in this section. For more details on routing, see the Angular documentation at <https://angular.io/docs/ts/latest/guide/router.html> and <https://angular.io/docs/ts/latest/tutorial/toh-pt5.html>.

Building the SpyStore Angular App

Now that the core concepts of Angular have been covered, it is time to get back to building the SpyStore application and applying these concepts to a full application. The first step in getting back to the application is to add routing and set up the product catalog page to start displaying products.

To begin, locate the sample code provided with the book, open the `Start-Part2` code, before continuing with the remainder of the chapter. The sample code for `Part2` takes what was created from the first chapter and adds some additional styling and images in order to prepare the experience of the SpyStore application as it's being built.

■ **Note** When opening a new section of the example code, readers will need to run `npm install` in the command line of the project to download the required modules. In the case of this `Start-Part3` project, navigate to `<book-code-directory>\Chapter09-Angular2\Start-Part3\src\SpyStore.Angular2\` and run the `npm install` command there before proceeding.

Adding Routing

The following steps outline what's needed in order to add routing to the SpyStore application.

- Add a product component
- Create the route
- Add router and components to the app module
- Set up the router outlet in the app component

Adding a Product Component

Before setting up routing, you need a component to route to. Similar to the root component, which was set up earlier, you create a `products` component that will eventually display a list of products from the SpyStore API. The `products` component needs both a component logic class and a view.

■ **Note** The naming convention of `products.component.ts` is based on the Angular team style recommendations. The style guide can be found online at <https://angular.io/docs/ts/latest/guide/style-guide.html>.

All of the pages for the SpyStore application are going to go in to a new folder. Create the new folder under the `scripts` folder called `components`. Inside the new `components` folder, create a new TypeScript file called `products.component.ts` and include the following.

```
import { Component, OnInit } from "@angular/core";

@Component({
  templateUrl: "/app/components/products.html",
})
export class ProductsComponent implements OnInit {
  products: any[];
  header: string;

  constructor() { }

  ngOnInit() {
    this.header = "Featured Products";
    this.products = [];
  }
}
```

This class sets up the basic placeholders for the products that will be loaded from the SpyStore API. In addition, there is a `Component` decorator, which specifies the view. Next, create the HTML page for the view as noted by the `templateUrl` in the decorator.

Create a new folder in the `wwwroot/app` called `components`. Inside the new `components` folder, create a new HTML file called `products.html`. Inside the HTML file, include the following code.

```
<div class="jumbotron">
  <a routerLink="/products" class="btn btn-info btn-lg">View Tech Deals &raquo;</a>
</div>

<h3>{{header}}</h3>
```

This html uses the `routerLink` directive of the router and specifies the products page as its destination. This is only a link back to itself at this point, but will be used later when the product page loads different categories of products and allows searching. With the product component and template set up, the routes can be created and then added to the module.

Creating the Route

For the SpyStore app, all the routes are going to be defined in a separate module that will be imported into the app. To define the routes, create a new file in the `scripts` folder called `app.routing.ts` and include the following code.

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';

import { ProductsComponent } from "../components/products.component";
```

```

const routes: Routes = [
  {
    path: '',
    redirectTo: '/products',
    pathMatch: 'full'
  },
  { path: 'products', component: ProductsComponent }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }

export const routedComponents = [ProductsComponent];

```

The code contains a single class called the `AppRoutingModule`. This class has no logic; what it does have is an `NgModule` decorator that specifies the routing imports that are defined. This will eventually be imported into our `AppModule` and the `RouterModule`. That is part of the exports that will ultimately be imported into the root app module.

Inside the module imports is a call to the `RouterModule.forRoot` method. The only argument to that method is a reference to the `routes` const, which is where all the routing will be configured. The routing const is an array of path arguments and their declarations. The first declaration specified matches the root URL for the site as denoted by the empty string value for the path. This root path is set up to redirect to the products route, which is set up in the next element of the array. The products route defines the path of products and maps that to the `ProductsComponent` that was set up earlier and imported at the top of the file.

The final const in this code is to an export called `routedComponents`. This is just a convenience, because all routes are already imported in this file. It's easy enough to include them here so they can be imported into the `AppModule` file later with ease.

Adding the Router and Components to the App Module

Now that the view and the router have been created, they need to be added to the app module. The `app.module.ts` file needs to be updated to include the following code.

```

import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppRoutingModule, routedComponents } from './app.routing';

import { AppComponent } from './app.component';

@NgModule({
  imports: [
    BrowserModule,
    AppRoutingModule
  ],

```

```

    declarations: [
      AppComponent,
      routedComponents
    ],
    providers: [ ],
    bootstrap: [AppComponent]
  })
export class AppModule {
}

```

The main change here from the first `AppModule` is the addition of the `AppRoutingModule`, both in the `import` at the top of the file and in the `NgModule` decorator. Also note that the `routedComponents` array, which aggregated all the components, is included as part of the declarations. This is nice because it only has to be included with one `import` rather than including all the components for every page. At this point, there is a component to route to, the route has been defined, and the router itself has been included as part of the `AppModule`, making it accessible to the app. If the app were to run at this point, an error would appear stating `Error: Cannot find primary outlet to load ProductsComponent`. This is because there is no `router-outlet` defined in the `AppComponent`. This is covered next in the last step.

Setting Up the Router Outlet in the App Component

When the router module was imported previously, it imported both the configuration for the routes in the current app, but also the router components and directives that are a part of the router module. Included in that is a directive called `router-outlet`. This defines the primary outlet, which the error message eluded to previously. To add this, open the `wwwroot/app/app.html` file and locate the `panel-body` element that states `-- App Pages Go Here --`. Replace the placeholder text with the `<router-outlet></router-outlet>` directive. The new element should look like the following code.

```

<div class="panel-body">
  <router-outlet></router-outlet>
</div>

```

At this point, the app can be run and instead of a shell of an application, the content section of the app will be replaced with the heading and start of the first page in the `SpyStore` application. Before building out the remaining pages, the next section walks through connecting to the services and retrieving data and displaying it within the `ProductsComponent` that was just created.

Connecting to Services

In this section, the application will be set up to connect to the `SpyStore` Web API services that were created previously in [Chapter 3](#).

■ **Note** For convenience, the final code for the services project is included as part of the this chapter's example code. Throughout the remainder of this chapter, the `SpyStore` service will need to be running in order to get the demos to work. To run the service, open the `SpyStore.Service` folder from the example code for this application in the command line. In the command line, run the `run_development` command, which launches the `run_development.cmd` script to run the services.

Given that the web services will be used by multiple components, the best place to set up the code that connects to the web services is by using an Angular service class. In addition, the app module will need to have an `import` added for the `HttpModule`, which will be needed within our new service class. Before setting up the service, open the `app.module.ts` class and add the `HttpModule` as listed in the code.

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { HttpModule }    from '@angular/http';
import { AppRoutingModule, routedComponents } from './app.routing';

import { APP_CONFIG, SPYSTORE_CONFIG } from './app.config';

import { AppComponent } from './app.component';

@NgModule({
  imports: [
    BrowserModule,
    HttpModule,
    AppRoutingModule
  ],
  declarations: [
    AppComponent,
    routedComponents
  ],
  providers: [
    { provide: APP_CONFIG, useValue: SPYSTORE_CONFIG }
  ],
  bootstrap: [AppComponent]
})
export class AppModule {
}
```

Notice the addition of the `HttpModule` in both the `import` statement on the top and the `imports` configuration in the `NgModule` decorator.

There is also the addition of an application configuration, which is used to store the `apiEndpoint` URL. The application config is provided as a non-class dependency using the `useValue` parameter in the providers. The application configuration is made up of three parts, which are required to supply a value-based provider for dependency injection. More information on value providers can be found online at <https://angular.io/docs/ts/latest/guide/dependency-injection.html#non-class-dependencies>.

The values for the application configuration need to be set up in a new file. To do this, create a new file in the `scripts` folder called `app.config.ts`. In this file, add the following code.

```
import { OpaqueToken } from '@angular/core';

export interface AppConfig {
  apiEndpoint: string;
}
export let APP_CONFIG = new OpaqueToken('AppConfig');
export const SPYSTORE_CONFIG: AppConfig = {
  apiEndpoint: 'http://localhost:40001/api/'
};
```

The application configuration is made up of three parts. The first part is the interface. This interface provides typing constraints when injecting the configuration into future services. Next is an `OpaqueToken`, which defines a declaration for the provider type. This is required because TypeScript interfaces don't actually write out anything in JavaScript when it's transpiled. The final `const` is the value of the actual configuration. This is used in the `useValue` part of the providers section to define a constant value for the object whenever it's requested by the service or component.

Now that the app is capable of calling services over HTTP, the service can be set up. To do this based on the Angular style guide, create a new file called `product.service.ts`. In this file, add the following code.

```
import { Injectable, Inject } from '@angular/core';
import { Http, Response } from '@angular/http';
import { Observable, Observer } from 'rxjs/Rx';
import 'rxjs/Rx';
import 'rxjs/add/operator/map'

import { AppConfig, APP_CONFIG } from './app.config'

@Injectable()
export class ProductService {
  constructor(private http: Http, @Inject(APP_CONFIG) private config: AppConfig) { }

  getFeaturedProducts(): Observable<any> {
    return this.http.get(this.config.apiEndpoint + "product/featured").map(response =>
      response.json());
  }
}
```

This service class has a single method called `getFeaturedProducts`. This method uses the `HttpModule` that was imported to call the `SpyStore` service endpoint for the featured products. This call to the `Get` method returns an observable. The service itself doesn't do anything with that response except set up a pipe to map the response when the app calls it. This map code converts the JSON string returned by the service into an object. This map function can also be used to convert the response to a strongly typed object.

There are two other points to note about this class. First, the `AppConfig`, as noted earlier, is just an interface in TypeScript. When TypeScript transpiles to code, nothing for the interface is actually written out in JavaScript, which means that JavaScript doesn't know what an `AppConfig` is. In order to support dependency injection in this case, the class needs to tell Angular what to inject, by applying an `@Inject` decorator on the parameter and specifying the `OpaqueToken` declared earlier.

■ **Note** Angular 2 has adopted *observables* rather than promises as its primary mechanism for asynchronous activity. Observables are part of the RxJS Framework, which is based on the initial work of the Reactive Extensions Framework in .NET. The primary motivation for the moving to observables over promises was to support multiple retry scenarios and cancellation, which were lacking in the promise implementation in previous versions. Observable on the HTTP service in conjunction with observables on the `routeParams` side also offers some unique scenarios that allow for component reuse and can add to performance.

While observables are similar in syntax to promises, there are some key difference. The most common one that causes developers issues is the fact that observables do not kick off until an initial subscriber is registered, which means you cannot simple call a service method that returns an observable and expect a result. You have to subscribe to the result and handle it before any call will be made.

More information on observables can be found online at <https://angular.io/docs/ts/latest/tutorial/toh-pt6.html#!#observables>.

Now that the service is set up, the product component can be injected with the service and used to load its array of products. The following code shows the updated products component, using the new service.

```
import { Component, OnInit } from "@angular/core";
import { ProductService } from '../product.service';

@Component({
  templateUrl: "/app/components/products.html",
})
export class ProductsComponent implements OnInit {
  products: any[];
  header: string;

  constructor(private _service: ProductService) { }

  ngOnInit() {
    this.header = "Featured Products";
    this._service.getFeaturedProducts().subscribe(products =>
      this.products = products);
  }
}
```

Besides the addition to the constructor for dependency injection, the new code calls the service in the `ngOnInit()` function. In order to kick off the observable, the code subscribes to the observable and provides a response call in the first parameter. In the response call, the code stores the web service response in the `products` array of the component. Now that the products are populated with actual products from the service, the view can be modified to display them. The following code shows the new products view.

```
<div class="jumbotron">
  <a routerLink="/products" class="btn btn-info btn-lg">View Tech Deals &raquo;</a>
</div>

<h3>{{header}}</h3>

<div class="row">
  <div class="col-xs-6 col-sm-4 col-md-3" *ngFor="let product of products">
    <div class="product">
      <img [src]='images/' + product.ProductImage" />
      <div class="price">{{product.CurrentPrice | currency:'USD':true:'1.2-2'}}</div>
      <div class="title-container">
        <h5>{{product.ModelName}}</h5>
      </div>
    </div>
  </div>
</div>
```

```

<div class="model-number"><span class="text-muted">Model Number:</span>
  {{product.ModelNumber}}</div>
<a [routerLink]="['/products', product.CategoryId]" class="category">
  {{product.CategoryName}}</a>
<a [routerLink]="['/product', product.Id]" class="btn btn-primary btn-cart">
  <span class="glyphicon glyphicon-shopping-cart"></span> Add to Cart</a>
</div>
</div>
</div>

```

This new element, with the class `row` on the bottom includes a loop of product listing cards, shown in Figure 8-6. The looping occurs because of the Angular call to `*ngFor`, which specifies the array of products on the component and the variable `let product`, which is populated with each element in the array and used in the child elements.



Figure 8-6. Product card on the product catalog page after loading from the services

Inside the child elements the product's contents are bound using the interpolation that was described earlier and is a shorthand to writing out the contents to the DOM. Under the price section, the interpolation call also uses a pipe in Angular to specify that the number in the property `CurrentPrice` should be formatted as currency. More information on pipes can be found at <https://angular.io/docs/ts/latest/guide/pipes.html>.

In addition to the interpolation, there is a `[routerLink]` attribute defined on the links in the bottom, which will be used in the next section. This attribute defines router links for the products page and product details page, which leverage route parameters. The app can be run now, and assuming the service is running, it will display a list of products. Clicking the links at this point will throw an error message stating `Cannot match any routes`.

The next section covers route parameters and how to use them to vary the resulting products displayed on the products page.

Route Parameters

Before beginning this section, open `Start-Part3` in your example code. This code differs from the code listed in the previous section and adds a new `CategoryLinksComponent`, which leverages the product service to pull a list of all categories and displays the list of categories along the top, as shown in Figure 8-7. In addition, the product service has been expanded to include other calls to the API needed for the application for getting products, categories, and product details. Two new API services have been added to support user management and shopping cart management with the API. This version also includes a `LoggingService` and `AppErrorHandler`, which provide simple logging and global error handling as part of the dependency injection framework.

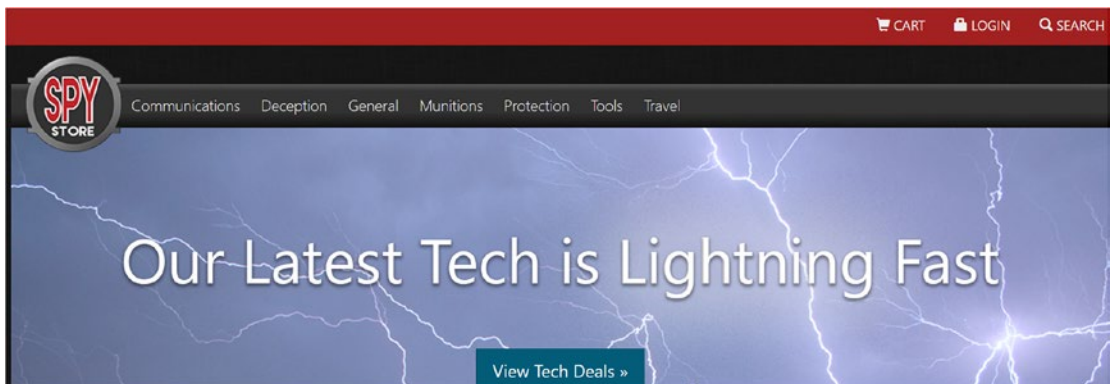


Figure 8-7. Category links across the top of the page loaded from the categories service endpoint

In this section, the SpyStore application will be modified to use the route parameters and load products for a selected category. The sample code for this part already includes the API calls in the product service that are needed by this section. This list highlights the steps needed to update the app.

- Add the parameterized route to the router
- Update the product component to handle the route parameter

In order to add the parameterized route to the router, open the `app.routing.ts` file and add the following line to the routes const.

```
{ path: 'products/:categoryId', component: ProductsComponent }
```

Finally, to update the product component to handle the route parameter, open `products.component.ts` and update it to the following.

```
import {Component, OnInit} from "@angular/core";
import { ActivatedRoute, Params } from '@angular/router';
import { ProductService } from '../product.service';
import { LoggingService } from '../logging.service';

@Component({
  templateUrl: "/app/components/products.html",
})
export class ProductsComponent implements OnInit {
```

```

products: any[];
  header: string;

constructor(
  private _route: ActivatedRoute,
  private _service: ProductService,
  private _loggingService: LoggingService) { }

ngOnInit() {
  this._route.params.subscribe(params => {
    if ("categoryId" in params) {
      let categoryId: number = +params["categoryId"];
      this._service.getCategory(categoryId).subscribe(category =>
        this.header = category.CategoryName, err => this._loggingService.
          logError("Error Loading Category", err));
      this._service.getProductsForACategory(categoryId).subscribe(products =>
        this.products = products, err => this._loggingService.logError("Error
          Loading Products", err));
    } else {
      this._service.getFeaturedProducts().subscribe(products => {
        this.header = "Featured Products";
        this.products = products
      }, err => this._loggingService.logError("Error Loading Featured Products", err));
    }
  });
}
}

```

In this code, the constructor was updated to include the `ActivatedRoute` service, which will be used in the `ngOnInit()`. The `ngOnInit` method has been updated as well. When the component initializes, instead of calling the product service directly, the code now subscribes to events on the param's observable. As noted in the HTTP section previously, route parameters use observables, which are part of the RxJS framework. More information can be found at <https://angular.io/docs/ts/latest/guide/router.html#route-parameters>. The subscribe method contains one parameter, which is the callback function that is fired whenever the route parameters change. This is helpful, because it allows the route to change without forcing the component to reload. This can be seen in action when navigating to different categories and using the back button in the browser. In the callback function there is a conditional statement, which checks for the presence of a `categoryId` from the `routeParams`, if one is present, then it loads the category name and category specific products. Otherwise, it loads the featured products from the previous example.

At this point, running the app and clicking on a category in the menu will change the list of products. Clicking the link in the jumbotron, or the `SpyStore` watch icon, will revert back to the Featured Products list.

Search Page

The final user experience that needs to be added to complete the product listing is search. The search textbox and button are located in the top right of the menu after clicking on the search magnifying glass, as shown in Figure 8-8.

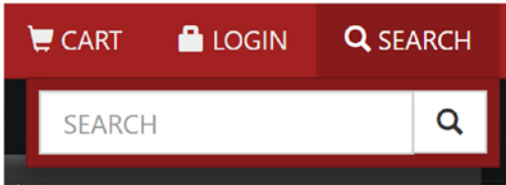


Figure 8-8. Clicking on the search button opens the search textbox

Given that the search textbox and button are located in the header, changes to this feature will need to be made to the app component and to the products component. The following list details the changes needed to implement the search feature.

- Add FormsModule to the project
- Add model binding to the app component view
- Add event handlers for the search event to the app component
- Add support for the search queryParams to the products component

One of the core new concepts in this feature that has not been introduced in previous features is two-way data binding. The search features uses two-way data binding with `ngModel` on the search textbox. In order to use `ngModel`, the app needs to import the `FormsModule`. To do this, update the `app.module.ts` file to include the `FormsModule` and add it to the imports.

```
import { NgModule, ErrorHandler } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { FormsModule } from '@angular/forms';
// Other imports omitted for brevity

@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    // other imports omitted
  ],
  // other module config omitted
})
export class AppModule {
}
```

Once the `FormsModule` is added, the app component can be set up to bind the textbox to a property on the app component. Before we can bind anything, open the `app.component.ts` file and add the following field to the top of the `AppComponent` class.

```
searchText: string;
```

Now, in the `app.html` in the `wwwroot/app` folder, locate the search textbox and change it so it binds the field to the textbox, as shown in the following code.

```
<input type="text" id="searchString" name="searchString" [(ngModel)]="searchText"
class="form-control" placeholder="SEARCH">
```

Notice the new attribute in the textbox with the two-way binding, or banana in a box, syntax. While the `app.html` component view is open, add the click event handler to the search button, which will be set up next in the `app.component.ts` file.

```
<button class="btn btn-default" type="button" (click)="search()">
```

In this button, the click event is bound to a method on the component called `search`. Let's go back to the `app.component.ts` file and implement the search method. The following code shows the new `app.component.ts` file after the search method has been added.

```
import { Component, OnInit } from "@angular/core";
import { Router, NavigationExtras } from "@angular/router";

@Component({
  selector: "spystore-app",
  templateUrl: "/app/app.html"
})
export class AppComponent implements OnInit {
  searchText: string;

  constructor(private _router: Router) {
  }

  ngOnInit() {
  }

  search() {
    let navigationExtras: NavigationExtras = {
      queryParams: { 'searchText': this.searchText }
    };

    this._router.navigate(['/products'], navigationExtras);
  }
}
```

In order to handle search, the app component injects the `Router` service and uses the `NavigationExtras` object to tell the router to navigate to the products page with the `searchText` as a `querystring` parameter, also called `searchText`. The products page will handle this `querystring` parameter.

In the `products.component.ts` file, update the `ngOnInit` method to the following code.

```
ngOnInit() {
  this._route.params.subscribe(params => {
    if ("categoryId" in params) {
      let categoryId: number = +params["categoryId"];
      this._service.getCategory(categoryId).subscribe(category =>
        this.header = category.CategoryName,
        err => this._loggingService.logError("Error Loading
        Category", err));
      this._service.getProductsForACategory(categoryId).subscribe(products =>
        this.products = products,
        err => this._loggingService.logError("Error Loading
        Products", err));
    } else if (!("searchText" in this._route.snapshot.queryParams)) {
      this.getFeaturedProducts();
    }
  });

  this._route.queryParams.subscribe(params => {
    if ("searchText" in params) {
      let searchText: string = params["searchText"];
      this.header = "Search for: " + searchText;
      this._service.getSearchProducts(searchText).subscribe(products =>
        this.products = products,
        err => this._loggingService.logError("Error Loading
        Products", err));
    } else if (!("categoryId" in this._route.snapshot.params)) {
      this.getFeaturedProducts();
    }
  });
}

getFeaturedProducts() {
  this._service.getFeaturedProducts().subscribe(products => {
    this.header = "Featured Products";
    this.products = products
  }, err => this._loggingService.logError("Error Loading Featured Products", err));
}
```

Note that this method now includes two subscription methods instead of one. The first one was already shown and handles the `categoryId` route parameter. The second one subscribes to the `queryParams` observable, which, similarly to the route parameters, updates every time the query string is changes on the route. The conditional of this method also loads `getFeaturedProducts` and, in an effort to not duplicate code, this logic has been extracted into a reusable method called `getFeaturedProducts()`. Also note that given the route parameters and query parameters by cause race conditions among each other, there is an extra conditional check on the else statements that checks for the existence of either the `searchText` or `categoryId` depending on the case. Since these checks are performed at a point in time rather than against the observable, the code checks these parameters using the `_route.snapshot` property, which offers a synchronous snapshot of the parameters at the current moment.

At this point the feature is complete. Run the app and test the products by category and also with search. Try searching for products with the word device and the list should display twelve products matching that search text. The header should also be updated to reflect searching for device.

Product Details Page

Now that the product catalog has been implemented in all its forms, from featured, to category, and lastly search. It is time to move on to the product details page. The product details page contains additional information about the product and allows for adding the product to the shopping cart. The product details page requires the following changes to be implemented.

- Set up the ProductDetails component
- Implement the CartService Add to Cart method
- Set up the product detail route

Setting Up the ProductDetails Component

The product details component is new to the application. In order to begin, create a new file in the `scripts/components` folder called `product.component.ts`. Inside the `product.component.ts` file, include the following code.

```
import {Component, OnInit} from "@angular/core";
import {ActivatedRoute, Params, Router, RouterLink} from '@angular/router';
import {ProductService} from '../product.service';
import {CartService} from '../cart.service';
import {LoggingService} from '../logging.service';
import {UserService, User} from '../user.service';

@Component({
  templateUrl: "app/components/productDetail.html"
})
export class ProductDetailComponent implements OnInit {
  message: string;
  isAuthenticated: Boolean;
  product: any;
  quantity: any = 1;

  constructor(
    private _router: Router,
    private _route: ActivatedRoute,
    private _productService: ProductService,
    private _cartService: CartService,
    private _userService: UserService,
    private _loggingService: LoggingService) {
    this.product = {};
    this.isAuthenticated = this._userService.isAuthenticated;
  }
}
```

```

ngOnInit() {
  this.isAuthenticated = this._userService.IsAuthenticated;
  this._userService.ensureAuthenticated().subscribe(_ =>
    this.isAuthenticated = true);

  this._route.params.subscribe((params: Params) => {
    let id: number = +params['id'];

    this._productService.getProduct(id).subscribe(product =>
      this.product = product, err => this._loggingService.logError(
        "Error Loading Product", err));
  });
}

addToCart() {
  this._cartService.addToCart({
    ProductId: this.product.Id,
    Quantity: this.quantity
  }).subscribe((response) => {
    if (response.status == 201) {
      this._router.navigate(['/cart']);
    }
    else {
      this._loggingService.log(response.statusText);
    }
  }, err => this._loggingService.logError("Error Adding Cart Item", err));
}
}

```

The product detail component's main responsibility is to load the specified product from the API. Later, when setting up the route, the specified product ID will be provided as part of the route parameters. To access the specified product ID, the route params observable will be used.

In addition, the product details component ensures that the user is authenticated, which will be needed later in order to add items to the shopping cart.

■ **Note** The authentication in the app is intentionally simplified. Authentication is notoriously complex and in an effort to focus on the features of the various single page apps, it has been removed from the scope of this chapter. When building authentication in production apps, consider authentication cautiously and verify with the appropriate security advisors in your organization.

The other method in addition to the `ngOnInit()` method is an event handler for the Add to Cart button, which is the only button on the page. The cart service POSTs a shopping cart record to the API in order to add the item to the shopping cart. In addition to the product ID, the shopping cart record also consists of a quantity field, which is provided by the view using the `ngModel` attribute of the forms module that was demonstrated previously.

The product detail component also contains a `templateUrl` as part of its component decorator. The following code shows the associated product detail template.

```
<h1 class="visible-xs">{{product.ModelName}}</h1>
<div class="row product-details-container">
  <div class="col-sm-6 product-images">
    <img [src]='images/' + product.ProductImageLarge" />
    <div class="key-label">PRODUCT IMAGES</div>
  </div>
  <div class="col-sm-6">
    <h1 class="hidden-xs">{{product.ModelName}}</h1>
    <div class="price-label">PRICE:</div>
    <div class="price">{{product.CurrentPrice | currency:'USD':true:'1.2-2'}}</div>
    <div class="units">Only {{product.UnitsInStock}} left.</div>
    <div class="product-description" [textContent]="product.Description">
    </div>
    <ul class="product-details">
      <li>
        <div class="key-label">MODEL NUMBER:</div> N100Z1
      </li>
      <li>
        <div class="key-label">CATEGORY:</div> <a [routerLink]="['/products',
        product.CategoryId]">{{product.CategoryName}}</a>
      </li>
    </ul>

    <div *ngIf="isAuthenticated" class="row cart-group">
      <label for="qty">QUANTITY:</label>
      <input type="text" name="qty" [(ngModel)]="quantity" class="cart-quantity form-
      control" />
      <button (click)="addToCart()" class="btn btn-primary">Add to Cart</button>
    </div>
    <a routerLink="/products">Back to List</a>
  </div>
</div>
```

The template for the product detail component contains a number of different binding techniques, which were all described previously as part of the templating section. At the bottom of the view, there is a `div` element. This element will only display after the user has been authenticated as noted by the `*ngIf="isAuthenticated"` directive. Inside this element, there is a quantity textbox that uses the `ngModel` binding to map to the quantity field on the component. There is also a `(click)` event on the Add to Cart button, which maps to the `addToCart()` method on the component.

CartService Add To Cart

The `addToCart()` method on the product detail component calls the `addToCart` method on the `cartService`, which was provided as part of the code for this section. The following code shows the POST code for the API from the `cart.service.ts` class in the `scripts` folder.

```

import {Injectable, Inject} from '@angular/core';
import {Http, Headers, Response} from '@angular/http';
import {Observable, Observer} from 'rxjs/Rx';
import { UserService, User } from './user.service';
import { APP_CONFIG, AppConfig } from './app.config';
import 'rxjs/Rx';
import 'rxjs/add/operator/map'

// code omitted for brevity

export interface ShoppingCartRecord {
  Id?: number;
  CustomerId?: number;
  ProductId: number;
  Quantity: number;
  TimeStamp?: any;
  CurrentPrice?: number;
  LineItemTotal?: number;
}

@Injectable()
export class CartService {

  constructor(private http: Http, private _userService: UserService, @Inject(APP_CONFIG)
    private config: AppConfig) { }

  addToCart(cartRecord: ShoppingCartRecord): Observable<Response> {
    var headers = new Headers();
    headers.append('Content-Type', 'application/json');
    return this.http.post(this.config.apiEndpoint + "shoppingcart/" + this._userService.
      User.Id, JSON.stringify(cartRecord), { headers: headers });
  }

  // other methods omitted for brevity
}

```

The `addToCart` method on the `CartService` is not that different from the other API calls that have been shown previously. The main difference is that the method being called on the `http` service is called `post` as opposed to `get`. This sends the HTTP request with the `Post` method instead of the `Get` method. In addition, since the WebAPI service is a REST service, the data that is posted needs to be in a JSON format. While the call to `JSON.stringify` handles the actual serialization, the ASP.NET WebAPI service does not automatically detect that format and instead uses the `content-type` header of the request to determine the format. In order to tell the WebAPI service that the request is coming in as JSON format, the code creates a new `Headers` object and adds the `Content-Type` header with the value of `application/json`.

Setting Up the Product Detail Route

The final step in order to get the product detail component working is to add it to the route. Open the `app.router.ts` file and add the appropriate imports, routes, and declarations to the `routerComponents` array.

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';

import { ProductsComponent } from "../components/products.component";
import { ProductDetailComponent } from "../components/productDetail.component";

const routes: Routes = [
  {
    path: '',
    redirectTo: '/products',
    pathMatch: 'full'
  },
  { path: 'products', component: ProductsComponent },
  { path: 'products/:categoryId', component: ProductsComponent },
  { path: 'product/:id', component: ProductDetailComponent }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }

export const routedComponents = [ ProductsComponent, ProductDetailComponent ];
```

The route for the product detail component is defined as `/product/:id` where `:id` represents the route parameter for the product ID. The router link to access this page was set up previously in the products template when the list of products was added.

Running the app at this point will display a list of the products on the home page. After clicking Add to Cart for a specific product, the product detail page will load and display the details for the product. At this point, the `addToCart` event on the product detail component is set up and adds the item via the API, but there is no route set up. Therefore, an error will appear when trying to navigate to the cart page, which will be set up next.

Cart Page

After adding the product to the API, the product details component redirects to the cart page. In this section the `CartComponent` will be set up. One of the biggest differences between this component and others is that the cart page contains a nested component for each cart record. Figure 8-9 shows a representation of the component tree for the cart page.

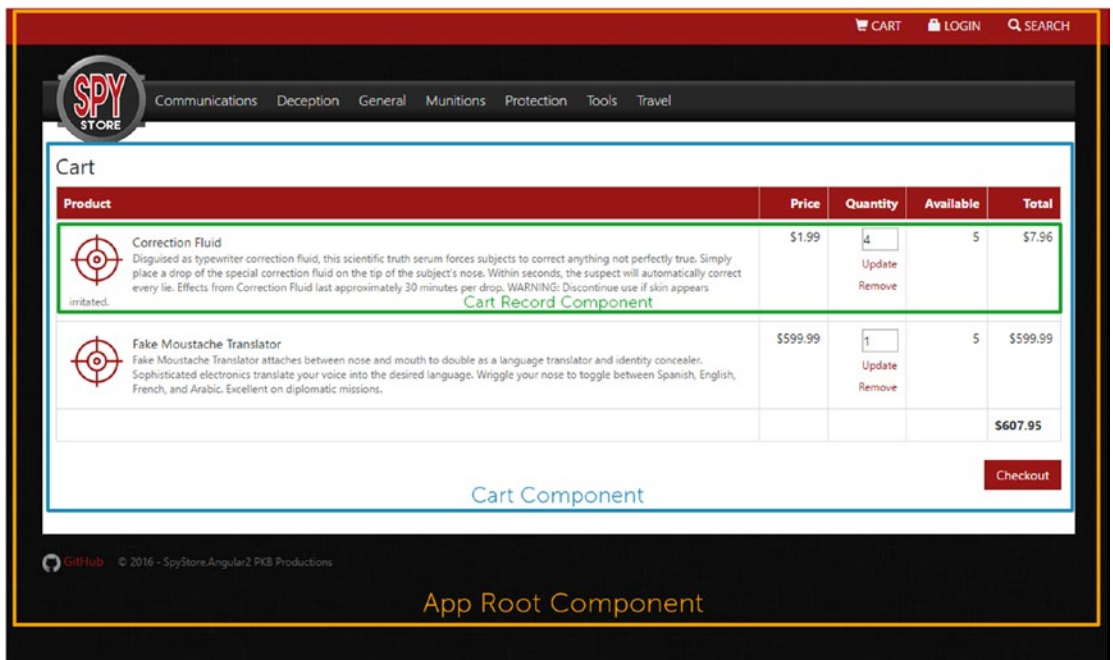


Figure 8-9. Annotated cart page with components

The following list describes the steps needed to implement the cart page.

- Create the Cart component
- Create the Cart Record component
- Create the Cart route
- Add the Cart Record component to the app module

Creating the Cart Component

In order to create the cart component, add a new file to the `scripts/components` folder called `cart.component.ts`. The following code is included in the cart component.

```
import {Component, OnInit} from "@angular/core";
import {Cart, ShoppingCartRecord, CartService} from '../cart.service';
import {CartRecordComponent} from './cartRecord.component'

@Component({
  templateUrl: "app/components/cart.html"
})
export class CartComponent implements OnInit {
  cart: Cart;

  constructor(private _cartService: CartService) { }
```

```

ngOnInit() {
  this._cartService.getCart().subscribe(cart =>
    this.cart = cart);
}

onCartRecordRemoved(record: ShoppingCartRecord) {
  var index :number = this.cart.CartRecords.indexOf(record, 0);
  if (index > -1) {
    this.cart.CartRecords.splice(index, 1);
  }
}
}

```

Because the cart page is architected to contain much of the update and removal of items as part of the nested cart record component, the logic for the cart component is fairly simple. In the `ngOnInit()` method, the cart is loaded from the API. Luckily, this is the only location where the cart is used so loading it here directly from the service is acceptable. The only other method in this component is for `onCartRecordRemoved`. This event is emitted from the cart record component and will be used later when we implement the cart record component. This method finds the record inside the cart and removes it from the `CartRecords` array using the JavaScript `splice` method.

Once the cart component is created, it is time to set up the cart view. The following code shows the HTML for the cart view.

```

<h3>Cart</h3>

<div class="table-responsive">
  <table class="table table-bordered product-table">
    <thead>
      <tr>
        <th style="width: 70%;">Product</th>
        <th class="text-right">Price</th>
        <th class="text-right">Quantity</th>
        <th class="text-right">Available</th>
        <th class="text-right">Total</th>
      </tr>
    </thead>
    <tbody>
      <tr cart-record [record]="record" (onRecordRemoved)="onCartRecordRemoved($event)"
        *ngFor="let record of cart?.CartRecords">
      </tr>
    </tbody>
    <tfoot>
      <tr>
        <th>&nbsp;</th>
        <th>&nbsp;</th>
        <th>&nbsp;</th>
        <th>&nbsp;</th>
        <th>{{cart?.CartTotal | currency:'USD':true:'1.2-2'}}</th>
      </tr>
    </tfoot>
  </table>
</div>

```



```
<div class="pull-right">
  <button routerLink="/checkout" class="btn btn-primary">Checkout</button>
</div>
```

Similar to the cart component class, the HTML is fairly simple. Most of the logic here is handled by the cart record component and is included in the `tbody` where the Angular template is set up to loop over the cart records, using `*ngFor` and load the `cart-record` component for each shopping cart record in the array. The cart record has two elements bound to it that will be shown later—the record is bound to the cart record component and the `onRecordRemoved` event is bound to the `onCartRecordRemoved` event that was set up previously.

Creating the Cart Record Component

If the app were launched at this point, an error would be thrown. In order for this cart view to work, there needs to be a cart record component set up. The following code shows the cart record component class.

```
import {Component, Input, Output, EventEmitter} from "@angular/core";
import {CartService, ShoppingCartRecord} from '../cart.service';
import {LoggingService} from '../logging.service';

@Component({
  selector: "[cart-record]",
  templateUrl: "app/components/cartRecord.html",
})
export class CartRecordComponent {
  @Input() record: ShoppingCartRecord;
  @Output() onRecordRemoved = new EventEmitter<ShoppingCartRecord>();

  constructor(private _cartService: CartService, private _loggingService: LoggingService) { }

  updateItem() {
    this._cartService.updateCartRecord(this.record).subscribe((response) => {
      if (response.status == 201) {
        this.record.LineItemTotal = this.record.Quantity * this.record.CurrentPrice;
        this.record.Timestamp = response.json().Timestamp;
      }
    }, err => this._loggingService.logError("Error Updating Cart Item", err),
    () => console.log('Update Complete'));
  }

  removeItem() {
    this._cartService.removeCartRecord(this.record).subscribe((response) => {
      if (response.status == 204) {
        this.onRecordRemoved.emit(this.record);
      }
    }, err => this._loggingService.logError("Error Deleting Cart Item", err),
    () => console.log('Delete Complete'));
  }
}
```

Starting with the component decorator, there is a `templateUrl`, just like the other components. There is also a selector component; notice the square brackets around the selector, which is how Angular defines an attribute selector. The reason for using the attribute selector is to maintain the integrity of the `tr` tags and the table structure from the DOM perspective.

The next thing to notice in the component class is the fields on the class have `Input()` and `Output()` decorators on them. Given that every new component has its own isolated scope, the `cartRecord` component cannot directly access the record object on the parent component. In order to display information about the record, that object needs to be passed into the nested component. The `Input()` directive on the `cart record` tells Angular that the parent component can specify, typically via binding, the card record as an attribute binding to the `cart record` component. The `Output()` decorator on the other hand tells Angular that the parent component can specify event handlers for the event specified. The type of the field for the `onRecordRemoved` event is an `EventEmitter`, which is a built-in type for Angular that provides the `emit` method and allows the nested component to broadcast the event when needed.

In addition to the constructor, which is used for dependency injection, there are two methods on the `cart record` component. These two methods are used to handle actions on the component, which will be bound in the template. The first method is for updating the cart. The method calls the `updateCart` method on the `CartService`, which in turn sends a `PUT` request to the API for updating the cart. Upon successful completion of the call, the callback updates the record so it is current with the backend as opposed to reloading the record, or cart, from the API. The second method is similar but it calls the `removeCartRecord` method on the `CartService`, which in turn sends a `DELETE` request to the API to remove the cart record. Upon successful completion of that method, the callback emits the `onRecordRemoved` event, which notifies the `cart` component to remove the cart record from the table in the UI.

Since the `tr` tag is already created as part of the `cart` component, the contents of the `cart` component template start with the `td` for the cart record. The following code shows the template for the cart record.

```
<td>
  <div class="product-cell-detail">
    
    <a [routerLink]="['/product', record.ProductId]" class="h5">{{record.ModelName}}</a>
    <div class="small text-muted hidden-xs">{{record.Description}}</div>
  </div>
</td>
<td class="text-right">{{record.CurrentPrice | currency:'USD':true:'1.2-2'}}</td>
<td class="text-right cart-quantity-row">
  <input type="number" [(ngModel)]="record.Quantity" class="cart-quantity" />
  <button class="btn btn-link btn-sm" (click)="updateItem()">Update</button>
  <button class="btn btn-link btn-sm" (click)="removeItem()">Remove</button>
</td>
<td class="text-right">{{record.UnitsInStock }}</td>
<td class="text-right">{{record.LineItemTotal | currency:'USD':true:'1.2-2'}}</td>
```

There is nothing new in this template; it consists of interpolation as well as attribute binding in order to interact with the `cart record` component. From a scoping perspective, it's important to remember that the `cart record` template can access properties and methods only on the `cart record` component and cannot access other properties or methods outside of its scope.

Creating the Cart Route

Now that both of the cart component and the cart record component have been created, the cart route can be added to the `app.routing.ts` file. As shown previously, open the app routing class and add the following code.

```
// other imports omitted for brevity
import { CartComponent } from "../components/cart.component";

const routes: Routes = [
  // other routes omitted for brevity
  { path: 'cart', component: CartComponent }
];
// routing module omitted for brevity
export const routedComponents = [ ProductsComponent, ProductDetailComponent, CartComponent ];
```

Adding the Cart Record Component to the App Module

While the cart component is imported and defined as part of the routing, the cart record component is not. Angular has no way of knowing that the cart record component class should be added to the module. To add the cart record component to the module, it needs to be included as part of the declarations in the `app.module.ts` file. The following code shows the cart record added to the declarations of the app module class.

```
// other imports removed for brevity
import { CartRecordComponent } from "../components/cartRecord.component";

@NgModule({
  // other module decorator config removed for brevity
  declarations: [
    AppComponent,
    CategoryLinksComponent,
    CartRecordComponent,
    routedComponents
  ],
})
export class AppModule {
}
```

Running the application now will result in mostly complete shopping experience including browsing and searching products, displaying product details, and manipulating the shopping cart. The final feature that will be implemented in this example is checkout. The next section walks through creating the checkout process.

Checkout

The checkout process is intentionally simplified for the SpyStore application. A real e-commerce site requires a significant amount of logic, which would be repetitive to demonstrate in this chapter. For the purposes of the SpyStore application, the checkout process involves setting up a checkout page that we linked the Checkout button to on the cart page, to contain a button which binds to an event on the checkout component to complete the order. The following steps outlines what is needed to set up the checkout page.

- Create the Checkout component
- Add a route for the Checkout component

Creating the Checkout Component

Similarly to other components, implementing the checkout component begins by creating a new checkout.component.ts file in the scripts/components folder. The following code shows the code for the checkout component.

```
import { Component } from "@angular/core";
import { Router } from '@angular/router';
import { CartService } from '../cart.service';
import { LoggingService } from '../logging.service';

@Component({
  templateUrl: "app/components/checkout.html"
})
export class CheckoutComponent {

  constructor(private _cartService: CartService,
               private _router: Router,
               private _loggingService: LoggingService) { }

  checkout() {
    this._cartService.buy().subscribe((response) => {
      if (response.status == 201) {
        this._router.navigate(['/products']);
      }
      else {
        this._loggingService.log(response.statusText);
      }
    }, err => this._loggingService.logError("Error Checking out", err));
  }
}
```

The checkout component contains a single method called checkout. This method calls the buy() method on the cart service, which in turn POSTs to the API and completes the purchase.

The following code shows the template for the checkout component.

```
<div class="row top-row">
  <div class="col-sm-offset-3 col-sm-6">
    <div class="panel panel-primary checkout-panel">
      <div class="panel-heading">
        <h1 class="panel-title">Checkout</h1>
      </div>
      <div class="panel-body">
        <h3>Checkout automatically as the logged in user</h3>
        <div class="text-muted">(User typically needs to enter billing and
          shipping info here)</div>
        <button (click)="checkout()" class="btn btn-primary">Auto-Checkout</button>
      </div>
    </div>
  </div>
</div>
```

Again, given that the checkout process for the SpyStore application is simplified, there is not much to this view. The view contains one event binding for the click event of the button. The click event maps to the checkout method on the component, which was described earlier.

Adding a Route for the Checkout Component

The final step for the checkout process is to add the checkout component to the route. The following code shows the code that is added to the `app.routing.ts` file to support checkout.

```
// other imports omitted for brevity
import { CheckoutComponent } from "../components/checkout.component";

const routes: Routes = [
  // other routes omitted for brevity
  { path: 'checkout', component: CheckoutComponent }
];

// routing module omitted for brevity

export const routedComponents = [ProductsComponent, ProductDetailComponent, CartComponent,
CheckoutComponent ];
```

At this point, the SpyStore application has the basic functionality that shows a concrete example of working with Angular 2. The app can be run along with the services and the SpyStore application should work as expected.

Summary

This chapter was an introduction to using Angular 2 to build applications. The remaining chapters cover other JavaScript frameworks. Angular has gained an impressive amount of popularity in recent years, and the team at Google has leveraged that popularity to create a next generation framework with Angular 2. It uses many of the recent and upcoming specifications in the web space and makes developing application simple and straightforward. Angular 2 is built with TypeScript in mind, which adds type safety to your code and protects against some nasty JavaScript side effects. Overall, Angular 2 is a very prescriptive framework, and it helps developers be successful early on by providing a very productive framework to use.

There are more features that were out of the scope of this chapter. Luckily, the Angular team has excellent documentation, thanks to the hard work of the team. For more information on Angular see the online documentation at <https://angular.io>. In addition, the community is very strong and leverages StackOverflow and GitHub openly to provide feedback to developers as they progress on their journey through learning Angular.

CHAPTER 9



React

In this chapter, we demonstrate an implementation of the basic SpyStore application using the React framework from Facebook. As with the previous chapter on Angular 2, we focus on highlighting the key aspects of React and how they can be utilized to develop powerful web interfaces.

There are numerous books and other resources focused solely on React and, while we hope this chapter will serve as a great introduction to this framework, it should not be considered an exhaustive guide to all the nuances and capabilities of this robust platform. Our goal is to provide a reference sample of the framework that can be compared to the other frameworks used throughout this book. Much of the core architecture of the SpyStore implementation in React is very close or the same as that of the previous Angular implementation. Both React and Angular are very powerful frontend frameworks for web development and, while they take different approaches to achieve this goal, they also work in much the same manner and do have many similarities. We will not spend much time directly comparing Angular to React but after reviewing the same SpyStore solution from the previously demonstrated Angular implementation and the React implementation here, you will be able to see for yourself some of the key ways these frameworks are similar and some of the implementation choices that make them different.

Many developers feel passionately that either Angular or React (or straight MVC) is the best choice for all their web frontend needs. The reality is that both Angular and React are very powerful; they have huge community support and enjoy a widespread adoption by developers working on sites of all sizes. They both have their flagship backers and you will find access to training and other resources for both frameworks very easy to find.

As with Angular, React is a constantly evolving framework and all code samples and content in this chapter are based on the current version at the time of this writing (15.4.2).

Solution Overview

Before we dive into the React implementation of the SpyStore application, it is worth taking a moment to discuss some of the key choices we will make in the development of the SpyStore solution in React. Some of these choices will make our solution very much like other framework examples in this book (making it easier to compare the various implementations), while other aspects of our solution and development workflow will be much different than you will see in other chapters.

First, in this chapter, the user interface we are going to be implementing via React is the same SpyStore interface previously developed in HTML and Bootstrap. All the core HTML and style assets from previous chapters will be utilized here and we will just copy any necessary static assets into our solution as necessary.

Next, when developing our SpyStore React solution, we will again utilize the TypeScript language for implementing the SpyStore React frontend, although many React developers still use raw JavaScript or ES6. Unlike Angular 2, which was natively developed in the TypeScript language, React has enjoyed huge success as a straight JavaScript library (and is not natively written in TypeScript). This fact bears little on our choice to use TypeScript as React has some very nice TypeScript type definition files and the tooling for developing React solutions with TypeScript makes for a very powerful match between this language and library.

We will again be taking advantage of the previously developed SpyStore API solution. All code developed in this chapter will assume that the SpyStore API is running and listening on port 40001.

■ **Note** For convenience, the final code for the services project is included as part of this chapter's example code. Throughout the remainder of this chapter, the SpyStore service will need to be running to get the demos to work. To run the service, open the `SpyStore.Service` folder from the example code for this application in the command line. In the command line, run the `run_development` command, which launches the `run_development.cmd` script to run the services.

We will continue to use NPM as our package manager to efficiently gather remote assets into our solution. Along with NPM, we will be using a popular module bundling tool called WebPack (<https://webpack.github.io/>). Chapter 06 provides a brief introduction to WebPack and, in this chapter, we will build on those concepts to build a richer development process. The previous chapter on Angular took advantage of Bower and Gulp to manipulate local assets and code as part of its development workflow. For the React solution we are going to demonstrate using WebPack to perform many of these same tasks. The choice of WebPack also affects how we develop web frontends and how our code and other assets are bundled and deployed, both during development and when preparing to publish to other environments. In this chapter, we demonstrate many of these concepts and show how WebPack can be leveraged to quickly set up a very efficient development workflow.

Our primary development tool will again be Visual Studio 2017 but the overall development workflow and tooling we will put in place will largely exist outside of Visual Studio. As you will see, WebPack will be utilized to compile all our client side TypeScript into JavaScript. In previous chapters this was done directly in Visual Studio. WebPack will also play a key role in the develop and debug cycles and it will be responsible for spinning up a web server and hosting our assets during development. Due to these facts, the React implementation of the SpyStore UI will rely less on Visual Studio's internal capabilities and will use the IDE more for its editing features than its full debug stack.

We will also only be utilizing the ASP.NET Core infrastructure to serve up our static content (in addition to being the stack responsible for all data access and API development via the solution implemented in Chapter 3). This is not necessarily a good or a bad thing and is done simply to demonstrate an alternative developer workflow for web interfaces. Many developers choose to do their server side C# development in Visual Studio and utilize an alternative editor such as Visual Studio Code or JetBrains's WebStorm to work with their client side assets. The workflow we will be demonstrating in this chapter will show how easy this can be.

Creating a New Visual Studio Core Project

The initial steps in setting up the project for our React solution will be identical to that of the previous chapter on Angular. As mentioned, our React project will rely on the ASP.NET Core runtime simply as a means of serving up our static files. To get started, open Visual Studio and create a new project. In the New Project dialog, shown in Figure 9-1, select a new ASP.NET Core Web Application (.NET Core) and click OK.

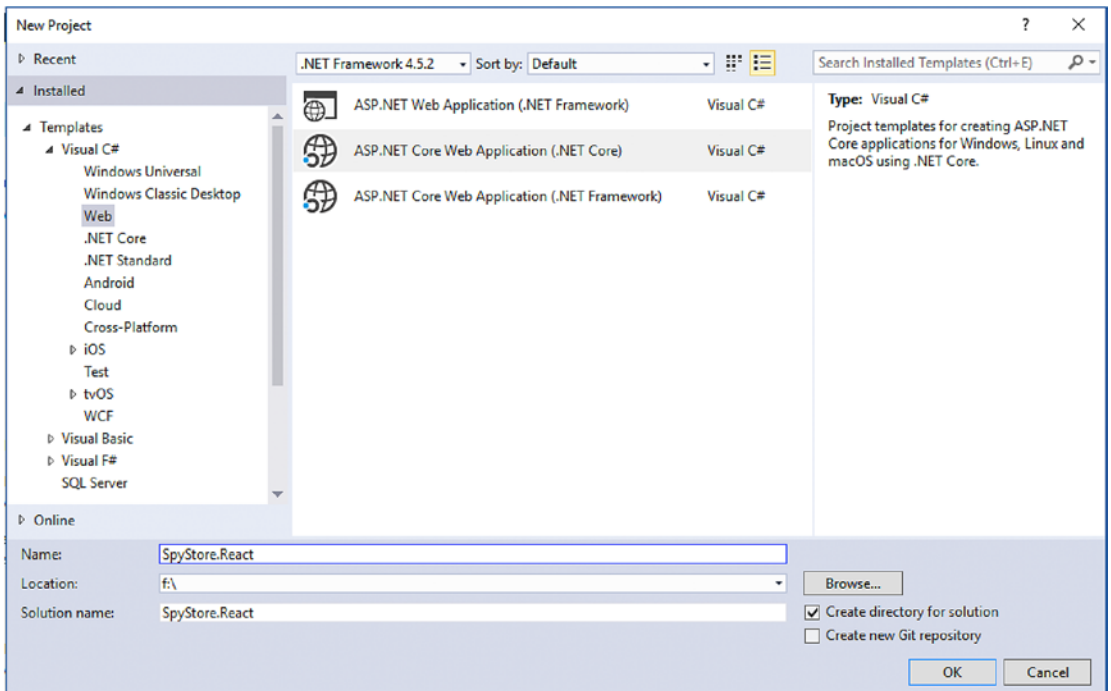


Figure 9-1. Dialog for creating a new project

After confirming your selection, a New ASP.NET Core Web Application dialog will appear, as shown in Figure 9-2. From this dialog, select an Empty project, which should be the default selection, and click OK. This will create a new project and automatically kickoff the package restoration process. At this point we have a new ASP.NET Core project configured with no functionality. We will use this project to host all of our project assets and to set up the React implementation of the SpyStore interface.

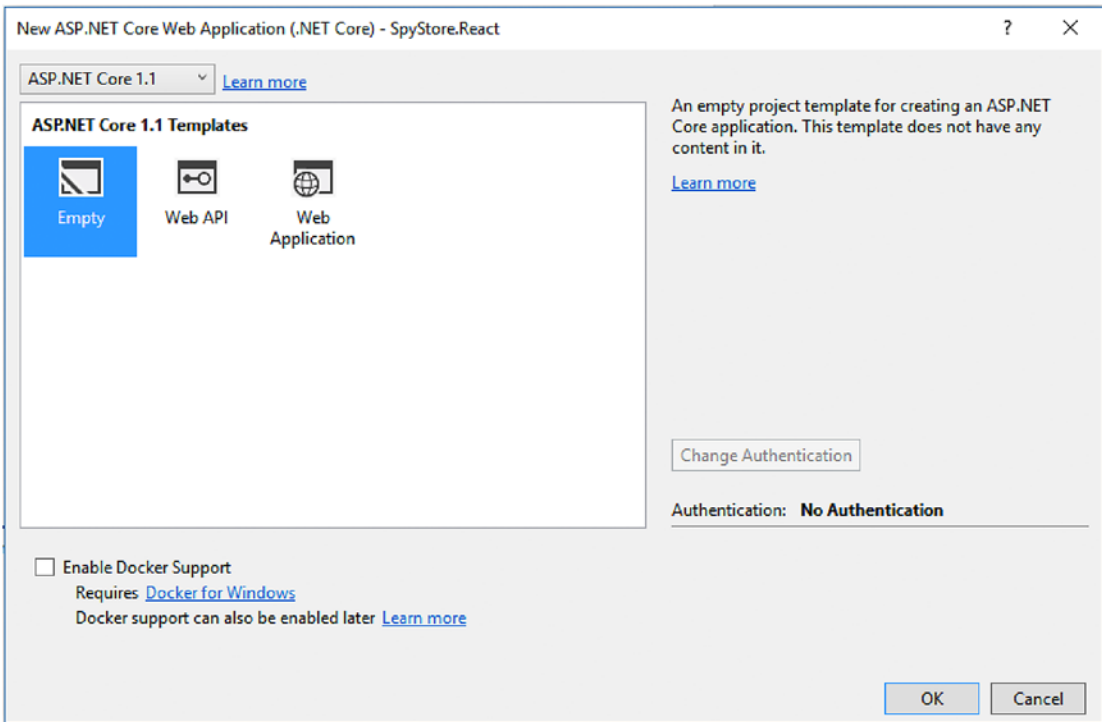


Figure 9-2. Second page of the dialog for creating a new project

Project Files

As you've seen previously in this book, the empty ASP.NET Core project we now have contains very few files. One of the primary files and the primary code thus far is contained in the `startup.cs` file. As with many ASP.NET Core projects, one of the first things we need to do is decide which capabilities our application will need and to configure our `startup.cs` class to support the necessary web request pipeline. For our React application, we will primarily be using ASP.NET Core to serve up static files. To achieve this, we need to add the static files capability via an external NuGet package. As demonstrated in prior solutions, the `Microsoft.AspNetCore.StaticFiles` package will be the one we need. Figure 9-3 shows adding the NuGet package to your project for static files, which allows for serving of any static content from our web server.

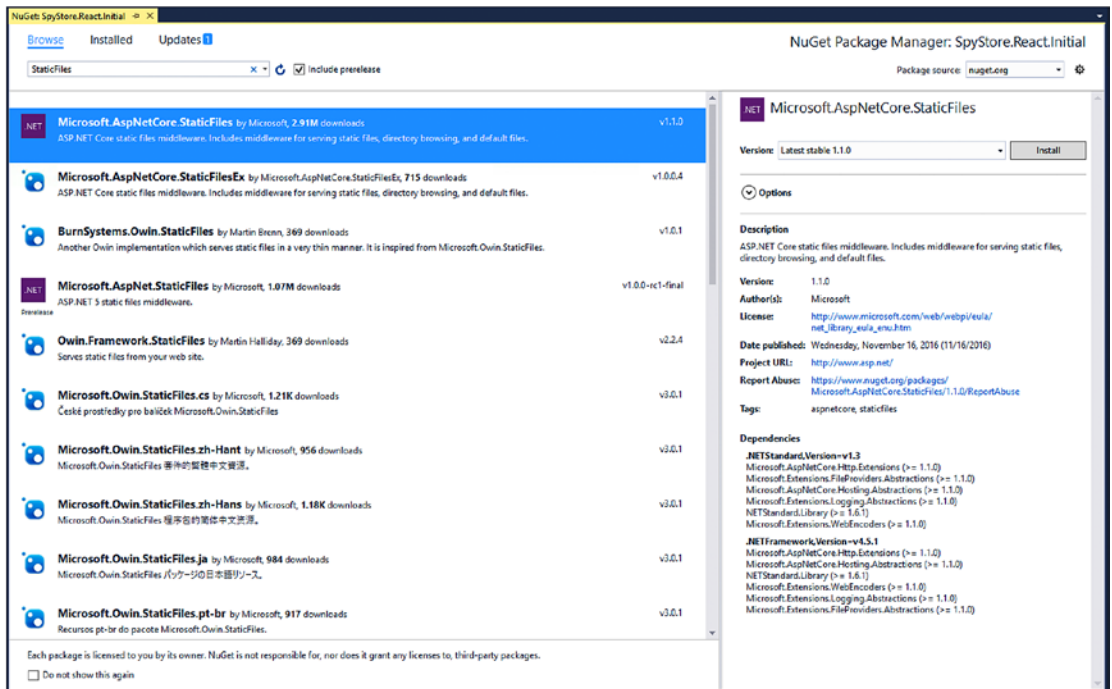


Figure 9-3. Adding the StaticFiles NuGet package

Setting Up the Startup Class

The next step to support the serving of static files within ASP.NET Core is to modify the ASP.NET Core pipeline, which is set up in the Startup.cs file. To this file we will add the same basic code that was utilized in the previous chapter on Angular. Inside the Startup.cs file, locate the configure method. In the configure method, you will find the following code.

```
app.Run(async (context) =>
{
    await context.Response.WriteAsync("Hello World!");
});
```

Replace the entire block of code with the single line of middleware code necessary for serving static files.

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

Also in startup.cs we will need to add some code to indicate we want our server to handle responses for default files. This enables default index.html files to be served when a directory is requested. Add the following line to the Startup class directly below the line indicating to use static files:

```
app.UseDefaultFiles();
```

As you will see later in this chapter, the React framework uses a routing infrastructure that provides a similar set of functionality as the routing infrastructure in Angular. To achieve this, it is also common for React routes to contain URLs that do not map back directly to files on the server. In these instances, React's routing configuration uses the URL to determine which components to render for a view. If a user refreshes his browser, the browser will try to perform an HTTP Get back to the server with the current URL. This will cause us problems because, while the client side React code knows what it should be showing for a route, the server won't be able to find any corresponding content and will return a HTTP 404 "Not Found" error. To get around this issue, we will need to configure the ASP.NET Core middleware to handle things in a way which allows our server to return static content for anything it can find physically on disk and, for anything it doesn't find, return a fixed file. In our case we want this file to be `index.html`.

To achieve this goal, add the following code:

```
using System.IO

...
// Route all unknown requests to app root
app.Use(async (context, next) =>
{
    await next();

    // If there's no available file and the request doesn't contain an
    // extension, then rewrite request to use app root
    if (context.Response.StatusCode == 404 && !Path.HasExtension(context.Request.Path.Value))
    {
        context.Request.Path = "/index.html";
        await next();
    }
});
```

At this point our ASP.NET Core plumbing has been customized to support static files, which is a necessary step for most frameworks, including Angular, React, or Aurelia. It is important to warn you that, although we made these changes to the `startup.cs` file as an initial step in setting up our project to support React, the development workflow we will put in place shortly will not take advantage of the ASP.NET Core runtime until we ultimately deploy our site to a test or production server. For our day-to-day development tasks, we will be taking advantage of WebPack's built-in development web server to serve up and debug our files. This scenario skips our ASP.NET Core server side code (including any customizations to `startup.cs`).

NPM Packages

The next step in our configuration process is to decide which packages we will need to support our React frontend as well as the development workflow we want to use. Previous examples of using NPM have listed individual packages and directed readers to install them via the command line using the `-save` or `-save-dev` flags to tell NPM to add the specified information to the projects `package.json` file. For our solution we are going to add a `package.json` file directly to our project via Visual Studio's Add New Item dialog, as shown in Figure 9-4.

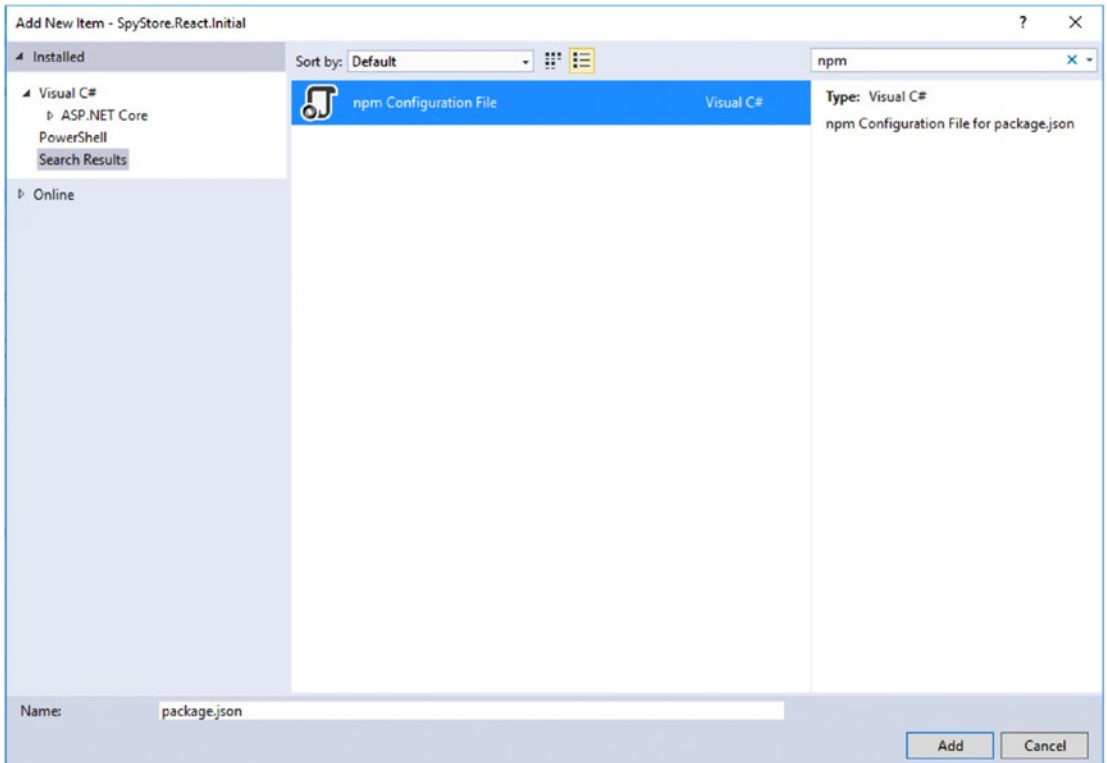


Figure 9-4. Adding an NPM configuration file to our project

Once this file has been added, we will edit it directly in Visual Studio. In many scenarios, editing this file manually to maintain your list of NPM package dependencies is preferable to adding them each from the command line. Visual Studio recognizes files called `package.json` as specifically being NPM configuration files and provides developers with powerful editing tools such as full IntelliSense. This makes working with NPM much easier. The following link provides a great overview of the NPM integration provided in Visual Studio 2017: <https://webtooling.visualstudio.com/package-managers/npm/>.

For our SpyStore React solution we will require several dependencies for libraries such as React itself and for tools to support our development workflow such as WebPack. We will start by adding the following content to the `package.json` file (replacing its default contents):

```
{
  "version": "1.0.0",
  "name": "SpyStore.React",
  "private": true,
  "dependencies": {
    "jquery": "3.1.1",
    "react": "15.4.2",
    "react-dom": "15.4.2",
    "react-router": "3.0.2"
  },
}
```

```

"devDependencies": {
  "@types/jquery": "1.10.31",
  "@types/react": "0.0.0",
  "@types/react-router": "^2.0.41",
  "typescript": "2.1.6",
  "tslint": "4.4.2",
  "webpack": "2.2.1",
  "webpack-dev-middleware": "1.10.0",
  "webpack-dev-server": "2.3.0",
  "webpack-md5-hash": "0.0.5",
  "webpack-merge": "2.6.1",
  "awesome-typescript-loader": "3.0.4-rc.2",
  "url-loader": "0.5.7",
  "ts-helpers": "1.1.2",
  "ts-loader": "2.0.0",
  "ts-node": "2.1.0",
  "tslint-loader": "3.3.0",
  "open-browser-webpack-plugin": "0.0.3",
  "extract-text-webpack-plugin": "1.0.1",
  "file-loader": "0.10.0",
  "html-loader": "0.4.4",
  "html-webpack-plugin": "2.28.0",
  "compression-webpack-plugin": "0.3.2",
  "copy-webpack-plugin": "4.0.1",
  "source-map-loader": "0.1.6"
}
}

```

If you manually enter this JSON into your `package.json` file, you will notice the IntelliSense in Visual Studio 2017 helps validate the names of the packages and will present you with current versions. The versions for each package were current as of this writing and have been used in developing the full SpyStore React solutions. The source code provided with this chapter also contains this file and some may find it easier to copy its contents from there.

In the list of dependencies, there are very few “standard” dependencies for our solution. These are the core React libraries in addition to jQuery (from which we will use the Ajax component to build a reusable wrapper around the SpyStore API). The bulk of the dependencies are ultimately going to be used to automate our development process and they relate to WebPack and its various plugins and loaders. These will be explained in more detail later but it is worth noting their usage now.

Once you have added this content to your `package.json` file, Visual Studio 2017 should automatically begin the process of restoring the packages. Visual Studio will watch for changes to this file and perform an NPM restore in the background so many of the packages may have been loaded as you typed (assuming you didn’t wait until the end to save your progress). If you don’t think packages are restoring, you can right-click on the `package.json` file in your Solution Explorer and select the Restore Packages option.

As described in previous chapters, installing a “package” into your solution via NPM copies all necessary package files and dependencies into the root of your project into a folder called `node_modules`. This folder and its contents are hidden by default in Visual Studio’s Solution Explorer. Visual Studio 2017 provides the ability to see your currently installed NPM packages under the `Dependencies/npm` node within Solution Explorer. An example of this can be seen in Figure 9-5.

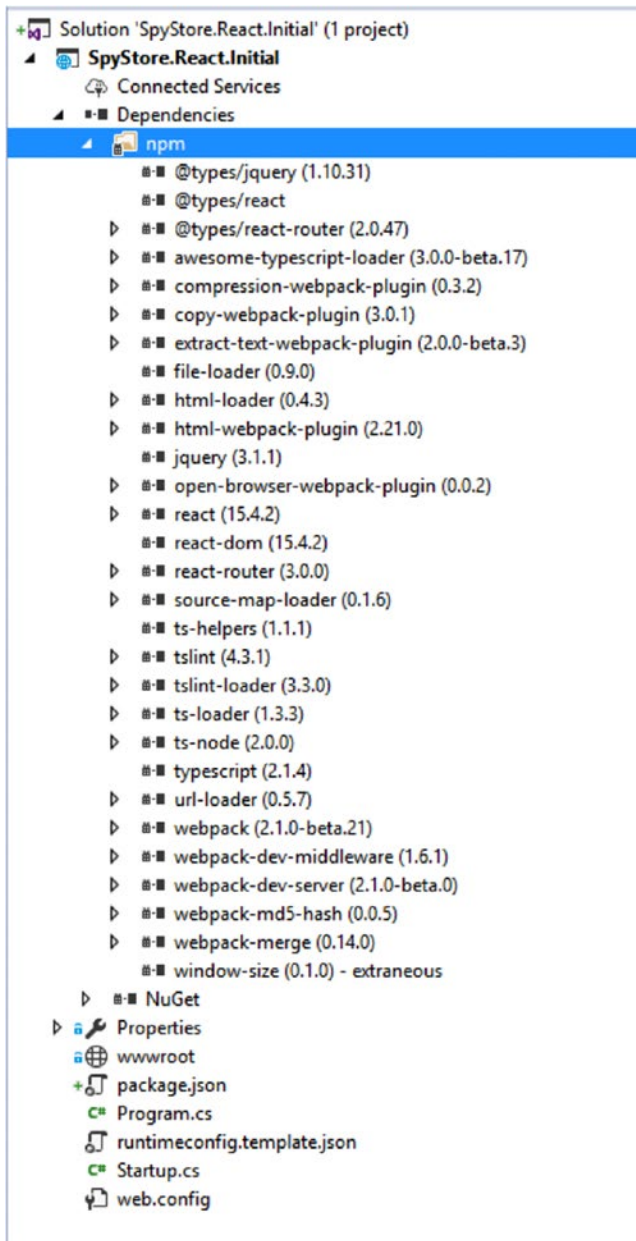


Figure 9-5. Installed NPM packages visible in Visual Studio 2017 Solution Explorer

TypeScript Setup

The next step in configuring our project to support the development process we want is to add a `tsconfig.json` file to the project and use it to specify the options we want the TypeScript compiler to honor as part of our workflow.

You may have noticed in the previous section that our NPM packages file included a package called “TypeScript”. As mentioned earlier, we are not going to be using the TypeScript compiler that ships with Visual Studio and instead we are going to use NPM to download a specific version of TypeScript (2.1.4). We will ultimately use WebPack to use this TypeScript compiler as part of its bundling and development process. This compiler is still the official TypeScript compiler but, by using it in our workflow like this, we have more control over specifically which version is utilized across all development machines and on our build servers.

Even though we use NPM to get the TypeScript compiler, it still depends on a `tsconfig.json` file to read its configuration. We need to add a file with this name to the root of our project and add the following content to it:

```
{
  "compilerOptions": {
    "strictNullChecks": true,
    "noImplicitAny": false,
    "noEmitOnError": true,
    "removeComments": false,
    "sourceMap": true,
    "target": "es5",
    "module": "commonjs",
    "jsx": "react"
  },
  "exclude": [
    "node_modules",
    "wwwroot"
  ],
  "compileOnSave": false
}
```

The settings included here differ pretty significantly from `tsconfig.json` files we used earlier in this book. We have included a new setting called `jsx` and provided it with the setting of `react`. We will discuss JSX files in much detail throughout this chapter but it is important to notice that TypeScript does natively support them.

The `compileOnSave` setting has been set to `false`. This tells Visual Studio not to auto compile our TypeScript files as we save and edit them. Instead we will rely on WebPack to watch for changes and rerun our development workflow and update our browsers.

Initial Project Folder

Before we dive into setting up our development process with WebPack, we will set up our initial project folder structure. The ASP.NET Core project we have created already contains a `wwwroot` folder where our final published files will go. For the React solution we are going to build, we will create a new folder in our project called `src`. This folder serves as the root of all our client side content. Within the `src` folder you will need to create three additional folders: `app`, `images`, and `css`. For purposes of development, all our code (i.e., TypeScript) and other assets will be organized into these three folders. We will configure WebPack to bundle everything we need and move the necessary files for deployment into the `wwwroot` folder, where it is expected in our test and production environments. We will see how this is accomplished in the next section. For now, it is only important to make note of this organization structure and to have the folders created within your project as shown in Figure 9-6.

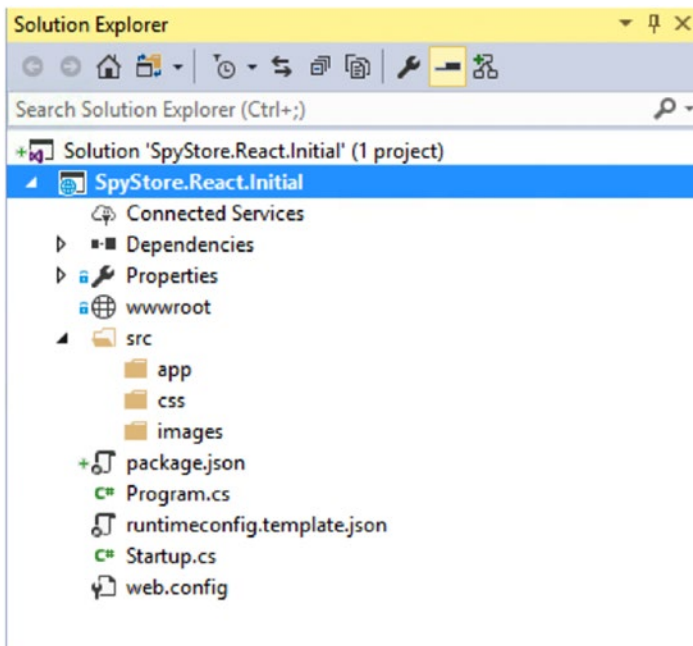


Figure 9-6. Initial folder structure for *SpyStore React* implementation

As we progress, we will add a few additional folders, but the ones shown in Figure 9-6 will serve as the root of most of our content.

Before we move onto setting up our development workflow with WebPack, we will want to add a few initial files to our `src` folder. First, directly within the `src` folder, add a new file called `index.html`. This will be the only static HTML file in our solution and will be the file served to the user who hosts our React content. Modify the contents of the `index.html` file to match the following:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8" />
  <title>SpyStore - React</title>
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
</head>
<body>
  <div id="app"></div>
</body>
</html>
```

This HTML file is about as simple as is possible. We will modify this file slightly later to reference the CSS necessary for our *SpyStore* interface.

Once you have added the `index.html` file, add another file in the `/src/app` folder of this project. This file will be called `index.tsx`. As we will discuss heavily throughout the rest of this chapter, the `*.tsx` extension denotes this file as a TypeScript JSX file. JSX is a special extension used by React to allow developers to easily intermingle HTML directly into their JavaScript. We will explain more about this later

but, for now, use the Add New File dialog in Visual Studio to add a new `index.tsx` file to the `/src/app` folder of the project, as shown in Figure 9-7.

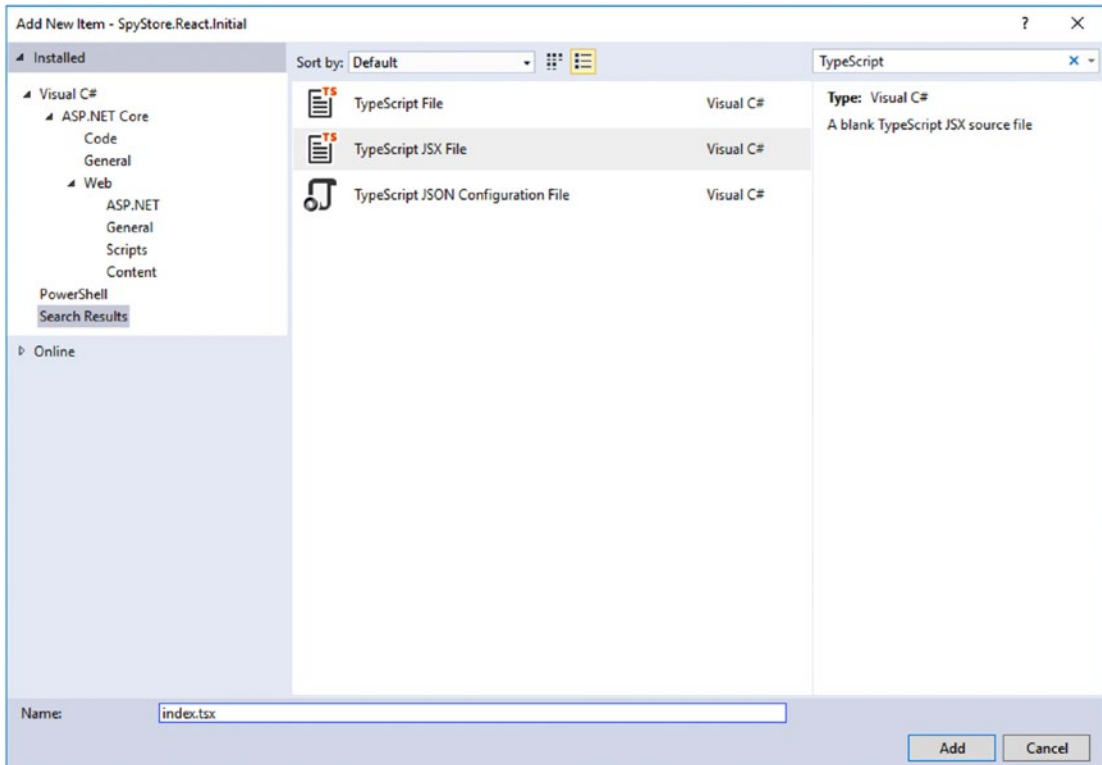


Figure 9-7. Add a new TypeScript JSX file to the project

After the `index.tsx` file has been added to your project, open it and add the following code (which is temporary and will be replaced as we get into the specifics of the React implementation):

```
import * as React from "react";
import * as ReactDOM from "react-dom";

ReactDOM.render((
  <div>Hello World</div>
), document.getElementById('app'));
```

You may notice an unusual syntax in this example. We have wrapped the words "Hello World" directly in a `<div>` tag without quotes and directly within a JavaScript file. This is the correct syntax and something we'll discuss heavily later in this chapter. For now, just go with it.

Once the `index.tsx` file has been added, we will need to gather up the styles and images used in the prior SpyStore implementations so that our React solution can mimic the same look and feel. These assets will need to be copied into our own project structure and the markup for the individual SpyStore components will be developed later as part of our React component tree.

The assets we need can be copied from the `SpyStore.Angular` solution from the previous chapter or are found in the `SpyStore.React.Initial` solution in the source code provided with this book. They include images and fonts in addition to the `SpyStore-bootstrap.css` file, which contains all the necessary CSS styles for our interface.

Once these files are added, your solution structure should look like Figure 9-8.

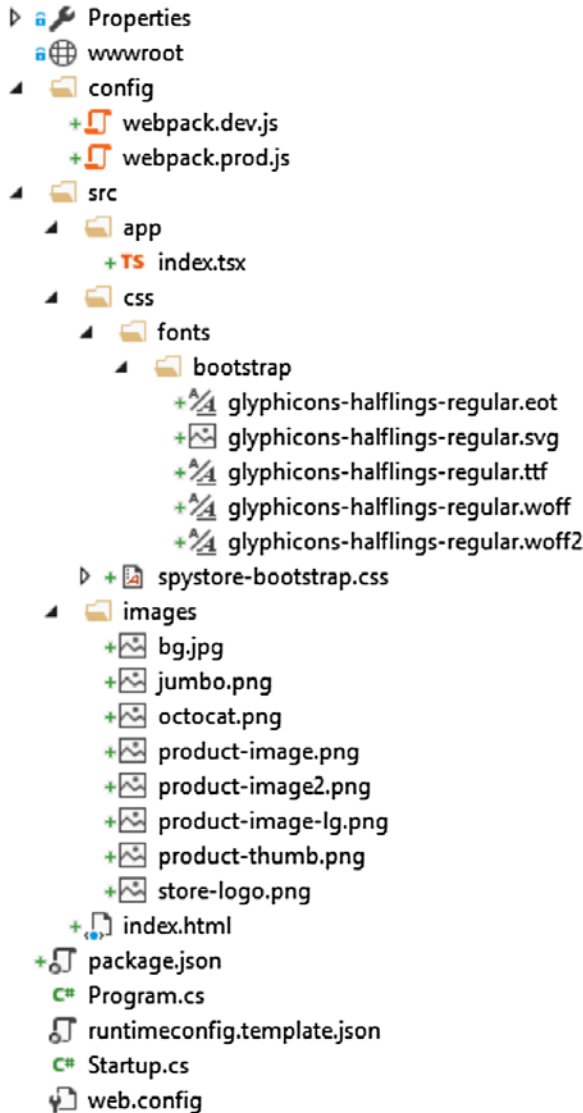


Figure 9-8. Initial project structure containing core UI files

Note in Figure 9-8 a few JavaScript files in a folder called `config`. These files relate to the WebPack configuration we will discuss next. They will be added to your project as part of that discussion.

Webpack

The next step in setting up our project is to go through the process of configuring the WebPack infrastructure to perform the tasks we need for a smooth development and deployment process. The use of WebPack is not specific to projects built on React. WebPack has been adopted heavily on projects using other frameworks such as Angular and Aurelia. Any frontend JavaScript framework that declares dependencies via a module system such as CommonJS (which we will demonstrate and discuss later) can take advantage of WebPack.

In the previous AngularJS chapter, we used a Gulp script to selectively copy assets from various directories into the `wwwroot` folder where the ASP.NET Core infrastructure serves up the files to the client. When developing large JavaScript applications many developers use tools such as Grunt, Gulp, or Bower to automate key tasks during development. These tasks range from simply copying files to minification of JavaScript files, enforcement of standards, compiling (or transpiling) code, or any other low-level task you may need to automate during your JavaScript build or debug cycle. WebPack is another option for performing these tasks and offers many benefits not easily achieved using the other tools.

Before we dive too deep into what WebPack is and how we are going to configure it within our current project, it's important to reiterate again that using WebPack versus one of the other tool or tools mentioned is a matter of choice. I have worked with teams with very efficient build cycles using Grunt or Gulp who have never had a need to consider an alternative like WebPack. I have also seen teams with poorly designed or inefficient processes using WebPack. As with comparing the various web frameworks presented in this book, you should also evaluate the various options for developing an efficient build cycle for your web assets. Toward that end, we will be implementing WebPack in this chapter.

First, a bit about what WebPack is and is not. WebPack is a “module bundler” designed and written specifically for the purposes of efficiently managing JavaScript assets through the development pipeline. Whereas some of the other tools mentioned are more general purpose “task runners” supported by large libraries of plugins, WebPack’s purpose in life is to efficiently scan and bundle JavaScript assets as they move through its series of loaders and plugins. Most of these tools can move files, or minimize JavaScript, or compile SASS files to CSS. WebPack can perform these tasks as well. But where it truly shines is in optimizing your modular JavaScript application into a small set of output bundles that include only the dependencies necessary for execution. This capability is just as valuable on a project using React as it is on a project using Angular or Aurelia.

WebPack is a development tool that is downloaded into your project via NPM. We saw in an earlier section how many of our initial NPM development dependencies were related to WebPack. Once WebPack and any additional loaders or plugins are installed via NPM, you can execute WebPack directly from the command line. You can perform most WebPack tasks from the command line directly but, for most complex uses (including the setup in this chapter), you will want to set up one or more config files for WebPack to utilize. WebPack config files are simply JavaScript files that specify the configuration of your WebPack pipeline. For our project, we are going to create a new folder in the root of the project called `config`. Within this `config` folder, we will add two new JavaScript files: `webpack.dev.js` and `webpack.prod.js`. This can be seen in the screenshot in Figure 9-9.

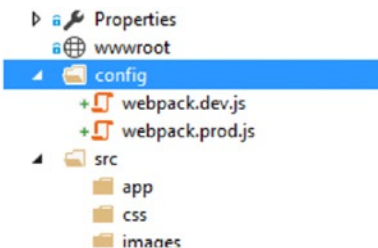


Figure 9-9. WebPack configuration files added to our project

Note that the names of these files are arbitrary, as we will be passing the specific filenames into the WebPack command line tool. Your project may have the need for more than two distinct configurations and it is also possible (and easy) to have a third file with any configuration that is common between a prod and dev setup. For purposes of this chapter, we will not demonstrate how to share configuration and will just copy any common settings between prod and dev.

The first thing to know about a WebPack configuration file is a valid JavaScript file that exports its configuration via a JavaScript Node module written in JavaScript. This differs dramatically from how a few of the other tools do it but, once you get used to it, you will find it very straightforward and most of your WebPack configuration settings will likely just be copies of very similar files moved from project to project. It is also important to note that every time you run the WebPack command line and specify a specific config file, the code in that config file is executed. This means you can use other utility libraries or log information to the console as needed. This is a powerful capability we will utilize in our own config files.

Next, let's look at the beginning of a very basic WebPack config file. This code will use the path package included with Node (which we are using via NPM) to help us cleanly build up relative paths to the various key folders of our project. This is done to simplify our remaining code and make it easy to see in the console that our locations are correct. The section of our WebPack configs related to building up our relative paths contain the following:

```
const path = require('path');
const webpack = require('webpack');

var paths = {
  root: path.resolve(__dirname, ".."),
  src: path.resolve(__dirname, "..", "src"),
  css: path.resolve(__dirname, "..", "src", "css"),
  images: path.resolve(__dirname, "..", "src", "images"),
  app: path.resolve(__dirname, "..", "src", "app"),
  output: path.resolve(__dirname, "..", "wwwroot"),
  scripts: path.resolve(__dirname, "..", "wwwroot", "scripts")
};
```

Building up a small utility object containing some path properties will simplify things later. Note that all paths generated in this code are relative to the location of the WebPack config file. It is for this reason we are using `path.resolve` to step back up a directory (via `..`).

Now that we have the means to easily find the path locations to the content in our project, we will begin adding the code to configure WebPack itself. There are many great references on the web and in books to provide the full details of how WebPack functions. For purposes of this chapter, we will provide two pre-written configuration files and briefly discuss the various sections and their purpose. They can be used “as is” but I encourage all readers wanting to do more with WebPack to do the research and add more functionality to the basic build process provided. There is a large WebPack community out there and dozens of great plugins and tools, which could greatly benefit your project.

Let's start by looking at the content of our “production” WebPack config file. When we say it is “production” we mean its job is to run all the necessary build steps and generate the necessary output files and structure in the `wwwroot` folder of our ASP.NET Core project. These steps would be executed prior to a deployment to a remote environment or as part of a continuous integration build. The full code for our production WebPack config file follows. This code can be found in the `SpyStore.React.Initial` project under the `config` folder. Its name is `webpack.prod.js`.

```
const path = require('path');
const CopyWebpackPlugin = require('copy-webpack-plugin');
const HtmlWebpackPlugin = require('html-webpack-plugin');
const webpack = require('webpack');
```

```

var paths = {
  root: path.resolve(__dirname, ".."),
  src: path.resolve(__dirname, "..", "src"),
  css: path.resolve(__dirname, "..", "src", "css"),
  images: path.resolve(__dirname, "..", "src", "images"),
  app: path.resolve(__dirname, "..", "src", "app"),
  output: path.resolve(__dirname, "..", "wwwroot"),
  scripts: path.resolve(__dirname, "..", "wwwroot", "scripts")
};

console.log("Src: " + paths.src);
console.log("Out: " + paths.output);

module.exports = {
  entry: paths.app + "/index.tsx",
  output: {
    filename: "bundle.js",
    path: paths.scripts
  },
  resolve: {
    extensions: [".ts", ".tsx", ".js"]
  },

  module: {
    loaders: [
      // All files with a '.ts' or '.tsx' extension will be handled by
      'awesome-typescript-loader'.
      { test: /\.tsx?$/, loader: "awesome-typescript-loader" }
    ]
  },
  plugins: [
    new CopyWebpackPlugin([{ from: paths.css, to: path.resolve(paths.output, 'css') }]),
    new CopyWebpackPlugin([{ from: paths.images, to: path.resolve(paths.output, 'images') }]),
    new HtmlWebpackPlugin({
      template: path.resolve(paths.src, 'index.html'),
      output: path.resolve(paths.output, 'index.html')
    }),
    new webpack.NoErrorsPlugin(),
    new webpack.optimize.UglifyJsPlugin(),
  ]
};

```

Let's discuss some key aspects of this file. First, the constants declared at the top of the file use the `require` function to resolve contents of additional modules. When we say `require('webpack')`, the resolution of the 'webpack' module is smart enough to look inside the `node_modules` folder and find the right resource. The modules that we require at the top of our file are either included with Node (i.e., `path`) or are modules that we previously included in our NPM package configuration and should already have been downloaded into our `node_modules` folder. We simply get access to the ones we need in the WebPack configuration by "importing" them into a variable we can access.

The next block of code is responsible for generating our paths into a nice structure called `paths`. This will be utilized throughout the rest of the configuration. Note that we output two of the primary paths to the console via calls to `console.log`. When executing this WebPack script, this will allow you to visually confirm things are going where they should be.

The next major (and most confusing) section is the WebPack configuration itself. In the WebPack configuration you will see several key sections where we will specify settings, paths, or specific items called *plugins* or *loaders* (these terms are key in the WebPack world).

First, WebPack needs to know where your application “starts”. It calls this the “entry point”. There may be one file that kicks things off or there may be many individual files. WebPack can support most configurations you can imagine. In our `SpyStore` implementation, the `/src/app/index.tsx` file we created earlier will be our single “entry point”. All other screens and functionality we will require to run can be figured out by looking at that file and figuring out what dependencies it has and then finding the dependencies of those files and so on. This is the most key function of WebPack and one we will talk more about and demonstrate shortly.

The next important piece of configuration information required by WebPack is where you want the “outputs” it generates written. It takes your inputs (i.e., `index.tsx` for us) and walks all their dependencies, runs everything through a defined pipeline to perform tasks, and writes the resulting “bundles” to a location or locations we specify. In the production configuration, we specify the folder `/scripts/bundle.js` as the place we want the results written. This is done with the following:

```
output: {
  filename: "bundle.js",
  path: paths.scripts
},
```

Note that the output will ultimately be in our `wwwroot/scripts/bundle.js` folder.

Next, we need to specify what *loaders* we want to process files as they are found by WebPack. Loaders are core piece of the WebPack ecosystem and there are many available to perform various tasks. When using WebPack loaders, we must first tell WebPack which file types we expect to be processed by loaders at all. This is done in the following code:

```
resolve: {
  extensions: [".ts", ".tsx", ".js"]
},
```

The `resolve` section simply provides a list of file extensions WebPack will use to determine if a file should be processed by any loaders. As you will see, individual loaders can specify which files they specifically need to process. The `resolve` section should include extensions for all file types for which you have specified at least one loader. With the following code, we specify only a single loader:

```
module: {
  loaders: [
    // All files with a '.ts' or '.tsx' extension will be handled by
    // 'awesome-typescript-loader'.
    { test: /\.tsx?$/, loader: "awesome-typescript-loader" }
  ]
},
```

When adding our loader to the collection, we specify a “test” of the RegEx that runs against every resolved file type to see if this loader should handle the file. The single loader in our production config file tests for file types of `ts` or `tsx`. This specifies that our loader should process any TypeScript files it resolves. It will not be used for any other file type. The next parameter we specify for our loader is its name. The loader we are using is called `awesome-typescript-loader` and the name directly relates to one of the previously loaded NPM modules. These loader names will be found and resolved via the `node_modules` folder.

So what does `awesome-typescript-loader` do? It uses the TypeScript compiler to efficiently compile all our TypeScript files (including those with the `TSX` extension we’ll discuss later). As part of the WebPack loading process, the contents of a TypeScript file are passed into this loader and compiled (or transpiled) JavaScript is returned out the other end. More information on this plugin can be found here: <https://github.com/s-panferov/awesome-typescript-loader>.

The final section of our current production WebPack configuration is the `plugins` section. It is here we specify any WebPack plugins that we want to run after the loaders have processed all of their files. Our plugin section contains the following:

```
plugins: [
  new CopyWebpackPlugin([{ from: paths.css, to: path.resolve(paths.output, 'css') }]),
  new CopyWebpackPlugin([{ from: paths.images, to: path.resolve(paths.output, 'images') }]),
  new HtmlWebpackPlugin({
    template: path.resolve(paths.src, 'index.html'),
    output: path.resolve(paths.output, 'index.html')
  }),
  new webpack.optimize.UglifyJsPlugin(),
]
```

Here we instantiate and configure several predefined plugins for various tasks. The first two plugins are of type `CopyWebpackPlugin` (which we resolved near the top of our config file). This plugin is commonly used to copy files. We use it here to move all content from our `/src/css` folder to the `/wwwroot/css` folder. We do the same to move images from `/src/images` to `/wwwroot/images`.

The next plugin we specify is called `HtmlWebpackPlugin`. This plugin is very cool but a bit confusing upon first look. This plugin takes the path to our `index.html` file (currently sitting empty in the `/src` folder). It specifies an output location of `/wwwroot/index.html`. One might initially think the job of this plugin is to copy that file (and it does copy it). What makes this plugin unique is that it not only copies the `index.html` to the output folder but it also injects HTML script tags for all WebPack outputs at the bottom of the `<body>` tag. Using the configuration settings we specified, the following tag is injected into `index.html` prior to saving it to its output directory:

```
<script type="text/javascript" src="/scripts/bundle.js"></script>
```

If we had specified multiple output bundles, multiple script tags would be put into our `index.html` file. This step is extremely handy, as we no longer need to manually keep track of all the referenced bundle files directly in our own root HTML file. In such a simple example like this, it may be hard to see the benefit of this small step, but in larger, more complex projects, it’s a definite advantage.

The final plugin we specify is `UglifyJsPlugin`. This plugin is packaged with WebPack directly. This plugin performs the final step of removing comments and minifying all JavaScript. This is a great step in a production release config file but it is likely not a plugin you would use in a WebPack script used during development (as it would complicate debugging).

There are many other useful WebPack plugins and loaders we are intentionally not using in this book but which you may well want to take advantage of. You can use WebPack plugins and loaders to do things such as:

- Compile SASS files down to CSS and bundle the CSS into a single file
- Run Linters such as TSLint or JSLint to help enforce our coding standards
- GZip compress CSS and JavaScript files for faster downloads
- Remove arbitrary calls to functions such as `log` from all the production JavaScript

These are just some examples of the power of the available plugins within the WebPack ecosystem. As we move from project to project and find handier WebPack utilities, we evolve our common set of configuration settings and improve our development process.

Now that we have briefly discussed the various options specified in the `config/webpack.prod.js` file let's see it in action. First, make sure the file exists in your project and the contents match what have previously specified. If you do not want to go through the headache of so much typing, feel free to grab the contents of this file from the code provided with this chapter. The next step is to open a command line console to the root of your project. We will use command line consoles frequently in this chapter so it may be handy to keep one around. Once you have navigated to the root of your project, execute the following command:

```
webpack --config config\webpack.prod.js --progress --profile
```

This line of code executes WebPack from the command line and specifies our configuration file. The final two parameters simply tell the tool to be a bit more verbose in what it writes out to the console and to provide some timings on various steps. These aren't required but are somewhat interesting if you are new the tool.

If any errors occur during the execution of the WebPack config file, you will need to troubleshoot your config file accordingly. Errors will be output to the console in bright red so they stand out.

If you were lucky enough to get everything running fine on the first try (i.e., no errors) then you should be able to navigate to the `wwwroot` folder of your ASP.NET Core project and see content that looks something like Figure 9-10.

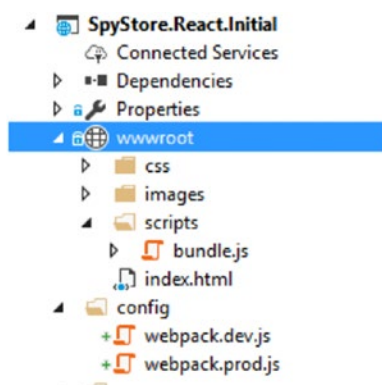


Figure 9-10. WebPack generated content in the `wwwroot` folder

If you open the `index.html` file you will see the original contents from the `src` folder version with the addition of the injected `<script>` tag to reference the `/scripts/bundle.js` file. If you open the `bundle.js` file you will see a single and long line of JavaScript. This single file contains our code (currently only what we put in `src/app/index.tsx`) and all dependencies. Since the few lines of code we included thus far included references to React components (i.e., `react` and `react-dom`), those libraries were parsed (from `node_modules`) and included in the output bundle as well. On my system, the `bundle.js` file is 214KB in size and includes everything necessary for my app to run (including React). If I add another TypeScript file to my application and reference it from the current entry point (`index.tsx`) and rerun the WebPack command line, the new file will automatically be found and included.

In the current configuration file, we generate a single output file (`bundle.js`) that will include all our own code plus all the referenced code from third-party libraries. This may look unusual to some and this setup isn't a good fit for all applications. We will use it now for simplicity but, in more robust applications, it's often preferable to have a small handful of output bundles. We frequently bundle our own code into one output file and third-party libraries like React or Angular into their own bundles. As these libraries change less frequently than our own code it allows them to remain cached in client-browsers even when changes to our own code occurs.

Now that we have completed a production WebPack config file, we are going to copy the same configuration to our development config file and modify it for a slightly different process. The development config file we created as `/config/webpack.dev.js`. For now, copy the contents of the `/config/webpack.prod.js` file into the `/config/webpack.dev.js` file.

Next, we will need to add another plugin to the top of the development config file. You can do so by adding this line:

```
const OpenBrowserPlugin = require('open-browser-webpack-plugin');
```

This is another plugin we have already downloaded via NPM. It will be used later to open a web browser.

Next, we will add some new content to our config specific to using it as part of our development process. As mentioned, we will need to remove the line indicating `UglifyJsPlugin`. We will then need to add two new WebPack plugins to the stack, as indicated in the bold lines:

```
plugins: [
  new CopyWebpackPlugin([{ from: paths.css, to: path.resolve(paths.output, 'css') }]),
  new CopyWebpackPlugin([{ from: paths.images, to: path.resolve(paths.output, 'images') }]),
  new HtmlWebpackPlugin({
    template: path.resolve(paths.src, 'index.html'),
    output: path.resolve(paths.output, 'index.html')
  }),
  new webpack.NoEmitOnErrorsPlugin(),
  new OpenBrowserPlugin({ url: 'http://localhost:3001/' })
],
```

Next, we need to add a few new sections to the root of our module called `devServer`. These sections should be put at the same level as `entry`, `output`, or `plugins` (just be sure not to nest these sections in another). The content for these sections follows:

```
devtool: "source-map",

devServer: {
  host: '0.0.0.0',
  port: 3001,
  historyApiFallback: true,
  stats: 'minimal'
}
```

The key purpose of the changes to the development config as compared to the production config is that we want to take advantage of WebPack's pipeline while developing. To make this easier, we will be utilizing another Node module we have already loaded called `webpack-dev-server`. Some of the configuration options we just added to the development configuration were specifically for this environment. To see this process in action, go to the command line in the root of your project and execute the following command:

```
webpack-dev-server --config config/webpack.dev.js --progress --profile --watch --content-base src/
```

If all goes as planned, you should see the console output of WebPack bundling your assets and executing your plugins (just as you saw with the production configuration). At the end of the process, the `webpack-dev-server` should open a browser and you should see a single page with the words "Hello World". There are a lot of moving parts here so, if you don't see these results or get errors, some troubleshooting may be required.

This command tells WebPack to perform the tasks specified in the config file (`webpack.dev.js`) and, due to the recent addition of the `devServer` config setting to our configuration file, it will launch a Node web server in-memory and host all resulting assets in memory at the URL `localhost:3001` (as we specified in the configuration). The final two parameters we specified in the command line (`--watch --content-base src/`) tell the tool to remain open and watch all the files in our project under the `/src` folder. If it detects any change to any of those files, it will re-execute the entire WebPack development process and refresh the browser. This means we can edit our TypeScript code and all other assets found under our `/src` folder and very quickly see all changes reflected in our browser. We can even take full advantage of the full TypeScript debugging capabilities of modern browsers like Chrome. If the WebPack process fails for any reason (such as errors in our TypeScript code), we will see the errors output in our console (in red) and the current browser will not refresh.

Here are a few important things to note about writing and debugging our code using the WebPack development web server.

- First, no files are written to disk during this process. The entire WebPack process occurs in memory and the files are hosted in memory and attached to the Node web server to be served as static files. This means you will not be able to physically find any of the files on disk. If you use the WebPack command line to execute the `/config/webpack.prod.js` configuration file you will see the physical files generated in the `/wwwroot` folder. The `/config/webpack.dev.js` file is configured to be generated in memory by the development server.
- Second, when using browsers such as Chrome to debug your TypeScript code in the browser (as discussed in Chapter 7), you will find all bundled TypeScript under a `webpack node` under the Chrome Sources tab, as shown in Figure 9-11.

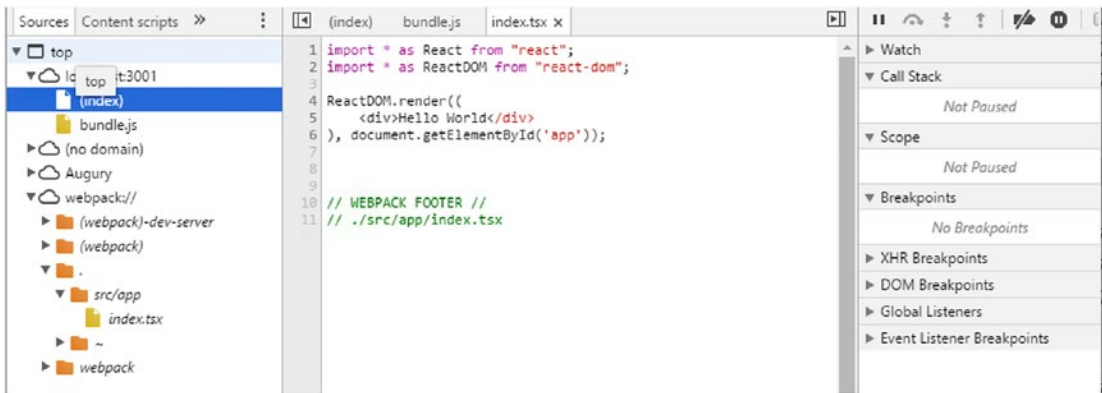


Figure 9-11. Debugging TypeScript in Chrome with WebPack

The settings in our development configuration tell WebPack not to minify any output and to include the TypeScript and corresponding JavaScript map files. Expanding the `webpack://` node in Chrome and navigating to the `./src/app` folder allow us to see our single TypeScript file. From here we can set breakpoints and debug accordingly. This all works seamlessly with the WebPack build process and editing any file under the `/src` folder in Visual Studio 2017 (or any other editor or IDE) will cause the process to be run again and will reattach all assets to your browser. The only requirement is that you keep your command line open and have executed the `webpack-dev-server` command line.

When dealing with complex command line statements necessary for a smooth development process, it is often preferable to simplify things as much as possible. For our projects we take advantage of the scripts section of an NPM packages .json file. We add a section such as the following to our package .json file:

```
"scripts": {
  "dev": "webpack-dev-server --config config/webpack.dev.js --progress --profile --watch
  --content-base src/",
  "pack:dev": "webpack --config config/webpack.dev.js --progress --profile",
  "pack:release": "webpack --config config/webpack.prod.js --progress --profile",
  "publish": "npm prune && npm install && npm run pack:release",
  "start": "npm run dev"
},
```

Once this section has been added, we can execute the scripts via the following commands executed from the root of our project:

```
npm start          // Execute the webpack-dev-server process and open a browser so we can develop
npm run publish    // Execute the webpack.prod.js configuration file and generate all content
                  into the /wwwroot folder
```

We are very consistent about the scripts section we include in our packages .json folder so that our developers know what to expect from project to project. Our automated deployment process executes the `npm run publish` step to generate all assets into the `/wwwroot` folder prior to using the ASP.NET Core infrastructure to package.

If you find yourself frequently needing other commands, you can add whatever you like to the `package.json` file. As a command line tool NPM has several parameters. The one we are taking advantage of is `run`. Executing `npm run XXX` will look in the `package.json`'s `script` section for a corresponding `XXX` command. This will be executed accordingly. Thus, we need to specify `npm run publish` to execute that specific command (`npm publish` is a different command related to publishing packages into the NPM registry). The `start` command is a unique case with NPM and executing the `npm start` command can be done without using the `run` option. Thus we configure our `start` option in the `package.json` file to perform an `npm run dev` command. We can also just type `npm run dev` from the command line to see the same result.

At this point you should have two WebPack config files prepared: one to facilitate a real-time build/debug process while you edit code and another to package all assets and bundles and physically copy them to your `wwwroot` folder. You should be able to execute the first configuration file from the command line at the root of your project via the `npm start` command. Once your browser opens and you see the words "Hello World," you can verify this entire process by opening the `/src/app/index.tsx` file and modifying the words "Hello World." Once you save this change you should see the console refresh with a new WebPack compilation process and the browser automatically refresh to display the newest content.

At this point we have spent a significant amount of time setting up our environment for all necessary packages, an initial folder structure and some simple content (`index.html` and `index.tsx`), and then a set of WebPack configuration files to support developing and deploying our site. Just this infrastructure on its own is a very powerful development process. It is also important to note that, while we've included some React packages in preparation for the remainder of this chapter, the WebPack and development setup we just performed can be used just as easily with Angular or Aurelia (or any other frontend framework for that matter).

Introduction to React

Now that we have a robust development cycle set up, we can move on to learning the core concepts of React and implementing the SpyStore interface with React.

First, it is important to note that React is very focused on being a framework for developing user interfaces. As such, in many ways, React contains much less than Angular in terms of core services. React does not contain any native means of dependency injection, no concept of modules, no core services for such things as modules. React primarily focuses on building robust user interfaces by composing a series of "components." It also supports a powerful routing infrastructure. Within the scope of these two areas React features overlap with Angular features. Beyond component-based interfaces and routing, developers using React need to look to external libraries to replace some of the functionality that Angular developers may get from core Angular libraries. This need is neither good nor bad and is instead a design decision made by the React team at Facebook. They focused on a robust library to do one thing very well and made the conscious decision not to include any more than they needed to accomplish these goals. On the other hand, the Angular team at Google made the decision to include a more opinionated framework and set of services to address a broader range of architectural hurdles and to encourage (and, in some ways, enforce) a more defined pattern for implementation.

Components

One of the core concepts in React is the concept of a "component." Components in React overlay nicely with the component infrastructure specified in ES6. The concept of a component in React is analogous to that of a component in Angular or other web frameworks.

A component is generally an HTML view or part of an HTML view. It could be an entire screen or something as simple as a row in a table. A React application of any size is really a series of components containing other components. The "root" level components would be considered your "pages" and a top-level

component may be built from varying smaller components. A well designed React system efficiently makes use of components to maximize reuse and isolate functionality specific to an area of the screen into a smaller component. Things like headers, navigation menus, and footers are traditionally separated into their own components.

To build a component in React is as simple as creating a class that inherits from (or “extends” in TypeScript) the base `React.Component` class. Since we’ve included `@types/react` in our list of included NPM modules, most modern TypeScript development environments will give us full IntelliSense for the native React classes. The following example demonstrates a very simple React component called `App` (presumably in an `app.tsx` file).

```
import * as React from "react";

export class App extends React.Component<any,any> {

    render() {
        return (
            <div>Hello World</div>
        )
    }
}
```

Note that the `App` component inherits from `React.Component`, which will give us access to some nice utility methods and properties. It includes a single method called `render()`. The `render()` method accepts no parameters and currently has no specified return type (which means any is inferred).

The `render` method currently has only a single line of code, which is used to return some markup. As we mentioned earlier, the markup in this example is directly placed right inside the JavaScript function. We are not returning a literal string such as “`<div>Hello World</div>`” but instead are embedding that markup directly inside the JavaScript. React achieves its purpose by dictating that components have a `render` method that returns the elements that a component wants to render. React uses its own “in-memory” representation of HTML elements using what is called a “virtual DOM”. The React library contains many utility classes and functions for building up its own representation of a DOM hierarchy. The `render` method could be rewritten to manually create its single element like this:

```
render() {
    return React.createElement('div', null, `Hello World`);
}
```

This code uses a common utility function of the React library (`createElement`) to instantiate a new instance of a React virtual element. The React rendering engine knows to render this element as a `<div>` tag containing the string `'Hello World'`. This is not very complex but, as you can imagine, building complex HTML user interfaces by nesting many calls to `createElement` would be extremely time consuming, error prone, and inefficient. For this reason, the React developers came up with a new file type called JSX. JSX files are simply regular JavaScript which are ran through a pre-processor that converts any native tags into the representative calls to `React.createElement`. It’s a bit more complex than that, but not much. The TSX extension we have used briefly in our development environment setup is simply a TypeScript variation of the JSX. This indicates that the same preprocessing of tags will occur and that TypeScript compiler should be used to generate the resulting file into JavaScript.

Using the previous code examples, with a TSX file containing a render method of a React component such as this:

```
render() {
  return (
    <div>Hello World</div>
  )
}
```

Once the TSX compiler completes its transformation step, the resulting TypeScript would look like this:

```
render() {
  return React.createElement('div', null, `Hello World`);
}
```

It is easy to see the direct correlation on this small example, but imagine a somewhat complex UI such as we will be developing as part of the SpyStore implementation. Representing our markup directly in our component's TypeScript is much easier to manage and maintain than a complex set of calls to various React methods for creating JavaScript element classes.

The JSX syntax (which we will refer to this as even though we are using the TSX extension due to our use of TypeScript) is very powerful but has some quirks that you must be aware of.

First, every render method of a React component must return markup that contains a single root component. You can achieve this by wrapping any complex content in a `<div>` or a `` tag.

Next, the element names in JSX are case sensitive. The JSX compiler assuming all lowercase elements it sees are intended to be HTML elements and it wraps them verbatim in `React.createElement` calls accordingly. This works great for standard HTML markup tags such as `<div>`, ``, and others. If the JSX processor encounters a tag with a capitalized name, it assumes it to be a variable or a child component.

Let's look at the following simple React component called `Greeting`:

```
import * as React from "react";

export class Greeting extends React.Component<any,any> {

  render() {
    return (
      <div>Hello { this.props.Name}!</div>
    )
  }
}
```

This component is much like our previous simple example but does have some different syntax in the JSX markup. We will discuss this momentarily. Now let's consider another React component (again) called `App`:

```
import * as React from "react";
import { Greeting } from "../Greeting";

export class App extends React.Component<any,any> {

  render() {
    return (
```

```

        <Greeting name="Kevin"></Greeting>
    )
}
}

```

In the `App` component, we see that we included an `import` statement that references the `Greeting` component. Then, in the `render` method of the `App` component we reference the `Greeting` component by including an element tag with the name `Greeting` (capitalization being important). This powerful concept allows a React component to utilize other components simply by importing them and including them as tags. Notice that, not only does `App` use the `Greeting` element in its markup, it also sets a property on that element called `name`. The `Greeting` component can access this property through its `props` object (exposed via the base class `React.Component`). Your one way of doing this in the `Greeting` component's `render`:

```

render() {
  return (
    <div>Hello { this.props.Name}</div>
  )
}

```

In the `render` method, we use the curly bracket syntax to inject a dynamic value into our markup tree. In this case, the output will be "Hello Kevin". Passing properties from one React component to its child components is a powerful concept we will discuss in more detail shortly.

The curly bracket syntax used in React is, in some ways, like that used in the Angular template engine (while in other ways slightly different). Using this syntax any JavaScript expression can be evaluated as part of JSX markup. We will call out many examples of how this syntax is used as we start to put the React implementation of `SpyStore` together.

Another key point worth making about the JSX syntax in React is that it is not 100% HTML. Since the design of React is to use JSX to intermingle your markup directly into your JavaScript, certain concessions had to be made for keywords that existed in both worlds. One big example is adding CSS class names to markup. In raw HTML you could have an element such as `<div class="header">`. Attempting to add the same markup to a JSX file would cause issues because the JavaScript language has a keyword called `class` that means something different. Thus, within your JSX, you need to use the attribute `className` instead of `class`. When the JSX file is parsed into JavaScript and then, ultimately, rendered to raw HTML elements as the virtual and physical DOM elements are kept in sync, it knows to replace `className` attributes with `class` attributes to be compatible with current browsers. There are many DOM elements that cannot be represented in markup within TSX/JSX files with the same name as they would be with straight HTML (due to naming conflicts with the JavaScript language). A full list of these can be found at this link: <https://facebook.github.io/react/docs/dom-elements.html>.

If an error is found in your JSX markup (or any of your valid TypeScript for that matter) the WebPack build process will fail and the details about what is wrong will be output in red in your console window.

The very first step in "bootstrapping" your React application is to include a line of code to render your first React component directly into the DOM. This is done via a call to `ReactDOM.render`. This function takes two arguments: the React component you want to render and the DOM element you want it to replace. We achieved this early with the following code:

```

import * as React from "react";
import * as ReactDOM from "react-dom";

ReactDOM.render((
  <div>Hello World</div>
), document.getElementById('app'));

```


In the call to `ReactDOM.render` we specified the React component to render directly as JSX markup. The `document.getElementById` looked in the `index.html` file (the only HTML file in our app and the root of our UI) and found the `<div id="app"/>` element to use as the root of the React project. As simple as that we could render our JSX markup into the actual DOM.

For those of you who are familiar with Angular 2 (or have worked through the examples in the previous chapter) you may already notice a large difference in how Angular manages UI markup and how React manages this markup. In Angular, any advanced markup is usually kept separate from the Angular component and the markup and component class (defined by a `@component` decorator recognized by Angular) are “bound” together. This separation of UI and the code is common in most MVVM or MVC frameworks and many developers prefer it. In React, the design decision was made to allow developers to tightly mix their code and markup for speed and to eliminate the need for many of the other elements Angular requires supporting its separation (such as directives). React is much lighter and faster in many scenarios because it implements less overhead and just focuses on a blazing fast UI rendering engine. We haven’t talked about how React efficiently updates the physical DOM with only the changes it detects in the virtual DOM managed by the React components. This is one of React’s strong points and something we’ll point out as the SpyStore UI comes together.

There is much more to React and its component structure and we will demonstrate many of these concepts as we build the SpyStore UI. For now, the overview of the basics of components and the JSX syntax should suffice.

Application Organization

Before we move into learning more about React through implementing the SpyStore UI with React, let’s discuss some early decisions on how we will be organizing our React code. Earlier in this chapter we decided that all our custom code and other UI assets would be organized under a `/src` folder off the root of our project. Within that `/src` folder we created three sub-folders: `/app`, `/css`, and `/images`. All our custom React code for the SpyStore application will be placed in the `/app` folder.

Within the `/src/app` folder of our project we will be organizing our React code into three separate folders: `components`, `services`, and `models`. This structure will suit our smaller application very well but, as always, when organizing your own production solutions, it is always beneficial to think through what may work best in your environment.

To set the stage for our SpyStore implementation, the three folders we will create under `/src/app` will contain the following:

- *Components:* All our React components will be organized into this folder. If necessary, we can further group them into subfolders with associated files such as tests or component-specific CSS styles.
- *Services:* This folder will contain any utility classes we need throughout our client side application. Within the SpyStore example these will primarily be classes responsible for encapsulating interaction with the SpyStore API
- *Models:* This folder will contain TypeScript classes representing the entity models we will be utilizing in the rest of the TypeScript application. We will be making API calls to the SpyStore service and the results will be mapped to these model types to provide us with a level of strong typing.

As we add items to each of the `/src/app` folders, they will be references (or imported) and used by various other components. The WebPack build and development process we put in place earlier in this chapter will automatically pick up these new files and bundle them into our output accordingly. We will not need to perform any extra configuration steps or modify our development process in any way. Simply add files to the appropriate location and utilize those files as needed. If WebPack determines some aspect of our application depends on those files then they are automatically included in the next refresh or publish.

Before we move into the React component implementation, the next two sections will discuss the setup of the various models and services our application will utilize.

Models

As mentioned earlier, one of the great advantages of TypeScript is the ability to write JavaScript applications while relying on a type-safe compiler to help enforce coding standards and lower the possibility of errors. It is possible to still use TypeScript and rely heavily on un-typed (i.e., any) objects but we would continue to be opening ourselves up to a variety of errors and bad coding practices. For this reason, the SpyStore React implementation will be utilizing a set of TypeScript classes that match the SpyStore API return types. This will allow us to cast our JSON API results to these classes and ensure that the rest of our TypeScript code utilizes these models accordingly. If we misspell a property name, the TypeScript compiler will let us know immediately. This might take a bit of extra work but I feel it is worth the overhead to enjoy the benefits of strongly typed models. In many of our projects we automated this process through a variety of tools and IDE plugins that can automatically generate the corresponding TypeScript classes from server side C# class files. One such example of this type of utility is a Visual Studio plugin called TypeWriter: <https://frhagn.github.io/Typewriter/>.

For the SpyStore React implementation, all the models will be hand coded. We will not include the full source of every one of them here but they can all be found in the code associated with that chapter. It is important to note that the TypeScript classes we are implementing to represent our data is not in any way related to React. We will be utilizing these models heavily from React components but React does not dictate how our data is represented. The choice to convert our API results from JSON to model classes is one we have made to streamline our application and not because of a React requirement.

Based on our knowledge of the SpyStore API, we know that every model returned from the API has a few common properties. To simplify our TypeScript, we can add a file called `baseModel.ts` to our `/src/app/models` folder. Within this file we can add the following code:

```
export abstract class BaseModel {
  Id: number;
  TimeStamp: any;
}
```

The `BaseModel` class will serve as the base for the rest of our strongly typed models and keep us from having to add an `Id` or `TimeStamp` property multiple times. If there were any other common properties or functionality that may be beneficial to add at this base level, this is where it would reside.

One of the primary models used by the SpyStore UI is that of displaying products from the SpyStore catalog. The corresponding TypeScript definition for product looks like the following:

```
import { BaseModel } from "./baseModel";

export class Product extends BaseModel {
  Description: string;
  ModelName: string;
  IsFeatured: boolean;
  ModelNumber: string;
  ProductImage: string;
  ProductImageLarge: string;
  UnitCost: number;
  CurrentPrice: number;
  UnitsInStock: number;
  CategoryId: number;
}
```

Note that the `Product` class inherits from the `BaseModel` class and then extends it to add all the properties returned from the API as related to `Product`. We will ultimately be doing a simple cast from an anonymous JSON result to an instance of this `Product` class. This works in our simple scenario as the `Product` class contains only properties and, as long as the JSON properties and our class properties line up, the cast will appear to work fine. If we had the need for our `Product` class to have methods or functions we would need to adjust our API service layer to instantiate new instances of `Product` prior to setting the property values rather than performing a simple cast. This is necessary because the TypeScript `new` keyword would be required to configure the JavaScript objects prototype and extend it with the necessary methods or functions.

The same pattern for created models has been followed for the following model classes:

- `Cart`
- `Category`
- `Customer`
- `Order`
- `OrderDetails`

The remainder of our `SpyStore` React implementation will take advantage of these strongly-typed model classes as we retrieve data from the server side API or we need to manipulate local data. You can see the final implementation of each of the model classes in the provided source code.

Services

Earlier chapters in this book demonstrated the implementation of the common data access code and the API wrappers used by the various user interface implementations. The React code in this chapter will utilize these same API endpoints to read and write all necessary data. As we progress through this chapter and begin to utilize the services described next to read the `SpyStore` data we will need to ensure that the appropriate API services are running and listening on the correct ports.

For the purposes of our React implementation of `SpyStore` we will be implementing a set of TypeScript classes called “services” which will serve as strongly-typed “wrappers” around the `SpyStore` API. Utilizing these service classes in other areas of our application will simplify our usage of the API and will ensure that data is serialized to and from the API correctly. All the service classes will be placed in the `/src/app/services` folder of our project and we will provide a service wrapper for each API endpoint (which, ultimately, corresponds to each ASP.NET API controller developed in Chapter 3).

Each of our API service classes will inherit from a simple abstract class called `BaseService`.

```
export abstract class BaseService {

  getRootUrl() {
    return "http://localhost:40001/api";
  }
}
```

This simple base class currently provides only one utility function called `getRootUrl()`. This function provides derived classes with the means of getting the base URL to the API service. This configuration data could be centralized elsewhere in the application but, for our purposes, putting it within the base class for all API services is a convenient way of managing it.

For making the API calls themselves we will utilize the jQuery Ajax utility. Frameworks such as Angular provide their own native classes (HTTP) to perform API calls but React focuses solely on the UI and provides no such functionality as part of its library. For that reason, if you choose to use React you need to select your own means of interacting with JSON API endpoints from your browser code. As you will see shortly, by simply referencing the jQuery library and utilizing the Ajax methods on the jQuery object, you have access to a very powerful set of capabilities. There are other libraries available to do the same thing and you may want to research which may be the best fit for your codebase.

It is worth noting that we have already added jQuery as a dependent package in our NPM configuration file and that the only features of jQuery we will be utilizing in this chapter are for the use of the Ajax calls to our API. React will be providing us all the necessary capabilities to render our user interfaces and to manipulate HTML. We are using the jQuery library simply for our HTTP calls.

The first “service” we will demonstrate is ProductService. The code for ProductService follows:

```
import * as $ from "jquery";
import { BaseService } from "./baseService";
import { Product } from "../models/product";
import { Category } from "../models/category";

export class ProductService extends BaseService{

  getProducts(): JQueryPromise<Product[]> {
    return $.getJSON(this.getRootUrl() + "/product");
  }

  getFeaturedProducts(): JQueryPromise<Product[]> {
    return $.getJSON(this.getRootUrl() + "/product/featured");
  }

  getProduct(id: number | string): JQueryPromise<Product> {
    return $.getJSON(this.getRootUrl() + "/product/" + id);
  }

  getProductsForACategory(id: number | string): JQueryPromise<Product[]> {
    return $.getJSON(this.getRootUrl() + "/category/" + id + "/products");
  }

  getCategories(): JQueryPromise<Category[]> {
    return $.getJSON(this.getRootUrl() + "/category");
  }

  getCategory(id: number | string): JQueryPromise<Category> {
    return $.getJSON(this.getRootUrl() + "/category/" + id);
  }

  searchProducts(searchText: string): JQueryPromise<Product[]> {
    return $.getJSON(this.getRootUrl() + "/search/" + encodeURIComponent(searchText));
  }
}
```

Let's look at the service implementation. First, there are a few `import` statements that are worth reviewing.

```
import * as $ from "jquery";
import { BaseService } from "../baseService";
import { Product } from "../models/product";
import { Category } from "../models/category";
```

The first `import` statement references the jQuery library and aliases its root module as the `$`. This will provide us strongly-typed access to jQuery with the commonly utilized `$`. If this class happened to be the first class to reference jQuery when scanned by WebPack while bundling our application, WebPack would note that jQuery is now a dependency and would include it in all generated bundles (which it does).

The next three `import` statements are to classes we will need during the implementation of the `ProductService`. As mentioned, the `BaseService` will provide our root URL to this implementation and will be the class we derive from. The `Product` and `Category` models are specific data types the `Product` API deals with. When making calls to the `Product` API the different endpoints either return or accept data of type `Product` or `Category`. As you may have noted from the `ProductService` implementation, every call to this API is an HTTP GET (or a read) so different calls return one or more `Product` or `Category` objects.

Let's look at a single method of the `ProductService` class:

```
getProducts(): JQueryPromise<Product[]> {
    return $.getJSON(this.getRootUrl() + "/product");
}
```

This method encapsulates a call to get a list of all products from the `SpyStore` API. The method returns a result of `JQueryPromise<Product[]>`. This return type specifies that the data itself is not returned directly but instead a "Promise" is returned. The concept of *promises* is very common when dealing with asynchronous operations such as API calls. We won't delve heavily into the concepts of promises in this chapter but we will make heavy use of them.

Before we do that, let's continue to look at the `getProducts` code. The actual body of the method contains only a single line of code. The call to the jQuery `getJSON` method accepts a single parameter: the URL for which to perform an HTTP Get. Using the root URL we retrieve from the base class and appending on the string `'/product'` allows us to build a full URL to the endpoint we want. Performing a Get against this URL will return a JSON result that contains an array of data representing a list of products. The `getJSON` call itself doesn't directly return the JSON but instead returns a promise that will, upon completion of the HTTP call, return the requested data or raise an error (if something does not happen as expected). The TypeScript typing information for jQuery exposes a generic jQuery promise class we can utilize to take advantage of the concept of promises as well as maintain strong-typing with the ultimate result.

Let's look at how a consumer would utilize the previous code with the following example:

```
var prodService = new ProductService();

prodService.getProducts().then(data => {
    //on success
    this.products = data;
},err => {
    //on error
    console.log(err);
},
() => {
    //finally
});
```

This code demonstrates a simple call to `getProducts` and how it utilizes promises to provide the caller with the means of responding to a successful API call or handling any type of possible error. The promise returned from `getProducts` exposes a method called `then`, which accepts multiple functions as its parameters. The first parameter is a function called upon successful completion of the promise. This function has a single parameter of the datatype expected by the ultimate result (in our case an array of `Product` classes). The second function is called when an error occurs and is passed an object with information on the error. The final (and optional) function passed to `then` can be considered a finally block of code and is executed upon either success or error. This function accepts no parameters. In the previous example, we specified all three functions and demonstrated how to keep the data upon a successful API call and how to log the error out to the console if it occurs.

There is much more to the concept of promises but, for purposes of using them with our own API service layer, the previous discussion should explain enough for you to understand how we will use them throughout the rest of this chapter in our `SpyStore` implementation.

Looking back at the rest of the `ProductService` methods, we will find that they are all very simple calls to jQuery's `getJSON` method. Each API call does need to know specific implementation details about the API itself. This includes having inherent knowledge of the URL structure for each call and the structure of the data being sent to the API or returned from the API.

The following API wrapper to get all products for a given category demonstrates one means of passing data to the API itself:

```
getProductsForACategory(id: number | string): JQueryPromise<Product[]> {
  return $.getJSON(this.getRootUrl() + "/category/" + id + "/products");
}
```

In this code the wrapper method accepts a parameter called `id` (which can be either a number or a string). This value is utilized when constructing the necessary URL to get the data.

Not every API call will be an HTTP Get and the jQuery object easily supports all the necessary HTTP verbs such as Get, Put, Post, and Delete. The following example demonstrates the use of a Post to send a complex object to an API endpoint:

```
addToCart(userId: number, productId: number, quantity: number): JQueryPromise<string> {
  var cartRecord = new ShoppingCartRecord();

  cartRecord.CustomerId = userId;
  cartRecord.ProductId = productId;
  cartRecord.Quantity = quantity;

  return $.ajax({
    url: this.getRootUrl() + "/shoppingcart/" + userId,
    data: JSON.stringify(cartRecord),
    method: 'post',
    headers: {
      'Content-Type': 'application/json'
    }
  });
}
```

In the `addToCart` method, we accept three parameters. These parameters are then used to instantiate an object of type `ShoppingCartRecord`. The jQuery Ajax method is used to perform the HTTP Post. In calling this method we pass in an object containing the URL, the data, the HTTP method we want to execute (Post), and any headers we want with the post (in this case the content-type dictating to the API we are passing JSON).

When we specify the data to be posted to the URL, we utilize the `JSON.stringify` function to convert our `ShoppingCartRecord` to valid JSON.

Using these patterns, we have implemented API wrappers in the `/src/app/services` for the following:

- `Cart.Service.ts`
- `Logging.Service.ts`
- `Order.Service.ts`
- `Product.Service.ts`
- `User.Service.ts`

Each of these service classes implements wrapper methods for calls to all the necessary endpoints for the corresponding ASP.NET API controller methods implemented in Chapter 3. You can find the full implementation of each of these services in the final code for this chapter.

Initial Components

Now that our TypeScript models are in place along with our API service layer we will begin putting the necessary React components in place to render the `SpyStore` user interface and to respond to navigation requests and other input from the users. As we do so we will evaluate some of the features of React and demonstrate various real-world scenarios and how React provides developers with a means of supporting these scenarios.

As mentioned earlier in this chapter an application built on the React framework is made up of components. We will be coding our React components using TypeScript and the components will utilize the `TSX` extension allowing us to embed our rich HTML directly into the TypeScript files. Our user interface will be “composed” of various components whereas every individual screen is made up of a component and various individual parts of that screen may also be made up of one or more individual sub-components. The concept of composing a UI out of components is not unique to React and the same general compositional architecture exists in Angular and other frameworks.

Before we begin to implement individual components, let’s first briefly review the various components we will need to implement the `SpyStore` interface:

- *App*: The `App` component can be considered our “shell” or “master page”. This is the root user interface in which other interface components are hosted. It will contain our main navigation and our footer.
- *Cart*: The `Cart` component represents the main shopping cart page. It will display the items currently within a user’s shopping cart.
- *CartRecord*: The `CartRecord` component represents an individual row within a user’s shopping cart. This is a sub-component that will be rendered multiple times from the `Cart` component.
- *CategoryLinks*: This component represents the dynamically rendered links to various product categories. This sub-component will be visible on every page and rendered via the `App` component.

- *ProductDetail*: This component will display information on a single product.
- *Products*: The `Products` component represents the main list of product cards and will be the default page for our `SpyStore` implementation. It will be used to display various product listing such as products by category, search results, or featured products.

Each of these React components will be implemented as a file in the `/src/app/components` folder. Each of the components will, as demonstrated previously, be implemented as a TypeScript class that inherits from `React.Component`.

The first step in putting the full `SpyStore` component structure in place is to create a new TSX file for each of the components in the `/src/app/components` folder. Each TypeScript file should have the name of the component and the extension TSX (such as `Products.tsx`). Within each file, we will simply code a very basic React component that returns some basic HTML containing the name of the component. As an example, the initial implementation of the `Products.tsx` file can be seen in the following code:

```
import * as React from "react";

export class Products extends React.Component<any, any> {
  render() {
    return <h1>Products</h1>;
  }
}
```

Once you have completed setting up a very basic React component for each of the components named in the previous list, we will begin the process of putting a full implementation into each component. Before we do this though, we need to introduce the concept of routing within React and demonstrate how React's routing infrastructure affects which components are currently visible to the users.

Routing

We have previously stated that the React framework focuses solely on the user interface and providing developers with the means of building a robust and maintainable frontend to their applications. Many of the responsibilities that other frameworks embrace, React leaves to others. The concept of "routing" is the ability to keep an interface for an application in sync with the URL of the browser. The React routing framework uses the current URL of the browser to determine which components should be visible to the user and handles the passing of data from one component to another via the browser's URL. The React routing framework provides many powerful capabilities but, for purposes of the React `SpyStore` implementation, we are going to focus on the most commonly used features.

Earlier in this chapter we briefly discussed how React was "bootstrapped" or, how the React framework was initially configured and the initial React component could be rendered into the browser's DOM. At the time this was done using a class within the React framework called `ReactDOM` and a call to the `render` method of this class. This can be seen as follows:

```
import * as React from "react";
import * as ReactDOM from "react-dom";

ReactDOM.render((
  <div>Hello World</div>
), document.getElementById('app'));
```


This code works only if the currently displayed HTML contains an element with the ID of `app`, as we can see in the `index.html` file we have previously added to our project:

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8" />
  <title>SpyStore - React</title>
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link href="/css/spystore-bootstrap.css" rel="stylesheet" />
</head>
<body>
  <div id="app"></div>
</body>
</html>
```

After the call to the `ReactDOM.render` method, the words "Hello World" would be added to the specified `div` tag. This is a simplistic use of the React bootstrapping infrastructure but it does demonstrate the point within our application where we are going to initially configure our React routing.

Our initial routing setup will be done in the `index.tsx` file located in the `/src/app` folder of our project. Previously we had included within this file a simple call to `ReactDOM.render` that displayed the words Hello World to the user. We are now going to replace the contents of `index.tsx` with this code:

```
import * as React from "react";
import * as ReactDOM from "react-dom";
import { render } from 'react-dom'
import { Router, Route, hashHistory, IndexRoute } from 'react-router'
import { App } from "./components/app";
import { Products } from './components/products';
import { OrderDetail } from './components/orderDetail';
import { Cart } from './components/cart';
import { Login } from './components/login';
import { ProductDetail } from './components/productDetail';
import { CheckOut } from './components/checkOut';

ReactDOM.render((
  <Router history={hashHistory}>
    <Route path="/" component={App}>
      <IndexRoute component={Products}/>
      <Route path="/products/:id" component={Products} />
      <Route path="/cart" component={Cart} />
      <Route path="/checkout" component={CheckOut} />
      <Route path="/login" component={Login} />
      <Route path="/product/:id" component={ProductDetail} />
      <Route path="/orderdetail" component={OrderDetail} />
    </Route>
  </Router>
), document.getElementById('app'));
```

This code assumes that we have previously created the base components described in the previous section (even if, at this point, they all simply render their own name).

The first major point to note in the code is that our routing configuration is rendered via a React component called Router that is found in the react-router NPM package, which have previously included in package.json. The <Router> element specified in the index.tsx code is capitalized and thus, as we have previously discussed, this means the React rendering engine assumes this element to be a React component. The specific <Router> component is imported from the react-router library.

The <Router> component is the root of the markup and contains a single configuration parameter that specifies that we want to use hashHistory as our mechanism for tracking browser history. This means our URLs will contain a pound sign and look something like http://localhost:3001/#/products. The other option for tracking history is to set this value to browserHistory and the hash (#) will not be rendered in our URL but we will need to ensure our web server doesn't attempt to resolve all URLs. For purposes of this demo, we are going to use the hash in our URLs.

Within the <Router> component is a nested (and sub-nested) group of <Route> components. The <Route> component is also found in (and imported from) the react-router package. Each <Route> component specifies the URL path it should match and the component that should be rendered if that path is matched.

In the previous example, we have a root <Route> path that looks like the following:

```
<Route path="/" component={App}>
</Route>
```

This root route contains all the other routes in our configuration. This means that the App component (/src/app/components/app.tsx) will serve as the master page and “host” all the other pages. We will demonstrate how this is done shortly.

Nested in the root route are the remaining <Route> elements for our apps basic navigation. The first of these elements is a new React component called <IndexRoute>. This route is defined as:

```
<IndexRoute component={Products}/>
```

The <IndexRoute> essentially defines the default route that will be rendered if a user navigates to http://localhost:3001/. In this scenario, the App component would be rendered and, underneath it, the Products component. For this *nesting* of components to work we will need to make a simple change to our App.tsx file and replace it with the following code:

```
import * as React from "react";

export class App extends React.Component<any,any> {
  render() {
    return <div> { this.props.children } </div>;
  }
}
```

The new implementation of App.tsx is to have the render method return a simple <div> tag that contains the results of evaluating this.props.children. We will discuss the use of React components and props in more detail in the next few sections but for now, note that the React router assigns sub components to this property. The results of a call to the render() method of a child component will be placed here, thus *nesting* the details of the current page within the master page.

The remaining <Route> elements nested under the App route are as specified:

```
<Route path="/products(/:id)" component={Products} />
<Route path="/cart" component={Cart} />
<Route path="/checkout" component={CheckOut} />
```

```

<Route path="/login" component={Login} />
<Route path="/product/:id" component={ProductDetail} />
<Route path="/orderdetail" component={OrderDetail} />

```

Each should be self-explanatory. If a user navigates to `http://localhost:3001/Cart` then the `Cart` component will be rendered inside the `App` component and displayed to the user.

There are two `<Route>` components in the previous list that are worthy of reviewing. The first is the configuration for the `/product` route:

```

<Route path="/product/:id" component={ProductDetail} />

```

The route configuration specifies that a user navigating to the `/product` page will need to specify an additional route parameter. In this case, the parameter is the `id` of the product we want to view the details for. Note the use of the semicolon to specify the route parameter. The name of the parameter (in this case `id`) is arbitrary but will be used later when we implement the `ProductDetail` component and want to retrieve this value so that we can display the correct information.

The second `<Route>` component worth reviewing is that of the `/products` page. This configuration looks like the following:

```

<Route path="/products(/:id)" component={Products} />

```

This route configuration also specifies a parameter called `id` but, in this instance, the parameter is surrounded by parentheses, which indicate that the parameter is optional. We will need to account for the possibility that it hasn't been specified when we implement the `Products` component later in the chapter.

At this point the routing configuration configured directly within the `index.tsx` file tells React how it should handle the various URLs and how these URLs translate into components to be rendered. As we implement each of the components we will review some additional features of the React routing infrastructure and how they can be used to easily render anchor tags for navigation and how we can look for and handle parameters passed into a component from the URL.

App Component

With our routing infrastructure in place we need to implement our main navigation and UI. We will do this by fully implementing the `App` component (`/src/app/components/app.tsx`). In doing so we will also introduce several new React concepts that will appear in most (if not all) of the remaining component implementations.

Due to the length of the raw HTML necessary to render the full `SpyStore` React implementation, much of the markup itself will be abbreviated in this text. We will instead focus on key elements of the React implementation to demonstrate how React works. To see the full implementation, I encourage you to review the source code provided with this chapter.

The first section of the `App` component looks like the following:

```

export class App extends React.Component<any, any> {
  componentWillMount() {
    this.setState({ categories: [] });
  }

  componentDidMount() {
    this.loadCategories();
  }
}

```

```

loadCategories()
{
  var _service = new ProductService();

  _service.getCategories().then((data) => {
    var categories: Category[] = data || [];

    this.setState({ categories: categories });
  });
}

```

There are some new concepts to be discussed in the previous code, which, at its core, is responsible for using our API wrapper to load a list of categories. We will use this list of categories to render the main navigation for the SpyStore site.

First, we see that we have two small methods called `componentWillMount()` and `componentDidMount()`. These methods are part of the React component lifecycle and it's very common that you will have the need to put certain types of initialization code within methods such as these.

The React component lifecycle is very straightforward and, during this process, the React framework calls certain methods on all React components at specific points when creating them and adding them to the DOM (a process it calls *mounting*) and then when it needs them to update their state and potentially their UI (a process called *updating*).

When initializing (or mounting) React components, the following component methods are called by React in this very specific order:

- `constructor()`
- `componentWillMount()`
- `render()`
- `componentDidMount()`

When React needs to update the component tree and have all (or specific) components update themselves, the following component methods are called. Again, the order of these calls is very specific:

- `componentWillReceiveProps()`
- `shouldComponentUpdate()`
- `componentWillUpdate()`
- `render()`
- `componentDidUpdate()`

There is another lifecycle method worth pointing out and it is used when React needs to remove a component from the DOM. This method is called `componentWillUnmount()`.

During our implementation of the SpyStore UI, we will demonstrate common uses of many of these lifecycle methods.

The App component initially provides its own implementation of the `componentWillMount()` method. As we have just seen, this method is called prior to a component being “mounted” (or added to the DOM). The App implementation of this method looks like the following:

```

componentWillMount() {
  this.setState({ categories: [] });
}

```

Every React component can manage its own state through some specific capabilities of the core `React.Component` class. Each React component has access to a property of this base class called `state`. When we initially declared the `App` component itself we specified that we wanted to inherit (or extend) the `React.Component` class. This declaration also specified two generic type parameters. In our declaration, we simply specified both as type `any`.

```
export class App extends React.Component<any, any> {
}
```

These generic parameters are used to specify the types used for a React component's properties and state. The concept of both properties and state are key to efficiently using React components and we will demonstrate and discuss both heavily when looking at each of the `SpyStore` React components. Our reason for pointing these parameters out now is that, by specifying both as type `any`, we are essentially telling React that, for our `App` component, we are going to forgo providing a concrete type or interface for either our properties or our state. Instead we are going to use them as “un-typed” and set them to whatever we want within our component. Some may consider this a bad practice but, for purposes of this book, it will save some extra steps in demonstrating these concepts. It is also worth noting that the first generic parameter specifies the type of the `Properties` for the component (which we haven't addressed yet). The second parameter specifies the type of the `State`.

With regards to a React component's state, every component has a `state` property, which can simply be accessed directly off the base class (i.e., `this.state`). Every component also has a function to set the state to a new value called `setState()`. A very important concept within the use of state inside a React component is that the state be considered immutable. This means that access to the state property should be considered read-only. Any modifications to this state should be done via a call to `setState(newState)`. We won't go into detail about the concepts of immutability here, but it is worth noting that React relies on the state of its components to only be changed during a call to `setState()`. This also includes any properties of state. Any changes at all to any piece of a component's overall state should not be made by directly changing the data off the state property itself. Instead, a copy of the state object should be made, the new value(s) set accordingly, and then the newly updated copy of the state passed to a call to `setState()`.

Since a React component is just a TypeScript class and is instantiated in memory and kept around if the component is visible, it would be fair to ask why we need to use this “state” infrastructure and why data should just be tracked as instance-level fields or properties available within our component class. This is possible and sometimes even preferred if the data itself has no effect on how a component renders. React watches for changes to a component's managed state (via calls to `setState()`) and uses these changes as triggers to re-execute a component's update cycle and potentially re-render any changes to the DOM. So, the generally accepted best practice is to track any component data utilized for purposes of rendering a UI as state. Any other utility objects or other internal data that doesn't affect a component's UI can be stored outside the state object but within the component class itself as necessary.

Within the `App` component's `componentWillMount()` method, we simply initialize our component's state to contain an empty array of categories with the following line of code:

```
this.setState({ categories: [] });
```

Then, when the component has been fully mounted and added to the DOM, we use our previously implemented `ProductService` API utility to make a call to the `SpyStore` API for purposes of getting a list of categories. Upon successful completion of that call, we again call the `setState()` method to update our component's state with the resulting data.

```
componentDidMount() {
  this.loadCategories();
}
```

```
loadCategories()
{
  var _service = new ProductService();

  _service.getCategories().then((data) => {
    var categories: Category[] = data || [];

    this.setState({ categories: categories });
  });
}
```

As we will see in the `App` component's `render()` method, this category data will be passed down to a sub-component for actual rendering.

The `render()` method for the `App` component is straightforward in that it contains the basic markup for our UI “shell”. Per our previous routing configuration, this component is rendered with every page within our site and “hosts” other components that provide the specific page interfaces. Near the middle of the `App` components markup you will see the following:

```
<div className="panel-body">
  {this.props.children}
</div>
```

It is within this area of the `App` component markup that child components will be rendered.

Another new concept visible in the `render()` method of this component is the navigation. Since `App` is meant to be our overall UI “shell,” it makes sense that we will find some of our global navigation represented within its markup. To make it easy to render the appropriate links, we need to be able to easily integrate with our previously configuring routing setup. Fortunately React makes this very easy.

At the top of the `App.tsx` file we have imported a new component from the `react-router` package called `Link`, as follows:

```
import { Link } from 'react-router'
```

Then, within our markup we can use this new component in places where we would have utilized HTML anchor tags. The `Link` component:

```
<Link to="cart" ><span className="glyphicon glyphicon-shopping-cart"></span> CART</Link>
```

The `<Link>` tag will render a fully functional anchor tag that is aware of the React routing configuration and will navigate to the proper route accordingly (as specified by the `to` property of the `Link` component). When using React's routing infrastructure, the `Link` component is the proper way to render anchor tags in your markup.

The final piece worth noting within the `App` component is the use of a sub-component (`CategoryLinks`) to render the individual links to various categories. To use a sub-component our first requirement was to specify our dependency on it via an `import` at the top of our `App.tsx` file like this:

```
import { CategoryLinks } from "./categoryLinks";
```

Once the dependency has been specified with the `import` we are able to use this component in our markup. You will see the `App` component specify the use of the `CategoryLinks` sub-component in its markup with the following line:

```
<CategoryLinks categories= { this.state.categories } />
```

Remember that the capitalization of the tag name indicates to React (or, more specifically, the TSX/JSX parser) that the tag is a component and not a raw-HTML tag, which should be rendered as is.

The `CategoryLinks` component is passed a single property called `categories`. This property is passed as a parameter directly in the markup and the actual value of this property is assigned to the value of `this.state.categories`. You should recognize this as using our previously loaded `categories` we stored within our component's state object. Another important thing to note in the markup that assigns the `categories` property is the lack of quotation marks around the `{ this.state.categories }`. Using the curly brackets (`{}`) is all that is required and the TSX parser will automatically assign the right object (an array in our case) to the properties of the sub-component. There are other scenarios when working with markup in React components where quotation marks are required.

CategoryLinks Component

In the `App` component we discussed previously, we added a reference to the `CategoryLinks` component and declared it within the markup. In doing so we also passed this new component a single property called `categories`. Properties are another key concept to understand when working with React components. Whereas `state` is the concept of a single React component managing its own data (in an immutable way), `properties` are the ability for a component to pass data down to child components. A React component's state may update over time whether due to loading data from an API or user interaction. The properties a component receives from its parents can never be changed by the component itself. Any changes to the properties received from a parent should be triggered by the parent.

To simplify: A React component's state can change, so the properties it receives should be treated as read-only. Furthermore, both state and properties are only visible within a component itself. Their values should be considered encapsulated within the component. There is means of communication between components (such as events), which we will discuss later, but it is important to note that just changing your component's state does not make these changes visible to others.

To provide an example of how a React component can access properties passed to it from a parent, let's review the code of the `CategoryLinks` component:

```
import * as React from "react";
import { Category } from "../models/category";
import { Link } from "react-router";

export class CategoryLinks extends React.Component<any, undefined> {
  render() {
    var categories: Category[] = this.props.categories;

    var links = categories.map((category, index) => {
      var url = "products/" + category.Id;

      return (<li key={ index }>
        <Link to={ url }>{ category.CategoryName }</Link>
      </li>)
    });

    return (<ul className="nav nav-pills hidden-sm" key="1">
      { links }
    </ul>)
  }
}
```

One of the first things to note is that our generic type parameters are specified a bit differently than was done with the `App` component. The `CategoryLinks` component specifies the second parameter (used for state) as `undefined`. This simply means we are not planning on needing state within this component and the type checking of the TypeScript compiler will warn us if we try (as we would get compiler errors by accessing an undefined value).

```
export class CategoryLinks extends React.Component<any, undefined> {
}
```

The next item worth noting in this component is how easy it is for a child component to retrieve properties passed down from a parent. This is done through accessing an object called `props` directly off the base class, as shown in this line of code from the `render()` method:

```
var categories: Category[] = this.props.categories;
```

The fact that we have a value on this object called `'categories'` is due to the choice of name in the `App` component when we declared the `CategoryLinks` component in markup. Since we are treating the `'props'` object of the `CategoryLinks` component as un-typed (i.e., `any`) there is no type checking done by the TypeScript compiler to ensure these names match so it is up to us as developers to be careful. In a real-world environment, it is generally preferable to declare interfaces for both the state and properties of your React components. By specifying these interfaces as your generic type parameters when declaring your components, the TypeScript compiler will help ensure you always assign and read the values correctly.

Next, let's review the rest of the `CategoryLinks` `render()` method:

```
var links = categories.map((category, index) => {
  var url = "products/" + category.Id;

  return (<li key={ index }>
    <Link to={ url }>{ category.CategoryName }</Link>
  </li>)
});

return (<ul className="nav nav-pills hidden-sm">
  { links }
</ul>)
```

The goal of this code is to render an unordered list (i.e., ``) that contains a link for every category. The first block declares a locally-scoped variable called `"links"` and uses the `map` method of JavaScript arrays to iterate over every category in our collection to build up a list of items. Notice that within the `map` method we build up a local URL variable to track the URL for a specific item and then, within the `return` statement for that block, we use that URL when generating the actual link (via the `'react-router'` `<Link>` component).

The biggest thing to point out in the generation of the `links` variable is the fact that the `map` method iterates over all the category objects and returns an array that contains markup. Let's look at some key elements of the lines:

```
var url = "products/" + category.Id;
return (<li key={ index }>
  <Link to={ url }>{ category.CategoryName }</Link>
</li>)
```


First, we wrap the markup itself with parentheses and the JSX parser will automatically recognize it as markup and render an array of the right type because of our `map`. Second, the `key` attribute of our `` tag is required by the React DOM parser. When we use React to render HTML the React framework builds an in-memory representation of the DOM (or a “virtual” DOM) and compares it in real time to what is currently rendered in the browser. If it detects a change it will quickly update what is visible to the browser. In some scenarios (mainly lists of items such as our category links) it needs to be able to uniquely identify individual items in the list so that it knows if new ones have been added or items have been removed. To do this efficiently requires that React force developers to specify a unique key for each item in an array of DOM elements. In this code, we simply use the index of the item in the array of Categories as the key. In other scenarios, we may use a database generated ID or other unique value for data when rendering a list of DOM elements.

Finally, when rendering the actual `<Link>` element itself we can utilize any in-scope variable to assign the right attributes or properties. We demonstrate this by generating our URL prior to the return statement and using it to assign the `to` property of the `<Link>` element. When the JSX parser evaluates statements found in the curly brackets of markup (i.e., `{ url }`), it evaluates any JavaScript expression in the current scope. Since the JSX parser will evaluate any JavaScript expression the return statement could also be written as:

```
return (<li key={ index }>
  <Link to={ "products/" + category.Id }>{ category.CategoryName }</Link>
</li>)
```

The final line of code in the `CategoryLinks` component is simply another return statement to wrap the list of category links in an HTML `` tag, as shown here:

```
return (<ul className="nav nav-pills hidden-sm" >
  { links }
</ul>)
```

The result is that the header of our `SpyStore` UI now displays a nicely formatted list of category links, as seen in Figure 9-12.

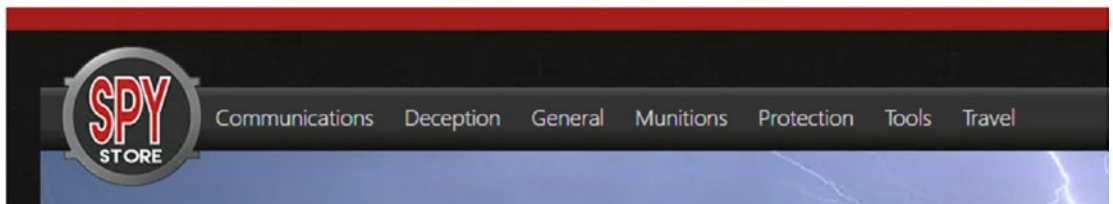


Figure 9-12. *Category links*

Products Component

Now that the `App` component, or the user interface “shell,” has been implemented, we will move on to our first “screen” component. We will start with the `Products` component, which will be responsible for displaying a list of products retrieved from the API in a card layout. An example of this interface can be seen in Figure 9-13.

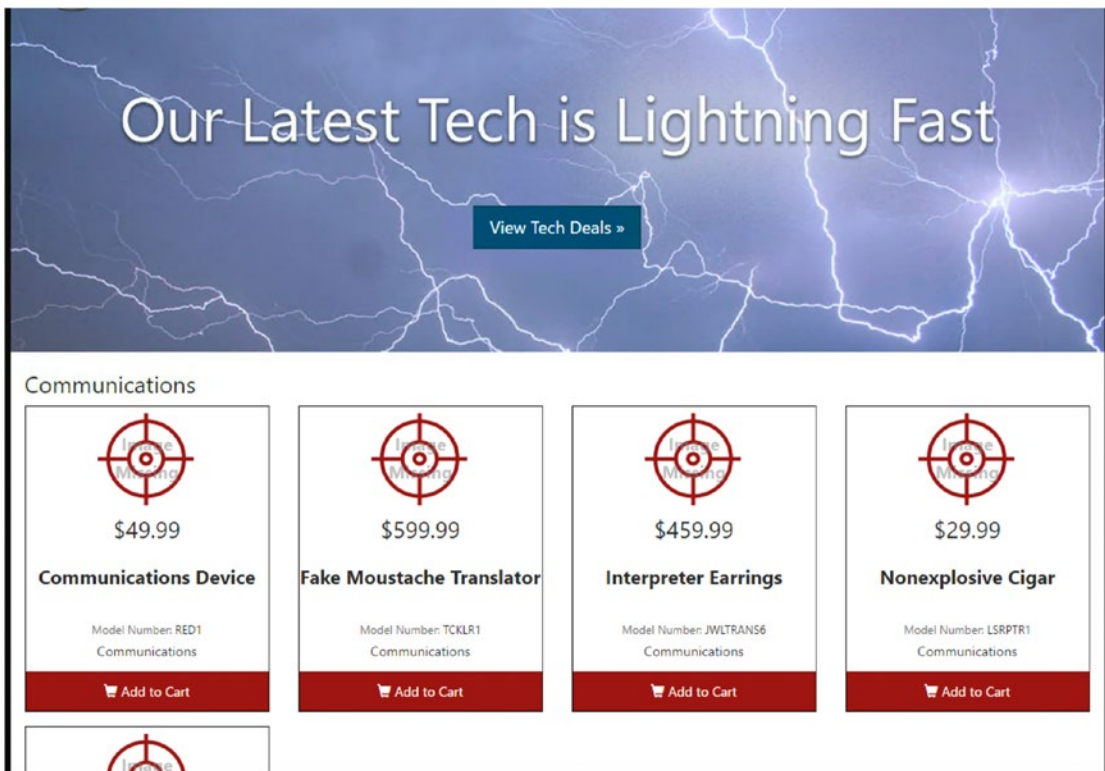


Figure 9-13. Products interface

As with all our components, the Products component will be implemented in file called `Products.tsx` in the `/src/app/components` folder of our solution.

The Products component will utilize `ProductsService` to interact with the remote API to retrieve the appropriate list of products for display. There are three scenarios in which the Products component will be used to retrieve data. These are:

- Get a list of featured products
- Get a list of products specific to a single category
- Get a filtered list of products based on the results of a user-initiated search

Each of these scenarios is covered by one of the three methods of the Products component. Each calls the appropriate API method it requires via the `ProductService` we implemented earlier. Upon a successful call to the API, each method updates the state of the Product component to have the appropriate list of products and a value for the header. We will display this header at the top of the screen accordingly.

```
loadFeaturedProducts() {
  var service = new ProductService();
  this.categoryID = null;
  service.getFeaturedProducts().then((data) => {
    this.setState({ header: "Featured Products", products: data});
  });
}
```

```

loadCategoryProducts(categoryId: string) {
  var service = new ProductService();
  service.getProductsForACategory(categoryId).then((data) => {
    this.setState({ header: "Category", products: data });
    this.updateCategoryName();
  });
}

searchProducts(searchText: string) {
  var _service = new ProductService();
  _service.searchProducts(searchText).then((data) => {
    this.setState({ header: "Search Results", products: data});
  });
}

```

To support the appropriate calling of each of these methods for loading data, we will again leverage some of React's built-in component lifecycle methods. These are shown here:

```

componentWillMount() {
  this.setState({ header: "", products: [] });
}

componentDidMount() {
  this.refreshProducts();
}

componentDidUpdate(prevProps, prevState) {
  if (this.props.params.id !== prevProps.params.id) {
    this.refreshProducts();
  }
}

```

As before, the code in `componentWillMount()` simply initializes our component state. The code in the next two lifecycle methods (`componentDidMount()` and `componentDidUpdate()`) make calls to `refreshProducts()`. While we have seen `componentDidMount()` previously, the method `componentDidUpdate()` is new. It is called when there are changes to a component's properties as passed down from a parent component. It is not called when a component is initially mounted with the initial properties. Instead it is only executed on a change to existing properties. For this reason we need to call `refreshProducts()` from both the previous methods. Note that the `componentDidUpdate()` accepts two parameters: `prevProps` and `prevState`. This information gives us the data to selectively perform our update only if a change to the `id` parameter has occurred.

The remaining two utility methods we will need within the `Product` component are the `refreshProducts()` method and a method to update the category name on the header (which is necessary as our URL route only contains the ID of a selected category and not the full name). These methods are shown here:

```

categoryID: number | null = null;

refreshProducts() {
  if (!this.props.params.id) {
    this.loadFeaturedProducts();
  }
}

```

```

    else if (this.props.params.id !== this.categoryID) {
      this.categoryID = this.props.params.id;

      this.loadCategoryProducts(this.props.params.id);
    }
  }

  updateCategoryName() {
    if (this.categoryID !== null) {
      var service = new ProductService();

      service.getCategory(this.categoryID).then((data) => {
        if (data && data.CategoryName) {
          this.setState({ header: data.CategoryName, products: this.state.products });
        }
      });
    }
  }
}

```

The `refreshProducts()` method should be fairly self-explanatory. If we do not find an ID in our route parameters (from the URL), we make a call to the `loadFeaturedProducts()` method to get a default list of products. If the ID is found we load products specific to that category. One unique element of this code is that the `Products` component class itself maintains a class-level variable called `categoryID`, which we use to keep track of the currently displayed category. If we attempt to refresh the page with the same category information we skip the call to load data from the API and, instead, leave the currently displayed data intact. This is a performance optimization to minimize any unnecessary calls to the API.

The `updateCategoryName()` method takes the currently displayed category ID from the class-level variable described previously and uses it to make another API call to get details about the product. If this call is successful, the component's state is updated to reflect this name.

The final key piece of the `Products` component is the `render()` method responsible for actually displaying our list of products to the user. The full implementation of the `Products render()` method is shown here:

```

render() {

  var products = this.state.products.map((product) => {
    var imageUrl = '/images/' + product.ProductImage;
    var isCurrentCategory = this.categoryID === product.categoryID;

    return (<div key={product.Id} className="col-xs-6 col-sm-4 col-md-3">
      <div className="product">
        <img src={imageUrl} />
        <div className="price">${ product.CurrentPrice.toFixed(2) }</div>
        <div className="title-container">
          <h5>{ product.ModelName }</h5>
        </div>
        <div className="model-number"><span className="text-muted">Model Number:</span>
        {product.ModelNumber }</div>
        {(isCurrentCategory) ? (
          <Link to={ 'products/' + product.CategoryId } className="category">{
            product.CategoryName }</Link> ) :

```

```

        ( <div className="category">{ product.CategoryName }</div> )}
        <Link to={ 'product/' + product.Id } className="btn btn-primary btn-cart">
        <span className="glyphicon glyphicon-shopping-cart"></span> Add to Cart</Link>
      </div>
    </div>
  });

  return <div>
    <div className="jumbotron">
      <Link to="products" className="btn btn-info btn-lg"
        dangerouslySetInnerHTML={{ __html: "View Tech Deals &raquo;" }}></Link>
    </div>

    <h3>{ this.state.header }</h3>

    <div className="row">
      { products }
    </div>
  </div>
}

```

The first section of the `render()` method takes the list of products and maps the result back to the markup necessary to display the collection of cards. This is like how the `map` function was used earlier in the `CategoryLinks` component.

A few interesting things to point out about how React handles generating this list. First, each product has an image associated with it and we generate the correct URL for a products image in this loop. When we want to render the image it can be done with a simple image tag such as the following:

```
<img src={imageUrl} />
```

Note the lack of quotation marks around the curly brackets surrounding the `imageUrl` variable name.

Next, since each product has a price stored as U.S. dollars we use the following code to render the display. There are numerous JavaScript libraries to do proper currency display for various cultures. The following code simply uses the built-in JavaScript functionality to display a fixed set of decimal places. We even go as far as to hardcode the `$` character prior to the currency value.

```
<div className="price">${ product.CurrentPrice.toFixed(2) }</div>
```

As we are displaying an arbitrary list of product items on this screen, we have the possibility of displaying a category link on every product card. This makes sense if the current items are not all from the same category (which we store in our `Product` class). But, if a product being displayed is from a category of products other than the current one, it would be nice to allow the user to quickly refilter their results to show products from that category.

Inside our `map` loop, we declare a Boolean variable that tells us if the current product belongs to the current category:

```
var isCurrentCategory = this.categoryID == product.categoryID;
```

Then, within our markup, we use this Boolean to conditionally determine whether to render a link or just a label with the category name:

```
{(isCurrentCategory) ?
(<Link to={ 'products/' + product.CategoryId } className="category">{ product.CategoryName
}</Link> ) :
(<div className="category">{ product.CategoryName }</div> )}
```

This snippet of React markup is somewhat unusual at first glance but is a good example of how a JavaScript expression (`isCurrentCategory`) can be evaluated to determine which of two blocks of markup should be rendered. Notice the parentheses around each block to specify to the JSX parser where one block starts and ends.

The final snippet of the `Products` component is the main return that displays the core of the interface. This code follows:

```
return <div>
  <div className="jumbotron">
    <Link to="products" className="btn btn-info btn-lg" dangerouslySetInnerHTML={{ __html:
      "View Tech Deals &raquo;" }}></Link>
  </div>

  <h3>{ this.state.header }</h3>

  <div className="row">
    { products }
  </div>
</div>
```

It is here that we render the banner at the top of the page, display the header, and then insert our previously generated list of product cards.

The only item of note in this markup is the line used to display the button in the middle of our `jumbotron` image.

```
<Link to="products" className="btn btn-info btn-lg" dangerouslySetInnerHTML={{ __html: "View
Tech Deals &raquo;" }}></Link>
```

Generally, the React rendering engine makes it impossible (or at least very difficult) to inject harmful HTML or scripts into your pages. When you have the need to render some content that you know to contain markup you will need to take a few extra steps to let the framework know of your intent. In this example, we use a property called `dangerouslySetInnerHTML` to specify some markup to display. This is React's alternative to the `innerHTML` property in HTML. It also requires you pass in an object with an `__html` property (two underscores preceding `html`). This is their way of forcing you to remember your decision and proactively ensure that rendering raw markup is your intent.

ProductDetail Component

The next React component we will discuss in this chapter is the `ProductDetail` component. This component, shown in Figure 9-14, is used to display information on a single product. It is generally reached from a link on the `Products` component page. When reviewing the key code for this component, we will demonstrate how React handles events such as clicking a button and how it can access data the user provides via fields on a form (i.e., the product quantity).



Figure 9-14. *ProductDetail* interface

The first thing to note about the `ProductDetail` component is that its state will consist of three parts: the product model being displayed, a quantity value tracking how many of this product a user wants to buy, and a reference to a user (or customer).

■ **Note** For purposes of this book and the sample applications we are providing, we are not going to concern ourselves with a robust security implementation. This choice was made for brevity within this text and not because security would not play a key role in an ecommerce system such as the `SpyStore` demo we are developing.

The `SpyStore` API we have demonstrated earlier in the book does support the concept of a customer and the API endpoints we will need for this component and the cart components will require we provide this data. For that reason, we will call the API to retrieve the first user in the results as the user we will emulate through the ordering process.

The initial lifecycle events of the `ProductDetail` component are as follows:

```
componentWillMount() {
  this.setState({ product: {}, quantity: 1, user: {} });
}

componentDidMount() {
  var service = new ProductService();

  this.loadUser();

  service.getProduct(this.props.params.id).then((data) => {
    this.setState({ quantity: 1, product: data, user: this.state.user });
  });
}

loadUser() {
  var userService = new UserService();
```

```

    userService.getUsers().then((data) => {
      if (data && data.length>0)
      {
        this.setState({ quantity: this.state.quantity, product: this.state.product,
          user: data[0]});
      }
    });
  }
}

```

This code configures our component’s state and loads the appropriate user and product information from the API. The ID for the current product is passed into this component from its parent component, as we have seen in other component interaction scenarios.

The `render()` method of the `ProductDetail` component contains a single line of code that returns all necessary HTML markup to display our product details with the appropriate formatting. Within this markup there are a few new React concepts worth pointing out: the ability to accept input from a user via an HTML input tag and the ability to respond to user events such as clicking on DOM elements.

First, regarding allowing a user to input the quantity of an item before adding it to their cart, let’s review the following input element:

```



```

The previous element binds its `value` property to the quantity value we have added in our component’s state. The other new concept demonstrated in the line is the ability to fire an event upon a user changing a value in this input field.

First, the name of DOM events within React is always camel-cased. In this scenario, the event is called `onChange`. Second, instead of passing in a string with the name of the event (as we do in HTML) with React we use the TSX syntax to bind the event to the appropriate event. In our example, we use the arrow syntax to specify the expression we want to execute when the event fires. The parameter (`e`) we pass into our method represents the `SyntheticEvent` class and is used by React to provide information on the event and its target. This optional parameter is specified by the W3C as a standard and you would use it for such things as getting the current value of an input control once it is changed. Let’s look at the code for the `quantityUpdated()` method:

```

quantityUpdated(event) {
  this.setState({quantity: event.target.value > 0 ? event.target.value : 0 , product:
    this.state.product, user: this.state.user});
}

```

This method accepts a single event parameter and simply updates the state of the current component. When updating the component’s state, it leaves the product and user properties the same (setting them to their current values) but updates the quantity value to reflect what the user has input into the quantity input field. This value is retrieved by utilizing the event parameter’s `target` property. This value is set to the HTML element the event was ultimately fired from. Using this element’s `value` property, we can retrieve (or update) the value of the input field as necessary.

A common “gotcha” when dealing with React events is the binding of the `this` variable within the event handler itself. In the previous example, we bound the `onChange` event to an arrow function (`"(e) => this.quantityUpdated(e)"`). This syntax was necessary for us to have access to the current object instance (`this`) within the event handler itself. We used the `this` reference to call `setState()` and perform the required action within our event handler.

To see an alternative syntax for binding an event let's look at how React allows us to handle the click event of the Add to Cart button:

```
<button className="btn btn-primary" onClick={this.addToCart}>Add to Cart</button>
```

In the markup, we simply bind the `onClick` event directly to a method without the arrow syntax. The `addToCart` method we use to handle this click event is as follows:

```
addToCart() {
  var cartService = new CartService();

  if (this.state.user && this.state.quantity>0) {
    cartService.addToCart(this.state.user.Id, this.state.product.Id, this.state.
      quantity).then((data) => {
      hashHistory.push("/cart");
    }).fail((err) => {
      if (err.responseJSON.Error)
      {
        alert(err.responseJSON.Error);
      }
    })
  }
}
```

First, we notice that this method does not accept any parameters. The ability to accept the event parameter is optional and not always necessary. With the `addToCart()` function, we do not need to know anything specifically about the event so we simply do not use this parameter. Second, within this method we use the `CartService` to call the Cart API and set up information about the product, the user, and the quantity of the product we'd like to add. This piece should be self-explanatory. To get the necessary values from our component's state we need to reference the current component via the `this` keyword. Unfortunately, this will not work due the "gotcha" described previously of needing to perform some extra work to use `this` in React event handlers. To solve this problem we must "bind" the `addToCart()` method to `this` with an extra line of code traditionally added to a component's constructor. This can be seen in the constructor of the `ProductDetail` component:

```
constructor() {
  super();
  this.addToCart = this.addToCart.bind(this);
}
```

Once this line has been added, our `addToCart()` method works as expected. Thus, to handle events in React they must be bound using the arrow syntax (in which case using `this` in the handler will work as expected) or we must take the extra step of binding the event handler method to `this` in our constructor (an extra step which then simplifies our syntax of binding the method).

Another small quirk worth pointing out regarding the use of events in React is that they do not return any value. Other platform's events sometimes return a Boolean value indicating whether the browser should continue to bubble this event up to higher level elements that may be listening. React uses the previously mentioned `SyntheticEvent` parameter to achieve this goal. To stop an event from propagating higher up the DOM tree, we would simply use the following line of code in our event:

```
event.preventDefault();
```

The React event system abstracts many (but not all) of the DOM events and provides a higher-level means of reacting to them within your React code. The use of a SyntheticEvent system allows React to hide platform specific implementations of events and to provide a consistent syntax and set of properties for accessing them. For a full list of the events supported by React (and the properties you can get for each one within its handler), see the following link: <https://facebook.github.io/react/docs/events.html#supported-events>. Our examples only demonstrated two of the most common React events: onChange and onClick.

One final line of code worth mentioning on the ProductDetail page is in the addToCart() method. Upon returning a successful value after calling the API to add an item to the current user's cart, we need to navigate the user to the CartComponent UI (which we discuss next). To do this, we use a component called hashHistory, which we import from the react-router package with the following line:

```
import { hashHistory } from "react-router";
```

Once the hashHistory component has been imported, we can use it to “push” a new route onto the stack and the React router will update the UI accordingly. We do that in addToCart() with the following line:

```
hashHistory.push("/cart");
```

It is also possible to pass parameters to the next route via the push() method. In the previous scenario, we do not need to pass any additional information and, instead, just need to display the Cart component.

Cart Component

The last major React component we will review in the SpyStore.React solution is the component used to display the shopping cart information to the user. Most React concepts used in this component have been demonstrated previously, although there are some new concepts worth reviewing.

Before we get into the implementation of the Cart component (and its child, the CartRecord component), let's look at the final UI we will be implementing. This can be seen in Figure 9-15.

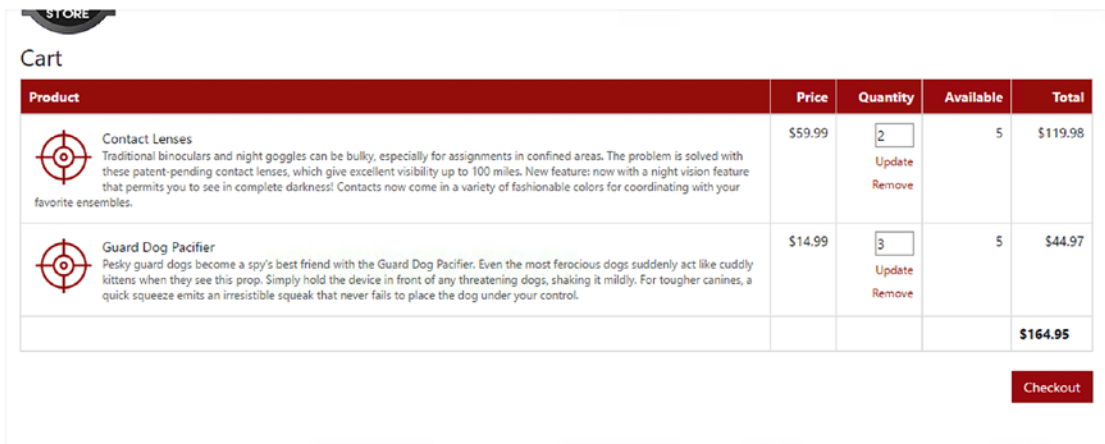


Figure 9-15. Cart interface

The Cart component will utilize the CartService to read all current records in a user's shopping cart and display them accordingly. Users will be allowed to remove items from their carts or update the quantities of items within their carts. The Cart component itself will render the top level of the table and be responsible for all the data access. Each individual row in a user's shopping cart will be rendered via a child component called CartRecord. The CartRecord component will handle user import regarding changing an item's quantity and removing an item from the cart. These events will be passed back from an individual CartRecord component to the parent Cart component, where the appropriate data access calls will be made to the API and the screen refreshed with updated data.

First, let's look at some of the setup code for the Cart component:

```
componentWillMount() {
  this.setState({ total: 0, items: {}, user: {}});
}

componentDidMount() {
  this.loadUser();
}

loadUser() {
  var userService = new UserService();
  var loggingService = new LoggingService();

  userService.getUsers().then((data) => {
    if (data && data.length>0)
    {
      this.setState({ total: this.calculateTotal(this.state.items), items: this.state.items, user: data[0]});

      this.loadCart();
    }
  }).fail((err) => {
    loggingService.logError(err);
  });
}

loadCart() {
  var _service = new CartService();
  var _loggingService = new LoggingService();

  if (this.state.user) {
    _service.getCart(this.state.user.Id).then((data) => {
      if (data) {
        data.forEach(item => item.OriginalQuantity = item.Quantity);

        this.setState({total: this.calculateTotal(data), items: data, user: this.state.user});
      }
    }).fail((err) => {
      _loggingService.logError(err);
    });
  }
}
```

```

    }
  }

  calculateTotal(items: ShoppingCartRecord[] | undefined): number {
    var total = 0;

    if (items && items.length>0)
    {
      items.forEach((row: ShoppingCartRecord) => {
        total += row.LineItemTotal;
      });
    }

    return total;
  }

```

The Cart component’s setup methods dictate that we will have a basic component state: the items in our cart, the current user, and a calculated total of the items. The `loadUser()` method will, again, simply load the first user returned from the API as we are not implementing a robust security infrastructure. The `loadCart()` method will use the `CartService` to load the current users shopping cart records. If they are returned successfully, we assign them to the component state and use the `calculateTotal()` function to get the sum of all shopping cart records. In the `loadCart()` record, we also update each cart record to keep track of its original quantity. This is so that we can optimize our client side code to not attempt to update a record’s quantity if it hasn’t changed from its original value.

Let’s now look through the `render()` method of the Cart component. The first section of the `render()` method maps the current shopping cart items to a set of table rows that will ultimately display them. Each row is ultimately managed by a child component called `CartRecord`. This first part of the `render()` method can be seen in the following code:

```

var cartRows: any[] = [];
var records: ShoppingCartRecord[] = this.state.items;

if (records && records.length > 0) {
  cartRows = records.map((record: ShoppingCartRecord) => {
    var rowKey = record.ProductId + "." + record.Quantity;

    return <CartRecord key={ rowKey } item={ record } onRowDeleted={ (record) => this.
      rowDeleted(record) }
      updateQuantity={ (record) => this.updateQuantity(record) }/>;
  });
}

```

Within the function mapping a shopping cart record to the appropriate markup we initially build our own value to be used by the row key. This value is a string represented by the ID of the product and its quantity. By using a hybrid of these two values, we can bypass React’s attempt to reuse components in a collection in which the key doesn’t change. This optimization is generally good but, in our scenario, when we reload data and the quantity has changed we want React to remove the old row and add a new one. There are several ways to accommodate this but generating a unique row key serves our purposes. Next, we generate a number of `CartRecord` components and assign a key, the product the row represents (“item”) and then assign some handlers to a few custom events the `CartRecord` will raise: `onRowDeleted` and `updateQuantity`. These events are not native React events and are instead just events we will define in `CartRecord` and trigger when a user clicks the Update or Remove links on an individual product row.

As mentioned, we wanted to centralize all data access within the `Cart` component. The `CartRecord` component will be simply for display and basic data management. Its implementation will react to user input on its individual row and will raise the right events when a call to the API needs to occur.

The assignment of the arrow style methods associates them as properties in the `CartRecord` component. Inside the `CartRecord` component we can call these assigned functions with code like the following (taken from `CartRecord`):

```
updateQuantity() {
  if (this.props.updateQuantity) {
    this.props.updateQuantity(this.state.item);
  }
}

removeRow() {
  if (this.props.onRowDeleted) {
    this.props.onRowDeleted(this.state.item);
  }
}
```

As you can see, the `CartRecord` code can execute these functions by accessing them as normal properties (albeit functions and not data). To be safe and follow a defensive pattern, we first check to ensure that the parent component has assigned a function to the property and, if so, we execute it and pass the appropriate data as parameters. While this seems simplistic, it is a powerful means of communication between child components in React and their parents. This is essentially providing child components with the means of raising events to notify parent components when key actions occur or when key data changes. In this scenario, each child component (`CartRecord`) handles the DOM click events on its own Update or Remove link. Rather than implementing the necessary code to fully handle these actions, each row simply calls the appropriate event (`updateQuantity` or `removeRow`) and passes its current product data up to the `Cart` component.

Another new concept introduced in the `render()` method of the `Cart` component is the ability to bind an element's style to an arbitrary object. The markup for the shopping cart table header can be seen here:

```
<tr>
  <th style={ this.columnStyle }>Product</th>
  <th className="text-right">Price</th>
  <th className="text-right">Quantity</th>
  <th className="text-right">Available</th>
  <th className="text-right">Total</th>
</tr>
```

In this markup, the first column header (`Product`) has a `style` property that is bound to an object called `columnStyle`. Earlier in the `Cart` component implementation you will find the implementation of a basic JavaScript object called `columnStyle`. Within this object, we can define any number of properties and values. These properties and values will be assigned as styles to the `Product` header element. The properties or the values of this object may be calculated as necessary and React will update the component's UI to reflect these changes in real-time. The implementation of `columnStyle` in our application looks like the following:

```
columnStyle = {
  width: '70%'
};
```

The final two key methods of the `Cart` component are the methods we use to handle the key actions triggered by each `CartRecord` component. These methods are as follows:

```

rowDeleted(row: ShoppingCartRecord) {

    var _service = new CartService();
    var _loggingService = new LoggingService();

    if (this.state.user != null) {

        _service.removeCartRecord(this.state.user.Id, row).then((data) => {
            this.loadCart();
        }).fail((err) => {
            if (err.responseJSON.Error) {
                alert(err.responseJSON.Error);
            }

            _loggingService.logError(err);
        });
    }
}

updateQuantity(row: ShoppingCartRecord) {
    var _service = new CartService();
    var _loggingService = new LoggingService();

    if (this.state.user != null && row.Quantity != row.OriginalQuantity) {
        _service.updateCartRecord(this.state.user.Id, row).then((data) => {
            this.loadCart();

        }).fail((err) => {
            if (err.responseJSON.Error) {
                alert(err.responseJSON.Error);
            }

            _loggingService.logError(err);
        });
    }
}

```

Both methods are very similar and both simply validate their inputs and then utilize the API wrappers to make the appropriate call to the API. Upon a successful API call, each of them triggers a new call to `loadCart()`. Calling `loadCart()` again triggers another API call to retrieve a full shopping cart from the API. This process refreshes the full `Cart` component and builds a new collection of `CartRecord` components. Upon the completion of this process, React will compare its virtual DOM with what is currently rendered to the user in the browser. Any updates will be immediately redrawn. The whole process is very efficient.

CartRecord Component

The final component within `SpyStore React` that we will look at is the `CartRecord` component. As mentioned, this small component is basically responsible for rendering a single row of the user's shopping cart. This component represents one row, as displayed in [Figure 9-15](#).

Since this component is so small we will go ahead and look at it in its entirety:

```
import * as React from "react";
import {Link} from "react-router";

export class CartRecord extends React.Component<any, any> {

  componentWillMount() {
    this.setState({quantity: -1, item: {}});
  }

  componentDidMount() {
    this.setState({quantity: this.props.item.Quantity, item: this.props.item});
  }

  quantityUpdated(event) {

    var quantity: number = Number(event.target.value);

    if (quantity > this.state.item.UnitsInStock) {
      quantity = this.state.item.UnitsInStock;
    }
    else if (quantity < 1) {
      quantity = 1;
    }
    this.state.item.Quantity = quantity;

    this.setState({quantity: quantity, item: this.state.item});
  }

  updateQuantity() {

    if (this.props.updateQuantity) {
      this.props.updateQuantity(this.state.item);
    }
  }

  removeRow() {
    if (this.props.onRowDeleted) {
      this.props.onRowDeleted(this.state.item);
    }
  }

  render() {
    return <tr>
      <td>
        <div className="product-cell-detail">
          
          <Link to={'/product/' + this.props.item.ProductId }
            className="h5">{ this.props.item.ModelName }</Link>
        </div>
      </td>
    </tr>
  }
}
```

```

        <div className="small text-muted hidden-xs">{ this.props.item.
          Description }</div>
      </div>
    </td>
    <td className="text-right">${ this.props.item.CurrentPrice.toFixed(2) }</td>
    <td className="text-right cart-quantity-row">
      <input type="number" className="cart-quantity" value={ this.state.quantity }
        onChange={ (e) => this.quantityUpdated(e) }/>
      <button className="btn btn-link btn-sm" onClick={ () => this.
        updateQuantity() }>Update</button>
      <button className="btn btn-link btn-sm" onClick={ () => this.removeRow()
        }>Remove</button>
    </td>
    <td className="text-right">{ this.props.item.UnitsInStock }</td>
    <td className="text-right">${ this.props.item.LineItemTotal.toFixed(2) }</td>
  </tr>
}
}

```

The component itself has a very basic state representation: the item (product) it represents and the current quantity of that item a user has in her cart. The user interface rendered via the `render()` method displays the record and allows a user to input a new quantity. The `onChange` event of the quantity input calls a method called `quantityUpdated()`. Within this method, we perform some basic validation to ensure the value isn't below 1 or above the available quantity for the current item.

Finally, when a user clicks on the Update or Remove buttons, we raise the appropriate event to the Cart component as described earlier.

Overall the `CartRecord` component's implementation is very light and, as we've previously stated, we have intentionally designed it to do little more than provide a visual representation of a single row. All heavy logic and functionality related to an individual row (beyond simple validation) is handled by the higher-level Cart component.

Additional Thoughts

Reviewing the provided source code for the `SpyStore`. React implementation should provide you with a good understanding of the core concepts of the React library. Its intent is to be a powerful framework for building user interfaces. Unlike other frameworks such as Angular, React maintains a narrow focus in only providing functionality specific to this goal.

If you were comparing the general architecture of the Angular solution developed in Chapter 8 with the React solution developed in this chapter, you should have noticed some major similarities. The overall component structure of both `SpyStore` implementations is very similar. The ability to utilize the TypeScript language to build a powerful client side implementation of both frontends is also common. Even the general organization of the components and other services looks similar (as was our goal).

When looking at a project and beginning to envision an architecture and frontend solution, the concepts and core structure I lean toward is very similar regardless of whether we choose to use Angular or React. The trick with either of these frameworks is to begin to envision complex user interfaces as a series of interconnected components and how they interact and share data between them. The concept of isolating all communication between the frontend interfaces and the server via a robust API layer is very common and would be done the exact same way regardless of your UI framework. We also use a very consistent development process and pipeline across our projects using tools such as NPM and WebPack. This same pipeline would be used for both an Angular or React-based application.

While we didn't discuss unit testing in this chapter, the component-based structure of a React application greatly simplifies your unit test process. The React framework itself provides several hooks to simplify testing and many React teams successfully test their React applications in test runners such as Jasmine (<https://jasmine.github.io/>) or Jest (<https://facebook.github.io/jest/>).

If your existing applications are not already written with React, your team may still consider adopting React for new features or as older features demand enough refactoring that a better UI library may suit the need. React's size and narrow focus on building interfaces makes it relatively easy to incorporate into existing web applications as needed. With React you can really pick and choose how you integrate a more modular set of UI components into an existing HTML-based structure. My team works on many legacy projects that consist of a mix of older technologies and newer, more cutting-edge technologies. We work with our customers to find creative and efficient ways to incorporate these new frameworks to take advantage of their patterns and capabilities alongside the more legacy code.

With regards to React, the community is very active and there is a vast amount of resources available. This is one area where both React and Angular stand out as extremely viable options for development teams. Both frameworks enjoy heavy adoption, which leads to readily available training materials, courses, books, and very active online communities of developers with real-world experience using these frameworks. In addition to training resources, there is also a massive collection of available React components and other packages to complement the core framework. If you find yourself in need of a nice `DatePicker` component and you are using React, there is a high likelihood that many options exist and are ready to be imported via your existing NPM/WebPack development process and used very quickly. Leveraging a development community that builds and maintains these packages in support of other React developers is another reason why the framework itself is so popular.

React also enjoys heavy support from the IDE community. Visual Studio 2017 and most other popular IDEs and source code editors natively support both the JSX and TSX file extensions. This makes developing and debugging React applications much easier than was possible just a few years ago. The overall IDE support for the React has grown as the adoption of the framework and today we get to reap the benefits through a very streamline development process.

To help you continue your journey of learning React, I will leave you with a few resources you may find useful:

- *Official React site* (<https://facebook.github.io/react/>). Here you will find the official documentation and tutorials.
- *Redux* (<http://redux.js.org/>). A framework for efficiently managing a centralized state within JavaScript applications such as those built with Angular or React.
- *React Native* (<https://facebook.github.io/react-native/>). A platform for using React to build native iOS and Android applications.

Summary

In this chapter, we have provided the basic implementation of the SpyStore ecommerce system using Facebook's React framework. In doing so, we continued to leverage the SpyStore API developed in earlier chapters.

In implementing the `SpyStore.React` solution, we started by setting up a robust development workflow utilizing Visual Studio 2017, ASP.NET Core, NPM, and WebPack. Once that environment had been configured, we copied the static assets necessary for our SpyStore interface.

To set up a clean React implementation of SpyStore, we implemented a strongly-typed set of "wrapper" classes to simplify integration between our React UI and the SpyStore API.

Finally, we individually implemented all the key React components necessary to provide users with the necessary experience in interacting with the SpyStore interface.

Index

■ A

Angular 2

- application initialization, 293–294
 - components, 294–295
 - core concepts, 293
 - event binding, 298
 - interpolation, 297
 - property binding, 297
 - routing, 300–302
 - services, 295–296
 - structural binding, 299
 - templating, 297
 - two-way data binding, 298–299
- ### Visual Studio
- Gulp Setup, 287
 - NPM Packages, 286
 - project creation, 282–283
 - project files, 284
 - Startup class, 284–285
 - TypeScript setup, 287–288

ASP.NET Core MVC Web API

- attribute routing, 86
- DI, 85
- HTTP request, 87
- and .NET Core, 84
- NuGet packages
 - data access layer, 90
 - updating and adding, 90
- project template, 88–89
- routing, 86
- runtime entity, 85
- “super” packages, 91
- URL templates, 86
- web application, 88

■ B

Bootstrapping, 354

Bower

- command, 223
- folder structure, 221

installing Git, 222

NPM, 227

- packages, 135–136
- prerequisites, 222
- search packages page, 224
- Visual Studio, 222, 224–226

Bundling and minification

- configuration, 137–138
- definition, 137
- enable bundle on build, 140
- JavaScript files, 138
- .NET Core CLI, 141–142
- produce output files, 139
- Task Runner Explorer, 141
- Visual Studio integration, 139

■ C

Classes, 246

Compiler settings, 251–252

Configure method, 97–98

ConfigureServices method, 98–100

Connection strategy, 21–22

Constructor, 96–97

Controllers

- and action, 102
- category, 111
- CategoryController class, 104
- customer, 112
- debugging profile, 102
- Formatted response, 103–104
- Get method, 105
- HTTP Get command, 105
- HTTP status code, 102–103
- launchsettings.json file, 101
- orders, 113–114, 168–169
- products, 114–115
 - constructor, 165
 - details action, 166
 - Error action, 165
 - GetListOfProducts helper method, 166

Controllers (*cont.*)

Index and Featured actions, 167

Search action, 167–168

RedirectActionResult, 104

search, 113

shopping cart, 115–118

AddToCart actions, 173

AutoMapper, 171

constructor, 170

HTTP get action, 173

HTTP post action, 174–175

Index action, 171

model binding, 171–172

route, 170

Core MVC Web applications

Bower packages, 135–136

Bundling (*see* Bundling and minification)

controller, 129–130

fake authentication, 151–152, 154

files and folders

appsettings.json, 124

Configure method, 125

ConfigureServices method, 125

controllers, 126

JavaScript, 128

lib and favicon.ico, 129

program.cs, 124

Site CSS and image files, 128

Startup.cs, 124

views, 126

wwwroot, 126–127

layouts, 133–134

NuGet Packages, 122–123

project creation, 120, 122

Razor View Engine, 133

routing

table, 123

URL templates and default values, 123

views, 130, 132–135, 154–155

WebApiCalls class (*see* WebApiCalls class)

web service locator, 142–143

CRUD operations

concurrency checking, 32

creating records, 31

deleting records, 33

EntityState, 33

no-tracking queries, 32

reading records, 31

update records, 32

■ D

Data access layer

data creation, 76–78

Down() method, 60

foreign keys, 50

initializer, 78–80

migration creation, 58

migration deploy, 58

migrations, 61

model class (*see* Model classes)

navigation properties, 50

NuGet packages, 81

ordertotal calculated field, 60

repositories

category, 67

customer, 69

ICategoryRepo interface, 64

ICustomerRepo interface, 65

InvalidQuantityException, 72

IOrderDetailRepo interface, 66

IOrderRepo interface, 65

IProductRepo interface, 65

IShoppingCartRepo interface, 66

order, 71–72

OrderDetail, 70–71

product, 67, 69

ShoppingCartRecords, 72, 74–75

Up() method, 59–60

View models (*see* View models)

Data access layer

NuGet packages, 90

DbContext class, 17–18

DbSet<T> collection type, 20

Dependency injection (DI), 85

Display attribute, 51

Down() method, 60

■ E

Entity Framework Core

Categories DbSet, 26

category model class, 24–25

connection strategy, 21–22

convention, 23

CRUD operations, 31, 33

data annotations support, 24

DbContext, 18

DbSet<T>, 20

EF 6.x, 4

entitybase class, 22, 23

framework migrations (*see* Migrations)

NuGet packages, 13

SpyStore.DAL project

adding packages, 14, 16

update and install packages, 16

SpyStore database, 4–5

StoreContext class, 19–20

stored procedures and functions, 51

testing (*see* Unit testing)

Exception filters
 action, 110
 SpyStoreExceptionHandler, 108–109

■ **F**

Fake authentication, 151–152, 154
 Foreign keys, 50

■ **G, H**

Generics, 250
 Gulp
 benefits, 227
 copying files, 229
 dependencies, 230–231
 installation, 228
 nested tasks, 231
 Task Runner Explorer, 231

■ **I**

Inheritance, 247–248
 Interfaces, 249
 Interpolation, 297

■ **J, K, L**

JavaScript application tools
 Bower, 220
 command, 223
 folder structure, 221
 installing Git, 222
 NPM, 227
 prerequisites, 222
 search packages page, 224
 Visual Studio, 222, 224–226
 Gulp, 227
 benefits, 227
 copying files, 229
 dependencies, 230–231
 installation, 228
 nested tasks, 231
 Task Runner Explorer, 231
 Node.js
 Chocolatey package
 manager, 213
 executable packages, 218–219
 locally *vs.* globally, install, 220
 manual installation, 212–213
 NPM, 215–216
 project creation, 215
 saving project dependencies, 217–218
 Visual Studio set up, 213–214

SystemJS, 233–236
 WebPack, 237–240

■ **M**

Migrations
 creating and executing, 27–28
 removal, 29
 SpyStore database, 30
 SQL scripts creation, 30
 update, 29
 Model binding, 106
 Model classes
 category model, 52
 customer model, 55
 DbSet<T>, 56
 Fluent API, 56–57
 order detail model, 54–55
 order model, 54
 product model, 52–53
 shopping cart record model, 53
 StoreContext, 56
 Model-View-Controller (MVC), 83–84
 Modules, 252–253
 MVC projects and files
 appsettings.json file(s), 93–94
 configure method, 97–98
 ConfigureServices method, 98–100
 constructor, 96–97
 controllers, 100–102
 program.cs File, 92–93
 runtimeconfig.template.json, 94
 Startup.cs, 94
 Startup services, 95–96

■ **N**

Navigation properties, 50
 .NET Core Command Line Interface (CLI), 8
 NuGet packages
 Core MVC Web applications, 122–123
 packages restoring
 CLI, 13
 manually, 13
 PMC, 13
 project reference, 14

■ **O**

Object Relational Mapping (ORM), 3

■ **P, Q**

Package Manager Console (PMC), 13

R

Razor View Engine, 133

React

- app component, 365–367, 369
- application organization, 355
- bootstrapping, 354
- cart component, 380–383, 384
- CartRecord component, 384–386
- CategoryLinks component, 369–371
- component, 351–354
- Initial components, 361–362
- models, 356–357
- NPM packages, 334–337
- ProductDetail component, 376–380
- products component, 371, 375–376
 - built-in methods, 373
 - interface, 372
 - loadFeaturedProducts(), 374
 - ProductService, 372
 - refreshProducts() method, 373
 - render(), 374–375
- project folder, 338–341
- routing, 362–365
- services, 357, 359–361
- setState() method, 367
- solution, 329
- Startup class, 333–334
- TypeScript setup, 337–338
- Visual Studio
 - project creation, 330–332
 - project files, 332–333
- WebPack, 343–344
 - configuration, 342–343
 - loader, 346
 - NPM, 342
 - paths, 345
 - plugins section, 346–349
 - production, 343–344
 - resolve section, 345
 - TypeScript code, 349
 - wwwroot folder, 347

Repository interface

- base repository, 43–46
- category repository, 46–47
- creation, 41–42

Routing, 300–302

Running application, 107

S

Solution and projects

- adding project, 10
- adding SpyStore, 11
- creation, 8–9

file globbing, 11

target framework, update, 12

SpyStore Angular 2

add routing

- app component, 306
- and components to app
 - module, 305–306
- creating route, 304–305
- product component, 303–304

addToCart method, 318–319

angular bootstrap, 292

app module, 292

app root components, 321

cart page

- app module, 325
- cart record component, 323–324
- cart route, 325
- components, 321, 323

checkout process, 326–327

connecting to service, 306–310

product detail route, 320

product details component, 316–318

product details page, 316

root app component, 291

root page, 289–291

route parameters, 311–312

search page, 313–315

SpyStore database, 4–5, 49

SpyStore MVC application

controller

- orders, 168–169
- products, 165–168
- shopping cart, 170–175
- running application, 205–206
- validation (*see* Validation)

View Components (*see* View Components)

SpyStore.Reactsolution. *See* React

StoreContext class, 19–20

Stored procedures and functions, 51

“Super” packages, 91

SystemJS, 233–236

T

Tag Helpers, 119

anchor, 160

built-in tag, 158–159

custom, 163–164

environment, 163

form, 160

HTML Helper, 157

image, 163

input, 160–161

link and script, 163

select, 161–162

- TextArea, 161
 - validation, 162
- Transpiler, 241
- TypeScript
 - adding files, 261–262
 - classes, 246, 263
 - compiler settings, 251–252
 - currentPrice method, 279
 - CustomerAnonymous.ts, 264–265
 - datatypes, 242–243, 245
 - debugging, 272, 274–275
 - ECMAScript features, 242
 - generics, 250
 - inheritance, 247–248
 - interfaces, 249
 - loadProducts method, 278
 - modules, 252–253
 - NPM packages, 259, 261
 - product interface, 276–277
 - product list, 266–272, 279
 - project creation
 - adding HTML page, 257
 - compiler options, 259
 - NuGet package manager, 256
 - template selection, 255
 - tsconfig.json file, 258
 - SPAs, 242
 - transpiler, 241

U

- Unit testing, 118
 - category record
 - adding, 37–38
 - delete, 40
 - retrieve, 38
 - update, 39
 - CategoryTests class, 34–36
 - concurrency checking, 41
 - creating and running, 36
 - CRUD operations, 37
- Up() method, 59–60

V

- Validation
 - client side, 180, 182–183
 - server side, 176–179
- View Components, 119
 - Add to Cart, 192–193
 - Cart, 195
 - client side code, 186
 - custom tag helpers, 186

- Details view, 201–203
 - display template, 193, 204–205
 - editor template, 193, 195, 198–199
 - Index view, 195–197, 200–201
 - layout view, 188–190
 - Login Partial View, 187–188
 - orders, 200
 - products view, 203–204
 - server side code, 183–185
 - shared folder, 187
 - Update partial view, 197–198
 - validation scripts, 191
- View models
 - cart record, 64
 - order detail, 63
 - Product and Category, 62
- Views
 - CartViewModelBase, 155
 - folder, 130–132
 - partial, 134
 - ProductAndCategoryBase, 154
 - Razor View Engine, 133
 - sending data, 134–135
 - strongly typed, 135
- Visual Studio
 - CLI, 8
 - installation, 5–6
 - launch settings, 106–107
 - .NET Core SDKs, 6–7

W, X, Y, Z

- WebApiCalls class
 - Base class, 144
 - Base HTTP delete calls, 147–148
 - Base HTTP Get calls, 145–146
 - Base HTTP Post and Put calls, 146–147
 - class creation, 148–150
 - DI container, 151
 - IWebApiCalls interface, 143
- Webpack, 237–240, 343–344
 - app component, 368
 - configuration, 342–343
 - loader, 346
 - NPM, 342
 - package.json file, 351
 - paths, 345
 - plugins section, 346–349
 - production, 343–344
 - resolve section, 345
 - TypeScript code, 349–350
 - wwwroot folder, 347