



Code for
new
releases
available
online

Pro Angular

Build Powerful and Dynamic Web Apps

—
Fifth Edition
—

Adam Freeman

Apress®

Pro Angular

Build Powerful and Dynamic Web Apps

Fifth Edition



Adam Freeman

Apress®

Pro Angular: Build Powerful and Dynamic Web Apps

Adam Freeman
London, UK

ISBN-13 (pbk): 978-1-4842-8175-8

<https://doi.org/10.1007/978-1-4842-8176-5>

ISBN-13 (electronic): 978-1-4842-8176-5

Copyright © 2022 by Adam Freeman

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

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

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

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

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

Cover designed by eStudioCalamar

Cover image designed by Shutterstock (www.shutterstock.com)

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

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

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

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub ([Github.com/apress](https://github.com/apress)). For more detailed information, please visit www.apress.com/source-code.

Printed on acid-free paper

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

Table of Contents

About the Author	xxiii
About the Technical Reviewer	xxv
■ Part I: Getting Ready.....	1
■ Chapter 1: Getting Ready.....	3
Understanding Where Angular Excels	4
Understanding Round-Trip and Single-Page Applications	4
Comparing Angular to React and Vue.js	5
What Do You Need to Know?	5
What Is the Structure of This Book?.....	5
Part 1: Getting Started with Angular	5
Part 2: Angular in Detail.....	5
Part 3: Advanced Angular Features.....	6
What Doesn't This Book Cover?	6
What Software Do I Need for Angular Development?.....	6
How Do I Set Up the Development Environment?	6
What If I Have Problems Following the Examples?	6
What If I Find an Error in the Book?	7
Are There Lots of Examples?.....	7
Where Can You Get the Example Code?	9
How Do I Contact the Author?	9

■ TABLE OF CONTENTS

What If I Really Enjoyed This Book?	9
What If This Book Has Made Me Angry and I Want to Complain?	10
Summary	10
■ Chapter 2: Jumping Right In.....	11
Getting Ready	11
Installing Node.js	11
Installing an Editor.....	13
Installing the Angular Development Package	13
Choosing a Browser.....	13
Creating an Angular Project.....	14
Opening the Project for Editing.....	14
Starting the Angular Development Tools.....	15
Adding Features to the Application	17
Creating a Data Model	17
Displaying Data to the User	19
Updating the Component.....	20
Styling the Application Content	24
Applying Angular Material Components	25
Defining the Spacer CSS Style.....	27
Displaying the List of To-Do Items.....	28
Defining Additional Styles.....	30
Creating a Two-Way Data Binding	31
Filtering Completed To-Do Items	34
Adding To-Do Items	35
Finishing Up	38
Summary	41
■ Chapter 3: Primer, Part 1.....	43
Preparing the Example Project.....	43
Understanding HTML.....	45
Understanding Void Elements.....	47
Understanding Attributes.....	47

Applying Attributes Without Values	47
Quoting Literal Values in Attributes	48
Understanding Element Content.....	48
Understanding the Document Structure	48
Understanding CSS and the Bootstrap Framework.....	50
Understanding TypeScript/JavaScript	51
Understanding the TypeScript Workflow.....	51
Understanding JavaScript vs. TypeScript	52
Understanding the Basic TypeScript/JavaScript Features.....	60
Defining Variables and Constants.....	60
Dealing with Unassigned and Null Values	60
Using the JavaScript Primitive Types.....	62
Using the JavaScript Operators.....	64
Summary.....	72
■ Chapter 4: Primer, Part 2	73
Preparing for This Chapter	73
Defining and Using Functions.....	74
Defining Optional Function Parameters.....	75
Defining Default Parameter Values.....	76
Defining Rest Parameters.....	76
Defining Functions That Return Results	77
Using Functions as Arguments to Other Functions.....	77
Working with Arrays	79
Reading and Modifying the Contents of an Array	80
Enumerating the Contents of an Array.....	81
Using the Spread Operator	82
Using the Built-in Array Methods.....	83
Working with Objects	84
Understanding Literal Object Types	85
Defining Classes	87
Checking Object Types.....	91

■ TABLE OF CONTENTS

Working with JavaScript Modules.....	92
Creating and Using Modules.....	92
Working with Reactive Extensions	94
Understanding Observables.....	95
Understanding Observers	96
Understanding Subjects	96
Summary.....	98
■ Chapter 5: SportsStore: A Real Application.....	99
 Preparing the Project	99
Installing the Additional NPM Packages	100
Preparing the RESTful Web Service.....	101
Preparing the HTML File	103
Creating the Folder Structure	104
Running the Example Application	104
Starting the RESTful Web Service.....	105
 Preparing the Angular Project Features	105
Updating the Root Component.....	106
Inspecting the Root Module.....	106
Inspecting the Bootstrap File.....	107
 Starting the Data Model	108
Creating the Model Classes.....	108
Creating the Dummy Data Source	109
Creating the Model Repository	110
Creating the Feature Module	111
 Starting the Store	111
Creating the Store Component and Template	112
Creating the Store Feature Module.....	113
Updating the Root Component and Root Module.....	114
 Adding Store Features the Product Details	115
Displaying the Product Details.....	115
Adding Category Selection	117

Adding Product Pagination	119
Creating a Custom Directive	122
Summary.....	126
■ Chapter 6: SportsStore: Orders and Checkout.....	127
Preparing the Example Application	127
Creating the Cart	127
Creating the Cart Model.....	128
Creating the Cart Summary Components.....	130
Integrating the Cart into the Store.....	131
Adding URL Routing.....	134
Creating the Cart Detail and Checkout Components	135
Creating and Applying the Routing Configuration.....	136
Navigating Through the Application.....	137
Guarding the Routes	140
Completing the Cart Detail Feature	142
Processing Orders	145
Extending the Model.....	145
Collecting the Order Details.....	148
Using the RESTful Web Service	152
Applying the Data Source	153
Summary.....	154
■ Chapter 7: SportsStore: Administration	155
Preparing the Example Application	155
Creating the Module	155
Configuring the URL Routing System.....	158
Navigating to the Administration URL.....	159
Implementing Authentication	161
Understanding the Authentication System	161
Extending the Data Source	162
Creating the Authentication Service	163
Enabling Authentication.....	164

■ TABLE OF CONTENTS

Extending the Data Source and Repositories	167
Installing the Component Library	170
Creating the Administration Feature Structure.....	172
Creating the Placeholder Components	173
Preparing the Common Content and the Feature Module	174
Implementing the Product Table Feature.....	178
Implementing the Product Editor.....	185
Implementing the Order Table Feature	190
Summary.....	194
■ Chapter 8: SportsStore: Progressive Features and Deployment.....	195
Preparing the Example Application	195
Adding Progressive Features	195
Installing the PWA Package	195
Caching the Data URLs	196
Responding to Connectivity Changes	197
Preparing the Application for Deployment.....	199
Creating the Data File	199
Creating the Server.....	200
Changing the Web Service URL in the Repository Class.....	202
Building and Testing the Application	203
Testing the Progressive Features	204
Containerizing the SportsStore Application.....	206
Installing Docker.....	206
Preparing the Application	206
Creating the Docker Container	207
Running the Application.....	208
Summary.....	209

■ Part II: Working with Angular	211
■ Chapter 9: Understanding Angular Projects and Tools	213
Creating a New Angular Project	213
Understanding the Project Structure	214
Understanding the Source Code Folder	217
Understanding the Packages Folder	218
Using the Development Tools	223
Understanding the Development HTTP Server	223
Understanding the Build Process	224
Using the Linter	230
Understanding How an Angular Application Works	235
Understanding the HTML Document	236
Understanding the Application Bootstrap	237
Understanding the Root Angular Module	238
Understanding the Angular Component	239
Understanding Content Display	239
Understanding the Production Build Process	241
Running the Production Build	242
Starting Development in an Angular Project	242
Creating the Data Model	242
Creating a Component and Template	245
Configuring the Root Angular Module	247
Summary	248
■ Chapter 10: Using Data Bindings	249
Preparing for This Chapter	250
Understanding One-Way Data Bindings	251
Understanding the Binding Target	253
Understanding the Expression	254
Understanding the Brackets	255
Understanding the Host Element	256

■ TABLE OF CONTENTS

Using the Standard Property and Attribute Bindings.....	256
Using the Standard Property Binding	257
Using the String Interpolation Binding.....	258
Using the Attribute Binding.....	259
Setting Classes and Styles.....	261
Using the Class Bindings	261
Using the Style Bindings.....	266
Updating the Data in the Application.....	270
Summary.....	273
■ Chapter 11: Using the Built-in Directives	275
Preparing the Example Project.....	276
Using the Built-in Directives.....	278
Using the nglf Directive	279
Using the ngSwitch Directive	281
Using the ngFor Directive	284
Using the ngTemplateOutlet Directive	294
Using Directives Without an HTML Element.....	296
Understanding One-Way Data Binding Restrictions	297
Using Idempotent Expressions	297
Understanding the Expression Context.....	300
Summary.....	302
■ Chapter 12: Using Events and Forms.....	303
Preparing the Example Project.....	304
Importing the Forms Module	304
Preparing the Component and Template	305
Using the Event Binding	306
Using Event Data	310
Handling Events in the Component.....	312
Using Template Reference Variables	314
Using Two-Way Data Bindings.....	315
Using the ngModel Directive.....	317

Working with Forms	319
Adding a Form to the Example Application.....	319
Adding Form Data Validation	322
Validating the Entire Form	331
Completing the Form.....	337
Summary.....	339
■ Chapter 13: Creating Attribute Directives	341
Preparing the Example Project.....	342
Creating a Simple Attribute Directive	344
Applying a Custom Directive.....	345
Accessing Application Data in a Directive	347
Reading Host Element Attributes.....	347
Creating Data-Bound Input Properties.....	350
Responding to Input Property Changes	353
Creating Custom Events	355
Binding to a Custom Event	357
Creating Host Element Bindings.....	358
Creating a Two-Way Binding on the Host Element	360
Exporting a Directive for Use in a Template Variable.....	364
Summary.....	366
■ Chapter 14: Creating Structural Directives.....	367
Preparing the Example Project.....	368
Creating a Simple Structural Directive	369
Implementing the Structural Directive Class	371
Enabling the Structural Directive.....	373
Using the Concise Structural Directive Syntax	375
Creating Iterating Structural Directives.....	376
Providing Additional Context Data	379
Using the Concise Structure Syntax	382
Dealing with Property-Level Data Changes	383
Dealing with Collection-Level Data Changes.....	384

■ TABLE OF CONTENTS

Querying the Host Element Content	395
Querying Multiple Content Children.....	399
Receiving Query Change Notifications	401
Summary.....	403
■ Chapter 15: Understanding Components.....	405
Preparing the Example Project.....	406
Structuring an Application with Components.....	407
Creating New Components.....	408
Defining Templates	413
Completing the Component Restructure	424
Using Component Styles	425
Defining External Component Styles	427
Using Advanced Style Features	428
Querying Template Content	436
Summary.....	438
■ Chapter 16: Using and Creating Pipes	439
Preparing the Example Project.....	440
Understanding Pipes	444
Creating a Custom Pipe.....	445
Registering a Custom Pipe	446
Applying a Custom Pipe.....	447
Combining Pipes.....	449
Creating Impure Pipes	449
Using the Built-in Pipes.....	454
Formatting Numbers.....	454
Formatting Currency Values	458
Formatting Percentages	461
Formatting Dates	463
Changing String Case	469
Serializing Data as JSON	471
Slicing Data Arrays	472

Formatting Key-Value Pairs	474
Selecting Values	475
Pluralizing Values	477
Using the Async Pipe	479
Summary.....	482
■ Chapter 17: Using Services	483
Preparing the Example Project.....	484
Understanding the Object Distribution Problem	485
Demonstrating the Problem.....	485
Distributing Objects as Services Using Dependency Injection	491
Declaring Dependencies in Other Building Blocks	497
Understanding the Test Isolation Problem.....	504
Isolating Components Using Services and Dependency Injection.....	505
Completing the Adoption of Services	508
Updating the Root Component and Template	509
Updating the Child Components	509
Summary.....	511
■ Chapter 18: Using Service Providers	513
Preparing the Example Project.....	514
Using Service Providers	516
Using the Class Provider.....	519
Using the Value Provider.....	528
Using the Factory Provider	530
Using the Existing Service Provider.....	533
Using Local Providers.....	534
Understanding the Limitations of Single Service Objects.....	534
Creating Local Providers in a Component.....	536
Understanding the Provider Alternatives	538
Controlling Dependency Resolution.....	543
Summary.....	545

■ TABLE OF CONTENTS

■ Chapter 19: Using and Creating Modules	547
Preparing the Example Project.....	548
Understanding the Root Module.....	550
Understanding the imports Property	552
Understanding the declarations Property	552
Understanding the providers Property.....	553
Understanding the bootstrap Property	553
Creating Feature Modules	555
Creating a Model Module.....	557
Creating a Utility Feature Module	563
Creating a Feature Module with Components	570
Summary.....	575
■ Part III: Advanced Angular Features.....	577
■ Chapter 20: Creating the Example Project.....	579
Starting the Example Project.....	579
Adding and Configuring the Bootstrap CSS Package	579
Creating the Project Structure	580
Creating the Model Module	580
Creating the Product Data Type	580
Creating the Data Source and Repository.....	581
Completing the Model Module.....	583
Creating the Messages Module	583
Creating the Message Model and Service	583
Creating the Component and Template	584
Completing the Message Module	585
Creating the Core Module.....	585
Creating the Shared State Service	585
Creating the Table Component.....	586

Creating the Form Component.....	588
Completing the Core Module	590
Completing the Project.....	591
Summary.....	592
■ Chapter 21: Using the Forms API, Part 1	593
Preparing for This Chapter	594
Understanding the Reactive Forms API	595
Rebuilding the Form Using the API.....	596
Responding to Form Control Changes	599
Managing Control State.....	602
Managing Control Validation.....	605
Adding Additional Controls	610
Working with Multiple Form Controls.....	612
Using a Form Group with a Form Element.....	616
Accessing the Form Group from the Template	618
Displaying Validation Messages with a Form Group.....	622
Nesting Form Controls.....	625
Summary.....	631
■ Chapter 22: Using the Forms API, Part 2	633
Preparing for This Chapter	633
Creating Form Components Dynamically	634
Using a Form Array	635
Adding and Removing Form Controls	641
Validating Dynamically Created Form Controls	643
Filtering the FormArray Values	644
Creating Custom Form Validation	648
Creating a Directive for a Custom Validator.....	650
Validating Across Multiple Fields.....	654
Performing Validation Asynchronously	660
Summary.....	664

■ Chapter 23: Making HTTP Requests	665
Preparing the Example Project.....	666
Configuring the Model Feature Module	667
Creating the Data File	667
Running the Example Project	668
Understanding RESTful Web Services	669
Replacing the Static Data Source.....	670
Creating the New Data Source Service	670
Configuring the Data Source.....	673
Using the REST Data Source.....	673
Saving and Deleting Data	675
Consolidating HTTP Requests	678
Making Cross-Origin Requests.....	680
Using JSONP Requests	681
Configuring Request Headers.....	683
Handling Errors	685
Generating User-Ready Messages.....	686
Handling the Errors.....	688
Summary.....	690
■ Chapter 24: Routing and Navigation: Part 1	691
Preparing the Example Project.....	692
Getting Started with Routing	693
Creating a Routing Configuration.....	694
Creating the Routing Component.....	696
Updating the Root Module	696
Completing the Configuration	696
Adding Navigation Links	698
Understanding the Effect of Routing.....	700
Completing the Routing Implementation.....	702
Handling Route Changes in Components	702
Using Route Parameters	705

Navigating in Code.....	713
Receiving Navigation Events	716
Removing the Event Bindings and Supporting Code	719
Summary.....	721
■ Chapter 25: Routing and Navigation: Part 2	723
Preparing the Example Project.....	723
Adding Components to the Project.....	728
Using Wildcards and Redirections.....	731
Using Wildcards in Routes	731
Using Redirections in Routes.....	734
Navigating Within a Component.....	735
Responding to Ongoing Routing Changes	736
Styling Links for Active Routes	738
Fixing the All Button.....	741
Creating Child Routes.....	743
Creating the Child Route Outlet	744
Accessing Parameters from Child Routes	746
Summary.....	750
■ Chapter 26: Routing and Navigation: Part 3	751
Preparing the Example Project.....	751
Guarding Routes.....	752
Delaying Navigation with a Resolver	753
Preventing Navigation with Guards	760
Loading Feature Modules Dynamically	773
Creating a Simple Feature Module	773
Loading the Module Dynamically.....	774
Guarding Dynamic Modules.....	777
Targeting Named Outlets.....	780
Creating Additional Outlet Elements	781
Navigating When Using Multiple Outlets.....	783
Summary.....	785

■ TABLE OF CONTENTS

■ Chapter 27: Using Animations	787
Preparing the Example Project.....	788
Disabling the HTTP Delay.....	788
Simplifying the Table Template and Routing Configuration	789
Getting Started with Angular Animation	791
Enabling the Animation Module.....	792
Creating the Animation	792
Applying the Animation.....	796
Testing the Animation Effect.....	799
Understanding the Built-in Animation States	801
Understanding Element Transitions.....	802
Creating Transitions for the Built-in States.....	802
Controlling Transition Animations	804
Understanding Animation Style Groups.....	809
Defining Common Styles in Reusable Groups	810
Using Element Transformations.....	811
Applying CSS Framework Styles	813
Summary.....	817
■ Chapter 28: Working with Component Libraries.....	819
Preparing for This Chapter	820
Installing the Component Library.....	822
Adjusting the HTML File.....	823
Running the Project	824
Using the Library Components	825
Using the Angular Button Directive.....	825
Using the Angular Material Table	829
Matching the Component Library Theme	839
Creating the Custom Component.....	839
Using the Angular Material Theme	841
Applying the Ripple Effect	846
Summary.....	848

■ Chapter 29: Angular Unit Testing.....	849
Preparing the Example Project.....	850
Running a Simple Unit Test	852
Working with Jasmine.....	854
Testing an Angular Component	855
Working with the TestBed Class	856
Testing Data Bindings.....	860
Testing a Component with an External Template.....	862
Testing Component Events	864
Testing Output Properties	867
Testing Input Properties.....	869
Testing an Angular Directive.....	871
Summary.....	873
■ Index.....	875

About the Author



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

About the Technical Reviewer

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

CHAPTER 1



Getting Ready

Angular taps into some of the best aspects of server-side development and uses them to enhance HTML in the browser, creating a foundation that makes building rich applications simpler and easier. Angular applications are built around a clear design pattern that emphasizes creating applications that are

- *Extendable*: It is easy to figure out how even a complex Angular app works once you understand the basics—and that means you can easily enhance applications to create new and useful features for your users.
- *Maintainable*: Angular apps are easy to debug and fix, which means that long-term maintenance is simplified.
- *Testable*: Angular has good support for unit and end-to-end testing, meaning you can find and fix defects before your users do.
- *Standardized*: Angular builds on the innate capabilities of the web browser without getting in your way, allowing you to create standards-compliant web apps that take advantage of the latest HTML and features, as well as popular tools and frameworks.

Angular is an open-source JavaScript library that is sponsored and maintained by Google. It has been used in some of the largest and most complex web apps around. In this book, I show you everything you need to know to get the benefits of Angular in your projects.

THIS BOOK AND THE ANGULAR RELEASE SCHEDULE

Google has adopted an aggressive release schedule for Angular. This means there is an ongoing stream of minor releases and a major release every six months. Minor releases should not break any existing features and should largely contain bug fixes. The major releases can contain substantial changes and may not offer backward compatibility.

It doesn't seem fair or reasonable to ask readers to buy a new edition of this book every six months, especially since most Angular features are unlikely to change even in a major release. Instead, I am going to post updates following the major releases to the GitHub repository for this book, <https://github.com/Apress/pro-angular-5ed>.

This is an ongoing experiment for me (and for Apress), but the goal is to extend the life of this book by supplementing the examples it contains.

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

Understanding Where Angular Excels

Angular isn't the solution to every problem, and it is important to know when you should use Angular and when you should seek an alternative. Angular delivers the kind of functionality that used to be available only to server-side developers, but delivers it entirely in the browser. This means Angular has a lot of work to do each time an HTML document to which Angular has been applied is loaded—the HTML elements have to be compiled, the data bindings have to be evaluated, components and other building blocks need to be executed, and so on.

This kind of work takes time to perform, and the amount of time depends on the complexity of the HTML document, on the associated JavaScript code, and—critically—on the quality of the browser and the processing capability of the device. You won't notice any delay when using the latest browsers on a capable desktop machine, but old browsers on underpowered smartphones can slow down the initial setup of an Angular app.

The goal is to perform this setup as infrequently as possible and deliver as much of the app as possible to the user when it is performed. This means giving careful thought to the kind of web application you build. In broad terms, there are two kinds of web applications: *round-trip* and *single-page*.

Understanding Round-Trip and Single-Page Applications

For a long time, web apps were developed to follow a *round-trip* model. The browser requests an initial HTML document from the server. User interactions—such as clicking a link or submitting a form—led the browser to request and receive a completely new HTML document. In this kind of application, the browser is essentially a rendering engine for HTML content, and all of the application logic and data resides on the server. The browser makes a series of stateless HTTP requests that the server handles by generating HTML documents dynamically.

Some current web development is still for round-trip applications, not least because they require little from the browser, which ensures the widest possible client support. But there are some drawbacks to round-trip applications: they make the user wait while the next HTML document is requested and loaded, they require a large server-side infrastructure to process all the requests and manage all the application state, and they require more bandwidth because each HTML document has to be self-contained (leading to a lot of the same content being included in each response from the server).

Single-page applications take a different approach. An initial HTML document is sent to the browser, along with JavaScript code, but user interactions lead to Ajax requests for small fragments of HTML or data inserted into the existing set of elements being displayed to the user. The initial HTML document is never reloaded or replaced, and the user can continue to interact with the existing HTML while the Ajax requests are being performed asynchronously, even if that just means seeing a “data loading” message. The single-page application model is perfect for Angular.

Tip Another phrase you may encounter is *progressive web applications* (PWAs). Progressive applications continue to work even when disconnected from the network and have access to features such as push notifications. PWAs are not specific to Angular, but I demonstrate how to use simple PWA features in Chapter 8.

Comparing Angular to React and Vue.js

There are two main competitors to Angular: React and Vue.js. There are some low-level differences between them, but, for the most part, all of these frameworks are excellent, all of them work in similar ways, and all of them can be used to create rich and fluid client-side applications.

The main difference between these frameworks is the developer experience. Angular requires you to use TypeScript to be effective, for example. If you are used to using a language like C# or Java, then TypeScript will be familiar, and it addresses some of the oddities of the JavaScript language. Vue.js and React don't require TypeScript (although it is supported by both frameworks) but lean toward mixing HTML, JavaScript, and CSS content together in a single file, which not everyone likes.

My advice is simple: pick the framework that you like the look of the most and switch to one of the others if you don't get on with it. That may seem like an unscientific approach, but there isn't a bad choice to make, and you will find that many of the core concepts carry over between frameworks even if you switch.

What Do You Need to Know?

Before reading this book, you should be familiar with the basics of web development, have an understanding of how HTML and CSS work, and have a working knowledge of JavaScript. If you are a little hazy on some of these details, I provide primers for the HTML and TypeScript/JavaScript I use in this book in Chapters 3 and 4. You won't find a comprehensive reference for HTML elements and CSS properties, though, because there just isn't the space in a book about Angular to cover all of HTML.

What Is the Structure of This Book?

This book is split into three parts, each of which covers a set of related topics.

Part 1: Getting Started with Angular

Part 1 of this book provides the information you need to get ready for the rest of the book. It includes this chapter and primers/refreshers for key technologies, including HTML and TypeScript, which is a superset of JavaScript used in Angular development. I also show you how to build your first Angular application and take you through the process of building a more realistic application, called SportsStore.

Part 2: Angular in Detail

Part 2 of this book takes you through the building blocks provided by Angular for creating applications, working through each of them in turn. Angular includes a lot of built-in functionality, which I describe in-depth, and provides endless customization options, all of which I demonstrate.

Part 3: Advanced Angular Features

Part 3 of this book explains how advanced features can be used to create more complex and scalable applications. I demonstrate how to make asynchronous HTTP requests in an Angular application, how to use URL routing to navigate around an application, and how to animate HTML elements when the state of the application changes.

What Doesn't This Book Cover?

This book is for experienced web developers who are new to Angular. It doesn't explain the basics of web applications or programming, although there are primer chapters on HTML and TypeScript/JavaScript. I don't describe server-side development in any detail—see my other books if you want to create the back-end services required to support Angular applications.

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

What Software Do I Need for Angular Development?

You will need a code editor and the tools described in Chapter 2. Everything required for Angular development is available without charge and can be used on Windows, macOS, and Linux.

How Do I Set Up the Development Environment?

Chapter 2 introduces Angular by creating a simple application, and, as part of that process, I tell you how to create a development environment for working with Angular.

What If I Have Problems Following the Examples?

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

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

If you still have problems, then download the project for the chapter you are reading from the book's GitHub repository, <https://github.com/Apress/pro-angular-5ed>, and compare it to your project. I created the code for the GitHub repository by working through each chapter, so you should have the same files with the same contents in your project.

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

What If I Find an Error in the Book?

You can report errors to me by email at adam@adam-freeman.com, although I ask that you first check the errata/corrections list for this book, which you can find in the book's GitHub repository at <https://github.com/Apress/pro-angular-5ed>, in case it has already been reported.

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

ERRATA BOUNTY

Apress has agreed to give a free ebook to readers who are the first to report errors that make it onto the GitHub errata list for this book. Readers can select any Apress ebook available through Springerlink.com, not just my books.

This is an entirely discretionary and experimental program. Discretionary means that only I decide which errors are listed in the errata and which reader is the first to make a report. Experimental means Apress may decide not to give away any more books at any time for any reason. There are no appeals, and this is not a promise or a contract or any kind of formal offer or competition. Put another way, this is a nice and informal way to say thank-you and to encourage readers to report mistakes that I have missed when writing this book.

Are There Lots of Examples?

There are *loads* of examples. The best way to learn Angular is by example, and I have packed as many of them as I can into this book. To maximize the number of examples in this book, I have adopted a simple convention to avoid listing the contents of files over and over. The first time I use a file in a chapter, I'll list the complete contents, just as I have in Listing 1-1. I include the name of the file in the listing's header and the folder in which you should create it. When I make changes to the code, I show the altered statements in bold.

Listing 1-1. A Complete Example Document

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';

@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

This listing is taken from Chapter 5. Don't worry about what it does; just be aware that this is a complete listing, which shows the entire contents of the file.

When I make a series of changes to the same file or when I make a small change to a large file, I show you just the elements that change, to create a *partial listing*. You can spot a partial listing because it starts and ends with an ellipsis (...), as shown in Listing 1-2.

Listing 1-2. A Partial Listing

```
...
class PaIteratorContext {
  odd: boolean; even: boolean;
  first: boolean; last: boolean;

  constructor(public $implicit: any,
    public index: number, total: number ) {

    this.odd = index % 2 == 1;
    this.even = !this.odd;
    this.first = index == 0;
    this.last = index == total - 1;

    setInterval(() => {
      this.odd = !this.odd; this.even = !this.even;
      this.$implicit.price++;
    }, 2000);
  }
}

...
...
```

Listing 1-2 is from a later chapter. You can see that just a section of the file is shown and that I have highlighted several statements. This is how I draw your attention to the part of the listing that has changed or emphasize the part of an example that shows the feature or technique I am describing. In some cases, I need to make changes to different parts of the same file, in which case I omit some elements or statements for brevity, as shown in Listing 1-3.

Listing 1-3. Omitting Statements for Brevity

```
import { Component, Input } from "@angular/core";
import { NgForm, FormControl, Validators } from "@angular/forms";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model"
import { Message } from "../messages/message.model"
import { MessageService } from "../messages/message.service";
import { MODES, SharedState, StateUpdate } from "./sharedState.service";

@Component({
  selector: "paForm",
  templateUrl: "form.component.html",
  styleUrls: ["form.component.css"]
})
export class FormComponent {
  product: Product = new Product();
  editing: boolean = false;
```

```
nameField: FormControl = new FormControl("", {
  validators: [
    Validators.required,
    Validators.minLength(3),
    Validators.pattern("^[A-Za-z ]+$")
  ],
  updateOn: "change"
});  
  
// ...constructor and methods omitted for brevity...
}
```

This convention lets me pack in more examples, but it does mean it can be hard to locate a specific technique. To this end, the chapters in which I describe Angular features in Parts 2 and 3 begin with a summary table that describes the techniques contained in the chapter and the listings that demonstrate how they are used.

Where Can You Get the Example Code?

You can download the example projects for all the chapters in this book from <https://github.com/Apress/pro-angular-5ed>.

How Do I Contact the Author?

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

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

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

What If I Really Enjoyed This Book?

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

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

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

Summary

In this chapter, I outlined the content and structure of this book. The best way to learn Angular development is by example, so in the next chapter, I jump right in and show you how to set up your development environment and use it to create your first Angular application.

CHAPTER 2



Jumping Right In

The best way to get started with Angular is to dive in and create a web application. In this chapter, I show you how to set up your development environment and take you through the process of creating a basic application. In Chapters 5–8, I show you how to create a more complex and realistic Angular application, but for now, a simple example will suffice to demonstrate the major components of an Angular app and set the scene for the other chapters in this part of the book.

Don't worry if you don't follow everything that happens in this chapter. Angular has a steep learning curve, so the purpose of this chapter is just to introduce the basic flow of Angular development and give you a sense of how things fit together. It won't all make sense right now, but by the time you have finished reading this book, you will understand every step I take in this chapter and much more besides.

Getting Ready

There is some preparation required for Angular development. In the sections that follow, I explain how to get set up and ready to create your first project. There is wide support for Angular in popular development tools, and you can pick your favorites.

Installing Node.js

Node.js is a JavaScript runtime for server-side applications and is used by most web application frameworks, including Angular.

The version of Node.js I have used in this book is 16.13.0, which is the current Long-Term Support (LTS) release at the time of writing. There may be a later version available by the time you read this, but you should stick to the 16.13.0 release for the examples in this book. A complete set of 16.13.0 releases, with installers for Windows and macOS and binary packages for other platforms, is available at <https://nodejs.org/dist/v16.13.0>.

Download and run the installer and ensure that the “npm package manager” option and the two Add to PATH options are selected, as shown in Figure 2-1.

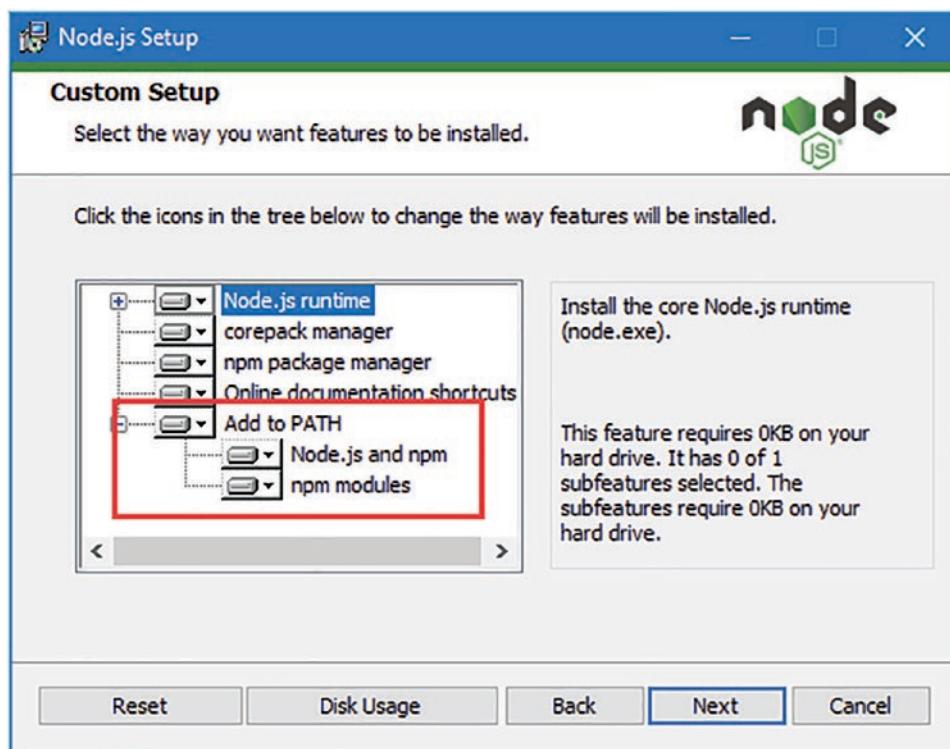


Figure 2-1. Installing Node.js

When the installation is complete, open a new command prompt and run the command shown in Listing 2-1.

Listing 2-1. Running Node.js

```
node -v
```

If the installation has gone as it should, then you will see the following version number displayed:

```
v16.13.0
```

The Node.js installer includes the Node Package Manager (NPM), which is used to manage the packages in a project. Run the command shown in Listing 2-2 to ensure that NPM is working.

Listing 2-2. Running NPM

```
npm -v
```

If everything is working as it should, then you will see the following version number:

8.1.0

Installing an Editor

Angular development can be done with any programmer's editor, from which there is an endless number to choose. Some editors have enhanced support for working with Angular, including highlighting key terms and good tool integration.

When choosing an editor, one of the most important considerations is the ability to filter the content of the project so that you can focus on a subset of the files. There can be a lot of files in an Angular project, and many have similar names, so being able to find and edit the right file is essential. Editors make this possible in different ways, either by presenting a list of the files that are open for editing or by providing the ability to exclude files with specific extensions.

The examples in this book do not rely on any specific editor, and all the tools I use are run from the command line. If you don't already have a preferred editor for web application development, then I recommend using Visual Studio Code, which is provided without charge by Microsoft and has excellent support for Angular development. You can download Visual Studio Code from <https://code.visualstudio.com>.

Installing the Angular Development Package

The Angular team provides a complete set of command-line tools that simplify Angular development. These tools are distributed in a package named `@angular/cli`. Run the command shown in Listing 2-3 to install the Angular development tools.

Listing 2-3. Installing the Angular Development Package

```
npm install --global @angular/cli@13.0.3
```

Notice that there are two hyphens before the `global` argument. If you are using Linux or macOS, you may need to use `sudo`, as shown in Listing 2-4.

Listing 2-4. Using `sudo` to Install the Angular Development Package

```
sudo npm install --global @angular/cli@13.0.3
```

Choosing a Browser

The final choice to make is the browser that you will use to check your work during development. All the current-generation browsers have good developer support and work well with Angular. I have used Google Chrome throughout this book, and this is the browser I recommend you use as well.

Creating an Angular Project

Angular development is done as part of a project, which contains all of the files required to build and execute an application, along with configuration files and static content (like HTML and CSS files). To create a new project, open a command prompt, navigate to a convenient location, and run the command shown in Listing 2-5. Pay close attention to the use of double and single hyphens when typing this command.

Listing 2-5. Creating a New Angular Project

```
ng new todo --routing false --style css --skip-git --skip-tests
```

The `ng` command is part of the `@angular/cli` package, and `ng new` sets up a new project. The arguments configure the project, selecting options that are suitable for a first project (the configuration options are described in Chapter 9). The process of creating a new project can take some time because there are a large number of other packages required, all of which must be downloaded the first time you run the `ng new` command.

Tip You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/Apress/pro-angular-5ed>. See Chapter 1 for how to get help if you have problems running the examples.

Opening the Project for Editing

Once the `ng new` command has finished, use your preferred code editor to open the `todo` folder that has been created and that contains the new project. The `todo` folder contains configuration files for the tools that are used in Angular development (described in Chapter 9), but it is the `src/app` folder that contains the application's code and content and is the folder in which most development is done. Figure 2-2 shows the initial content of the project folder as it appears in Visual Studio Code and highlights the `src/app` folder. You may see a slightly different view with other editors, some of which hide files and folders that are not often used directly during development, such as the `node_modules` folder, which contains the packages on which the Angular development tools rely.

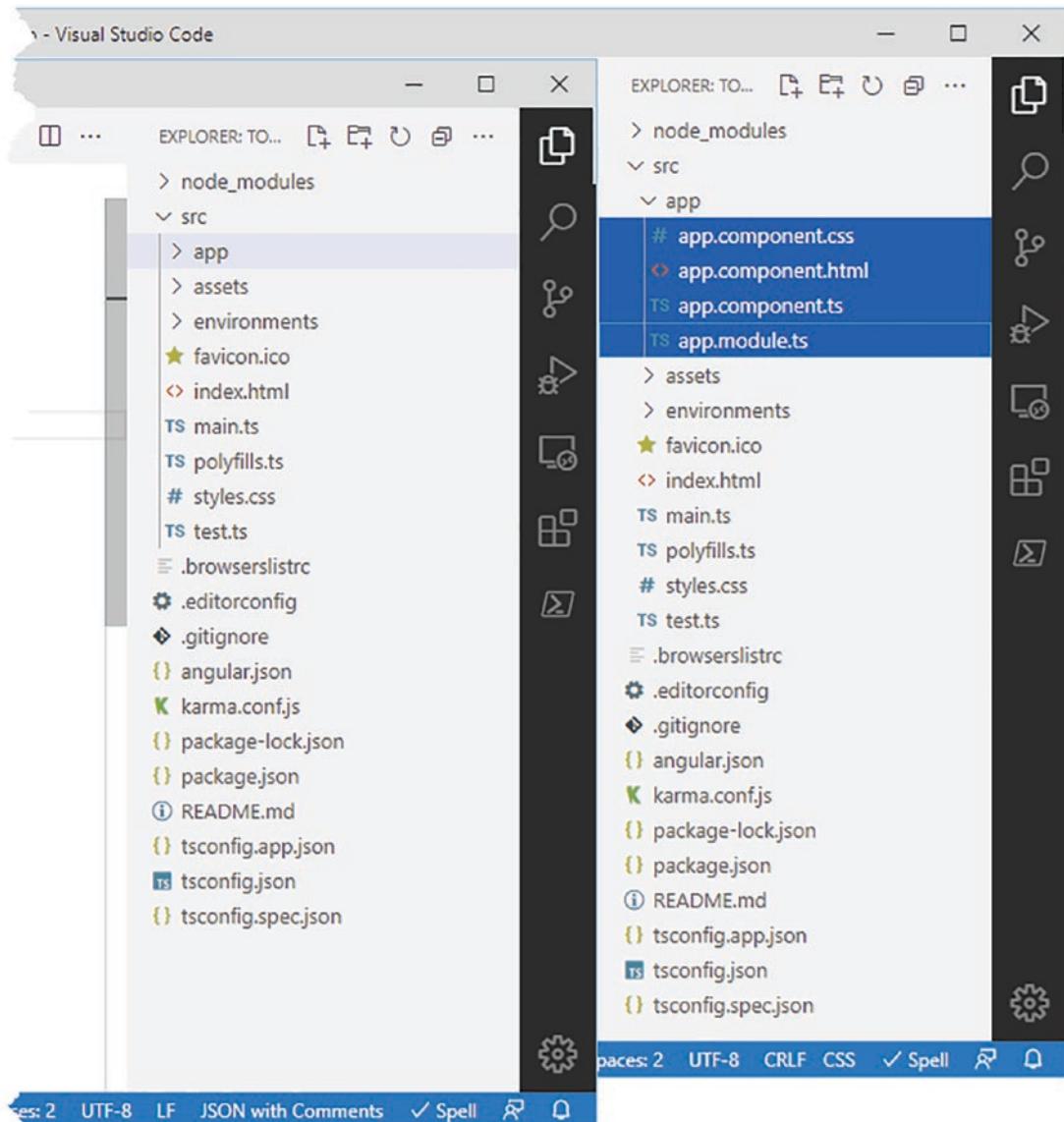


Figure 2-2. The initial contents of an Angular project

Starting the Angular Development Tools

The final part of the setup process is to start the development tools, which will compile the placeholder content added to the project by the `ng new` command. To start the Angular development tools, use a command prompt to run the command shown in Listing 2-6 in the todo folder.

Listing 2-6. Starting the Angular Development Tools

```
ng serve
```

This command starts the Angular development tools, which include a compiler and a web server that is used to test the Angular application in the browser. The development tools go through an initial startup process, which can take a moment to complete. During the startup process, you will see messages like these displayed by the `ng serve` command:

```
Browser application bundle generation complete.
Initial Chunk Files | Names           | Size
vendor.js           | vendor          | 1.83 MB
polyfills.js        | polyfills       | 339.11 kB
styles.css, styles.js | styles          | 212.38 kB
main.js             | main            | 51.42 kB
runtime.js          | runtime         | 6.84 kB
                           | Initial Total  | 2.43 MB
Build at: 2021-11-25T17:35:50.484Z - Hash: 8c7aa6b9cef0e8e7 - Time: 13641ms
** Angular Live Development Server is listening on localhost:4200, open your browser on
http://localhost:4200/ **
Compiled successfully.
```

Don't worry if you don't see the same output, just as long as you see the "compiled successfully" message at the end of the process. The integrated web server listens for requests on port 4200, so open a new browser window and request `http://localhost:4200`, which will show the placeholder content shown in Figure 2-3.

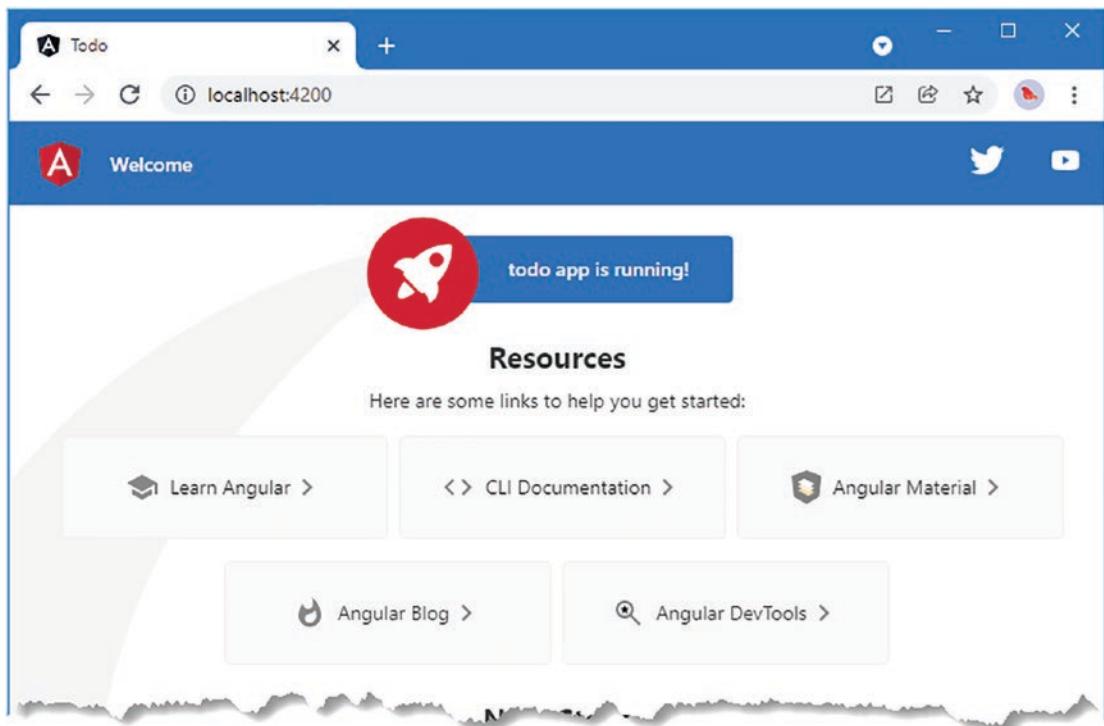


Figure 2-3. The placeholder content in a new Angular project

Adding Features to the Application

Now that the development tools are running, I am going to work through the process of creating a simple Angular application that will manage a to-do list. The user will be able to see the list of to-do items, check off items that are complete, and create new items. To keep the application simple, I assume that there is only one user and that I don't have to worry about preserving the state of the data in the application, which means that changes to the to-do list will be lost if the browser window is closed or reloaded. (Later examples, including the SportsStore application developed in Chapters 5-8, demonstrate persistent data storage.)

Creating a Data Model

The starting point for most applications is the data model, which describes the domain on which the application operates. Data models can be large and complex, but for my to-do application, I need to describe only two things: a to-do item and a list of those items.

Angular applications are written in TypeScript, which is a superset of JavaScript. I introduce TypeScript in Chapters 3 and 4, but its main advantage is that it supports static data types, which makes JavaScript development more familiar to C# and Java developers. (JavaScript has a prototype-based type system that many developers find confusing.) The `ng new` command includes the packages required to compile TypeScript code into pure JavaScript that can be executed by browsers.

To start the data model for the application, add a file called `todoItem.ts` to the `todo/src/app` folder with the contents shown in Listing 2-7. (TypeScript files have the `.ts` extension.)

Listing 2-7. The Contents of the todoItem.ts File in the src/app Folder

```
export class TodoItem {

    constructor(public task: string, public complete: boolean = false) {
        // no statements required
    }
}
```

The language features used in Listing 2-7 are a mix of standard JavaScript features and extra features that TypeScript provides. When the code is compiled, the TypeScript features are removed, and the result is JavaScript code that can be executed by browsers.

The `export`, `class`, and `constructor` keywords, for example, are standard JavaScript. Not all browsers support these features, so the build process for Angular applications can translate this type of feature into code that older browsers can understand, as I explain in Chapter 9.

The `export` keyword relates to JavaScript modules. When using modules, each TypeScript or JavaScript file is considered to be a self-contained unit of functionality, and the `export` keyword is used to identify data or types that you want to use elsewhere in the application. JavaScript modules are used to manage the dependencies that arise between files in a project. See Chapter 4 for details of how JavaScript modules are used.

The `class` keyword declares a class, and the `constructor` keyword denotes a class constructor. Unlike other languages, such as C#, JavaScript doesn't use the name of the class to denote the constructor.

Tip Don't worry if you are not familiar with these JavaScript/TypeScript features. Chapters 3 and 4 provide a primer for the JavaScript and TypeScript features that are most used in Angular development.

Other features in Listing 2-7 are provided by TypeScript. One of the most jarring features when you first start using TypeScript is its concise constructor feature, although you will quickly come to rely on it. The `TodoItem` class defines a constructor that receives two parameters, named `task` and `complete`. The values of these parameters are assigned to `public` properties of the same names. If no value is provided for the `complete` parameter, then a default value of `false` will be used:

```
...
constructor(public task: string, public complete: boolean = false) {
...
}
```

The concise constructor avoids a block of boilerplate code that would otherwise be required to define properties and assign them values that are received by the constructor.

The concise constructor syntax is helpful, but the headline TypeScript feature is static types. Both of the constructor parameters in Listing 2-7 are annotated with a data type:

```
...
constructor(public task: string, public complete: boolean = false) {
...
}
```

In standard JavaScript, values have types and can be assigned to any variable, which is a source of confusion to programmers who are used to variables that are defined to hold a specific data type. TypeScript adopts a more conventional approach to data types, and the TypeScript compiler will report an error if incompatible types are used. This may seem obvious if you are coming to Angular development from C# or Java, but it isn't the way that JavaScript usually works.

Creating the To-Do List Class

To create a class that represents a list of to-do items, add a file named `todoList.ts` to the `src/app` folder with the contents shown in Listing 2-8.

Listing 2-8. The Contents of the `todoList.ts` File in the `src/app` Folder

```
import { TodoItem } from "./todoItem";

export class TodoList {

    constructor(public user: string, private todoItems: TodoItem[] = []) {
        // no statements required
    }

    get items(): readonly TodoItem[] {
        return this.todoItems;
    }

    addItem(task: string) {
        this.todoItems.push(new TodoItem(task));
    }
}
```

The `import` keyword declares a dependency on the `TodoItem` class. The `TodoList` class defines a constructor that receives the initial set of to-do items. I don't want to give unrestricted access to the array of `TodoItem` objects, so I have defined a property named `items` that returns a read-only array, which is done using the `readonly` keyword. The TypeScript compiler will generate an error for any statement that attempts to modify the contents of the array, and if you are using an editor that has good TypeScript support, such as Visual Studio Code, then the autocomplete features of the editor won't present methods and properties that would trigger a compiler error.

Displaying Data to the User

I need a way to display the data values in the model to the user. In Angular, this is done using a *template*, which is a fragment of HTML that contains expressions that are evaluated by Angular and that inserts the results into the content that is sent to the browser.

When the project was created, the `ng new` command added a template file named `app.component.html` to the `src/app` folder. Open this file for editing and replace the contents with those shown in Listing 2-9.

Listing 2-9. Replacing the Contents of the `app.component.html` File in the `src/app` Folder

```
<h3>
  {{ username }}'s To Do List
  <h6>{{ itemCount }} Items</h6>
</h3>
```

I'll add features to the template shortly, but this is enough to get started. Displaying data in a template is done using double braces—`{{ and }}`—and Angular evaluates whatever you put between the double braces to get the value to display.

The {{ and }} characters are an example of a *data binding*, which means they create a relationship between the template and a data value. Data bindings are an important Angular feature, and you will see more of them in this chapter as I add features to the example application (and I describe them in detail in Part 2 of this book). In this case, the data bindings tell Angular to get the value of the `username` and `itemCount` properties and insert them into the content of the `h3` and `div` elements.

As soon as you save the file, the Angular development tools will try to build the project. The compiler will generate the following errors:

```
Error: src/app/app.component.html:2:6 - error TS2339: Property 'username' does not exist
on type 'AppComponent'.
2  {{ username }}'s To Do List
~~~~~
```

```
src/app/app.component.ts:5:16
5   templateUrl: './app.component.html',
~~~~~
```

Error occurs in the template of component AppComponent.

```
Error: src/app/app.component.html:3:10 - error TS2339: Property 'itemCount' does not exist
on type 'AppComponent'.
3  <h6>{{ itemCount }} Items</h6>
~~~~~
```

```
src/app/app.component.ts:5:16
5   templateUrl: './app.component.html',
~~~~~
```

Error occurs in the template of component AppComponent.

These errors occur because the expressions within the data bindings rely on properties that don't exist. I'll fix this problem in the next section, but these errors show an important Angular characteristic, which is that templates are included in the compilation process and that any errors in the template are handled just like errors in regular code files.

Updating the Component

An Angular *component* is responsible for managing a template and providing it with the data and logic it needs. If that seems like a broad statement, it is because components are the part of an Angular application that does most of the heavy lifting. As a consequence, they can be used for all sorts of tasks.

In this case, I need a component to act as a bridge between the data model classes and the template so that I can create an instance of the `TodoList` class, populate it with some sample `TodoItem` objects, and, in doing so, provide the template with the `username` and `itemCount` properties it needs.

When the project was created, the `ng new` command added a file named `app.component.ts` to the `src/app` folder. As the name of the file suggests, this is a component. Apply the changes shown in Listing 2-10 to the `app.component.ts` file.

Listing 2-10. Editing the Contents of the `app.component.ts` File in the `src/app` Folder

```
import { Component } from '@angular/core';
import { TodoList } from "./todoList";
import { TodoItem } from "./todoItem";
```

```

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  private list = new TodoList("Bob", [
    new TodoItem("Go for run", true),
    new TodoItem("Get flowers"),
    new TodoItem("Collect tickets"),
  ]);

  get username(): string {
    return this.list.user;
  }

  get itemCount(): number {
    return this.list.items
      .filter(item => !item.complete).length;
  }
}

```

The code in the listing can be broken into three main regions, as described in the following sections.

Understanding the Imports

The `import` keyword declares dependencies on JavaScript modules, both within the project and in third-party packages. The `import` keyword is used three times in Listing 2-10:

```

...
import { Component } from '@angular/core';
import { TodoList } from './todoList';
import { TodoItem } from './todoItem';
...

```

The first `import` statement is used in the listing to load the `@angular/core` module, which contains the key Angular functionality, including support for components. When working with modules, the `import` statement specifies the types that are imported between curly braces. In this case, the `import` statement is used to load the `Component` type from the module. The `@angular/core` module contains many classes that have been packaged together so that the browser can load them all in a single JavaScript file.

The other `import` statements are used to declare dependencies on the data model classes defined earlier. The target for this kind of import starts with `./`, which indicates that the module is defined relative to the current file.

Notice that the `import` statements do not include file extensions. This is because the relationship between the target of an `import` statement and the file that is loaded by the browser is handled by the Angular build tools, which I explain in more detail in Chapter 9.

Understanding the Decorator

The oddest-looking part of the code in the listing is this:

```
...
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
...
```

This is an example of a *decorator*, which provides metadata about a class. This is the `@Component` decorator, and, as its name suggests, it tells Angular that this is a component. The decorator provides configuration information through its properties. This `@Component` decorator specifies three properties: `selector`, `templateUrl`, and `styleUrls`.

The `selector` property specifies a CSS selector that matches the HTML element to which the component will be applied.

When you request `http://localhost:4200`, the browser receives the contents of the `index.html` file, which was added to the `src` folder when the project was created. This file contains a custom HTML element, like this:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Todo</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

The Angular development tools automatically add `script` elements to this HTML, which instruct the browser to request the JavaScript files that provide the Angular framework and the custom features defined in the project.

When the Angular code is executed, the value of the `selector` property defined by the component is used to locate the specified element in the HTML document, and it is this element into which the content generated by the application is introduced. I am skipping over some details for brevity in this chapter, but I return to this topic in more detail in later chapters. For now, it is enough to understand that the value of the component decorator's `selector` property corresponds to the element in the HTML document.

The `templateUrl` property is to specify the component's template, which is the `app.component.html` file for this component and is the file edited in Listing 2-9.

The `styleUrls` property specifies one or more CSS stylesheets that are used to style the elements produced by the component and its template. The setting in this component specifies a file named `app.component.css`, which I use later in the chapter to create CSS styles.

Understanding the Class

The final part of the listing defines a class that Angular can instantiate to create the component.

```
...
export class AppComponent {
  private list = new TodoList("Bob", [
    new TodoItem("Go for run", true),
    new TodoItem("Get flowers"),
    new TodoItem("Collect tickets"),
  ]);
  ...
  get username(): string {
    return this.list.user;
  }
  get itemCount(): number {
    return this.list.items.filter(item => !item.complete).length;
  }
}
...
...
```

These statements define a class called `AppComponent` that has a private `list` property, which is assigned a `TodoList` object and is populated with an array of `TodoItem` objects. The `AppComponent` class defines read-only properties named `username` and `itemCount` that rely on the `TodoList` object to produce their values. The `username` property returns the value of the `TodoList.user` property, and the `itemCount` property uses the standard JavaScript array features to filter the `TodoItem` objects managed by the `TodoList` to select those that are incomplete and returns the number of matching objects it finds.

The value for the `itemCount` property is produced using a *lambda function*, also known as a *fat arrow function*, which is a more concise way of expressing a standard JavaScript function. The arrow in the lambda expressions is read as “goes to” such as “`item` goes to not `item.complete`.”

When you save the changes to the TypeScript file, the Angular development tools will build the project. There should be no errors this time because the component has defined the properties that the template requires. The browser window will be automatically reloaded, showing the output in Figure 2-4.

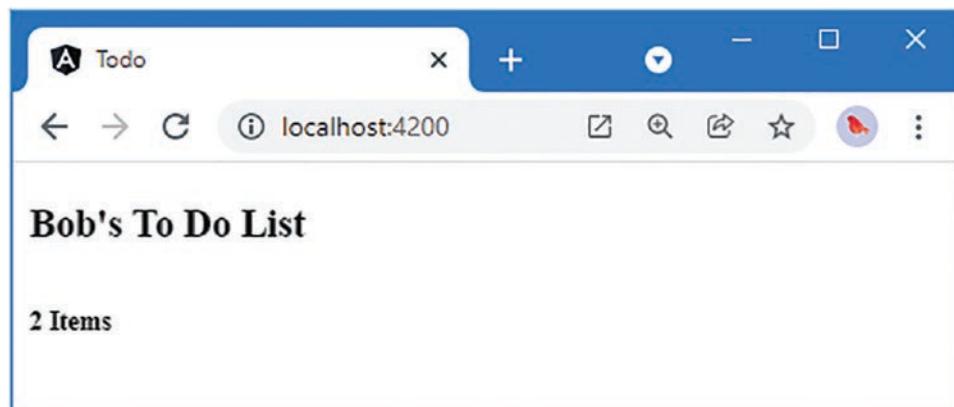


Figure 2-4. Generating content in the example application

Styling the Application Content

To style the HTML content produced by the application, I am going to use the Angular Material package, which contains a set of components for use in Angular applications. Angular Material is as close as you can get to an “official” component library, and it has the advantage of being free to use, full of useful features, and well-integrated into the rest of the Angular framework.

Note Angular Material isn’t the only component package available, and as you will see in later chapters, you don’t need to use third-party components at all if that is your preference.

Use Control+C to stop the Angular development tools, and use the command prompt to run the command shown in Listing 2-11 in the todo folder.

Listing 2-11. Adding the Angular Material Package

```
ng add @angular/material@13.0.2 --defaults
```

When prompted, press Y to install the package. Once the package has been installed, open the app.module.ts file in the src folder and make the changes shown in Listing 2-12. These changes declare dependencies on the Angular Material features that are used in this chapter. Confusingly, this file is also called a *module*, which means that there are two types of modules in an Angular project: JavaScript modules and Angular modules. This is an example of an Angular module, which is described in more detail in Chapter 9. For this chapter, it is enough to know that this is how features from the Angular Material package are included in the example project.

Listing 2-12. Adding Dependencies in the app.module.ts File in the src Folder

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

import { FormsModule } from '@angular/forms'
import { MatButtonModule } from '@angular/material/button';
import { MatToolbarModule } from '@angular/material/toolbar';
import { MatIconModule } from '@angular/material/icon';
import { MatBadgeModule } from '@angular/material/badge';
import { MatTableModule } from '@angular/material/table';
import { MatCheckboxModule } from '@angular/material/checkbox';
import { MatFormFieldModule } from '@angular/material/form-field';
import { MatInputModule } from '@angular/material/input';
import { MatSlideToggleModule } from '@angular/material/slide-toggle';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
```

```

imports: [
  BrowserModule,
  BrowserAnimationsModule,
  FormsModule,
  MatButtonModule, MatToolbarModule, MatIconModule, MatBadgeModule,
  MatTableModule, MatCheckboxModule, MatFormFieldModule, MatInputModule,
  MatSlideToggleModule
],
providers: [],
bootstrap: [AppComponent]
})
export class AppModule { }

```

Each feature used by the application increases the amount of JavaScript code that must be downloaded by the browser, which is why features are enabled individually. You must pay close attention to the changes shown in Listing 2-12 because errors will prevent the example application from working as expected. If you encounter issues, then compare your file with the one included in the GitHub repository for this book, which can be found at <https://github.com/Apress/pro-angular-5ed>.

Applying Angular Material Components

The next step is to use components contained in the Angular Material package to style the content produced by the application. Components are applied using HTML elements and attributes in the template file, as shown in Listing 2-13.

Listing 2-13. Applying Components in the app.components.html File in the src/app Folder

```

<mat-toolbar color="primary" class="mat-elevation-z3">
  <span>{{ username }}'s To Do List</span>
  <mat-icon matBadge="{{ itemCount }}" matBadgeColor="accent">checklist</mat-icon>
</mat-toolbar>

```

The new template content relies on features from the Angular Material package, each of which is applied differently. The first feature is the toolbar, which is applied using the `mat-toolbar` element, with the contents of the toolbar contained within the opening and closing tag:

```

...
<mat-toolbar color="primary" class="mat-elevation-z3">
...
</mat-toolbar>
...

```

The `color` attribute is used to specify the color for the toolbar. The Angular Material package uses color themes, and the `primary` value used to configure the toolbar represents the predominant color of the theme.

The class that the `mat-toolbar` element has been assigned applies a style provided by the Angular Material package for creating a raised appearance for content:

```

...
<mat-toolbar color="primary" class="mat-elevation-z3">
...

```

The other features are an icon and a badge, which are used together to indicate how many incomplete items are in the user's to-do list. The icon is applied using the `mat-icon` element:

```
...
<mat-icon matBadge="{{ itemCount }}" matBadgeColor="accent">checklist</mat-icon>
...
```

Icons are selected by specifying a name as the content of the `mat-icon` element. In this case, the `checklist` icon has been selected:

```
...
<mat-icon matBadge="{{ itemCount }}" matBadgeColor="accent">checklist</mat-icon>
...
```

You can see the complete set of icons that are available by visiting <https://fonts.google.com/icons?selected=Material+Icons>. Icons are distributed using font files, and the command used to add Angular Material to the project in Listing 2-13 adds the links required for these files to the `index.html` file in the `src` folder.

The badge is applied as an option to the `mat-icon` element using the `matBadge` and `matBadgeColor` attributes:

```
...
<mat-icon matBadge="{{ itemCount }}" matBadgeColor="accent">checklist</mat-icon>
...
```

Badges are small circular status indicators, used to present the user with a number or characters, which makes them ideal for indicating how many to-do items there are. The value of the `matBadge` attribute sets the content of the badge, and the `matBadgeColor` attribute is used to set the color, which is `accent` in this case, denoting the theme color that is used for highlighting.

Save the changes to the template file and use the `ng serve` command to start the Angular development tools. Once the tool startup sequence is complete, use a browser to request `http://localhost:4200`, and you will see the content shown in Figure 2-5.

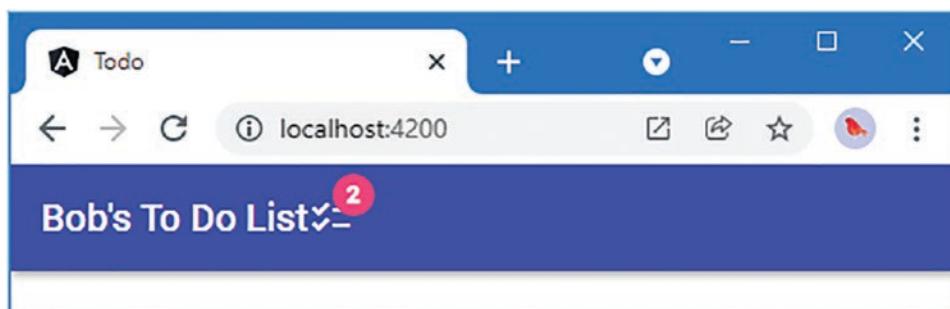


Figure 2-5. Introducing Angular Material components

I improve the layout in the next section, but the addition of the Angular Material components is already an improvement over the raw HTML content. Notice that the template in Listing 2-13 still contains the same data bindings introduced earlier in the chapter, and they still work in the same way, providing access to the data provided by the component.

Defining the Spacer CSS Style

The Angular Material package is generally comprehensive, but one omission is spacers to help position content. I want to position the `span` element that contains the user's name centrally within the title bar and have the icon and badge appear on the right. The first step is to create a CSS class that will configure HTML elements to grow to fill available space. As noted earlier, the decorator in the `app.component.ts` file contains a `styleUrls` property, which is used to select CSS files that are applied to the component's template. Add the style shown in Listing 2-14 to the `app.component.css` file, which is the file specified by default when the project is created.

Listing 2-14. Adding a CSS Style in the `app.component.css` File in the `src/app` Folder

```
.spacer { flex: 1 1 auto }
```

The addition in Listing 2-14 applies a style to any element assigned to a class named `spacer`. The style sets the `flex` property, which is part of the CSS *flexible box* feature, also known as *flexbox*. Flexbox is used to lay out HTML elements so they adapt to the space that is available and can respond to changes, such as when a browser window is resized or a device screen is rotated. The setting in Listing 2-14 configures an element to grow to fill any available space, and if there are multiple HTML elements in the same container assigned to the `spacer` class, then the available space will be allocated evenly between them. Add the elements shown in Listing 2-15 to the template file to introduce spacers into the layout.

Listing 2-15. Adding Elements in the `app.component.html` File in the `src/app` Folder

```
<mat-toolbar color="primary" class="mat-elevation-z3">
  <span class="spacer"></span>
  {{ username }}'s To Do List
  <span class="spacer"></span>
  <mat-icon matBadge="{{ itemCount }}" matBadgeColor="accent">checklist</mat-icon>
</mat-toolbar>
```

When you save the file, the Angular development tools will detect the changes, recompile the project, and trigger a browser reload, producing the new layout shown in Figure 2-6.

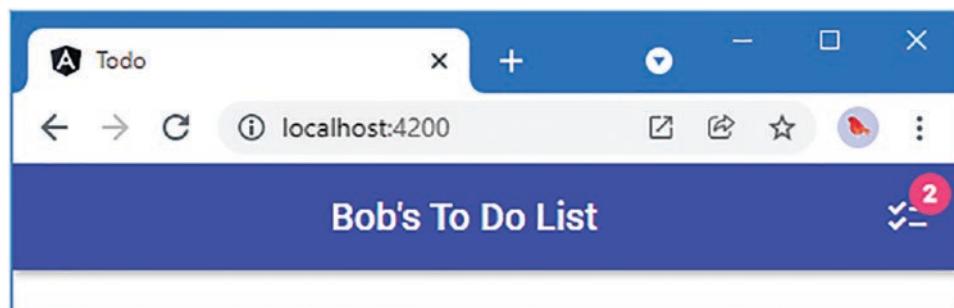


Figure 2-6. Adding spacers to the component layout

Displaying the List of To-Do Items

The next step is to display the to-do items. Listing 2-16 adds a property to the component that provides access to the items in the list.

Listing 2-16. Adding a Property in the app.component.ts File in the src/app Folder

```
import { Component } from '@angular/core';
import { TodoList } from './todoList';
import { TodoItem } from './todoItem';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  private list = new TodoList("Bob", [
    new TodoItem("Go for run", true),
    new TodoItem("Get flowers"),
    new TodoItem("Collect tickets"),
  ]);

  get username(): string {
    return this.list.user;
  }

  get itemCount(): number {
    return this.list.items.filter(item => !item.complete).length;
  }

  get items(): readonly TodoItem[] {
    return this.list.items;
  }
}
```

To display details of each item to the user, I am going to use the Angular Material table component, as shown in Listing 2-17, which makes it easy to present the user with tabular data. (I explain how you can create your own equivalent to the table component in Part 2, using the same Angular features as the Angular Material package.)

Listing 2-17. Adding a Table in the app.component.html File in the src/app Folder

```
<mat-toolbar color="primary" class="mat-elevation-z3">
  <span class="spacer"></span>
  <span>{{ username }}'s To Do List</span>
  <span class="spacer"></span>
  <mat-icon matBadge="{{ itemCount }}" matBadgeColor="accent">checklist</mat-icon>
</mat-toolbar>

<div class="tableContainer">
  <table mat-table [dataSource]="items" class="mat-elevation-z3 fullWidth">
```

```

<ng-container matColumnDef="id">
  <th mat-header-cell *matHeaderCellDef>#</th>
  <td mat-cell *matCellDef="let i = index"> {{ i + 1 }} </td>
</ng-container>

<ng-container matColumnDef="task">
  <th mat-header-cell *matHeaderCellDef>Task</th>
  <td mat-cell *matCellDef="let item"> {{ item.task }} </td>
</ng-container>

<ng-container matColumnDef="done">
  <th mat-header-cell *matHeaderCellDef>Done</th>
  <td mat-cell *matCellDef="let item"> {{ item.complete }} </td>
</ng-container>

<tr mat-header-row *matHeaderRowDef="['id', 'task', 'done']"></tr>
<tr mat-row *matRowDef="let row; columns: ['id', 'task', 'done'];"></tr>
</table>
</div>

```

The Angular Material table component is applied by adding the `mat-table` attribute to a standard HTML table element, and the data the table will contain is specified using the `dataSource` attribute:

```

...
<table mat-table [dataSource]="items" class="mat-elevation-z3 fullWidth">
...

```

The square brackets (the [and] characters) denote an attribute binding, which is a data binding that is used to set an element attribute, providing the Angular Material table component with the data that it will display. Angular defines a range of data bindings for use in different situations, and these are all described in detail in Part 2. This binding configures the table to display the values returned by the `items` property defined in Listing 2-16.

The table component is configured by defining the columns that will be displayed to the user. The `ng-container` element is used to group content together, and, in this case, it is used to group the elements that define a header and a content cell for a column, like this:

```

...
<ng-container matColumnDef="task">
  <th mat-header-cell *matHeaderCellDef>Task</th>
  <td mat-cell *matCellDef="let item"> {{ item.task }} </td>
</ng-container>
...

```

This arrangement of elements defines the header and content table cells for a column named `task`. The header cell is defined using a `th` element to which the `mat-header-cell` and `*matHeaderCellDef` attributes have been applied:

```

...
<th mat-header-cell *matHeaderCellDef>Task</th>
...

```

The effect of the `mat-header-cell` attribute is to configure the appearance of the header cell so that it matches the rest of the table. The effect of the `*matHeaderCellDef` attribute is to configure the behavior of the cell.

Note When you start working with Angular, the template syntax can feel arcane and impenetrable, with endless combinations of curly braces, square braces, and asterisks. All of these features are described in later chapters, but for now, make sure you don't omit the asterisks from the attributes when they are shown in the listings.

The content cell is defined using a `td` element to which the `mat-cell` and `*matCellDef` attributes are applied. The `*matCellDef` attribute is used to select the content that will be displayed in each table cell:

```
...
<td mat-cell *matCellDef="let item"> {{ item.task }} </td>
...
```

I explain how this feature works in detail in Part 2, but for the moment, it is enough to know that the expression assigned to the `*matCellDef` attribute will be evaluated for each element in the data source, which will be assigned to a variable named `item`, and this variable is used in a data binding to populate the table cell. In this case, the value of the `task` property will be displayed in the table cell.

The Angular Material table component provides additional context data as it creates table rows, including the index of the data item for which the current row is being created and which can be accessed like this:

```
...
<td mat-cell *matCellDef="let i = index"> {{ i + 1 }} </td>
...
```

The expression used for this table cell assigns the `index` value provided by the table component to a variable named `i`, which is used in the data binding to produce a simple counter.

The columns for the table header and body are selected by applying attributes to `tr` elements, like this:

```
...
<tr mat-header-row *matHeaderRowDef="['id', 'task', 'done']"></tr>
<tr mat-row *matRowDef="let row; columns: ['id', 'task', 'done'];"></tr>
...
```

These elements select the `id`, `task`, and `done` rows defined in Listing 2-17. It may seem odd that the columns are not applied automatically, but this approach is useful when you want to select different columns based on user input.

Defining Additional Styles

The final step of setting up the table is to define additional CSS styles, as shown in Listing 2-18.

Listing 2-18. Defining Styles in the app.component.css File in the src/app Folder

```
.spacer { flex: 1 1 auto }
.tableContainer { padding: 15px }
.fullWidth { width: 100% }
```

The first new style selects any element that has been assigned to the `tableContainer` class and applies padding around it. There is a `div` element in Listing 2-18 that I added to this class and that contains the `table` element. The second new style sets elements assigned to the `fullWidth` class to occupy 100 percent of the width available to them.

Save the changes, and the Angular development tools will compile the project and reload the browser, producing the content shown in Figure 2-7.

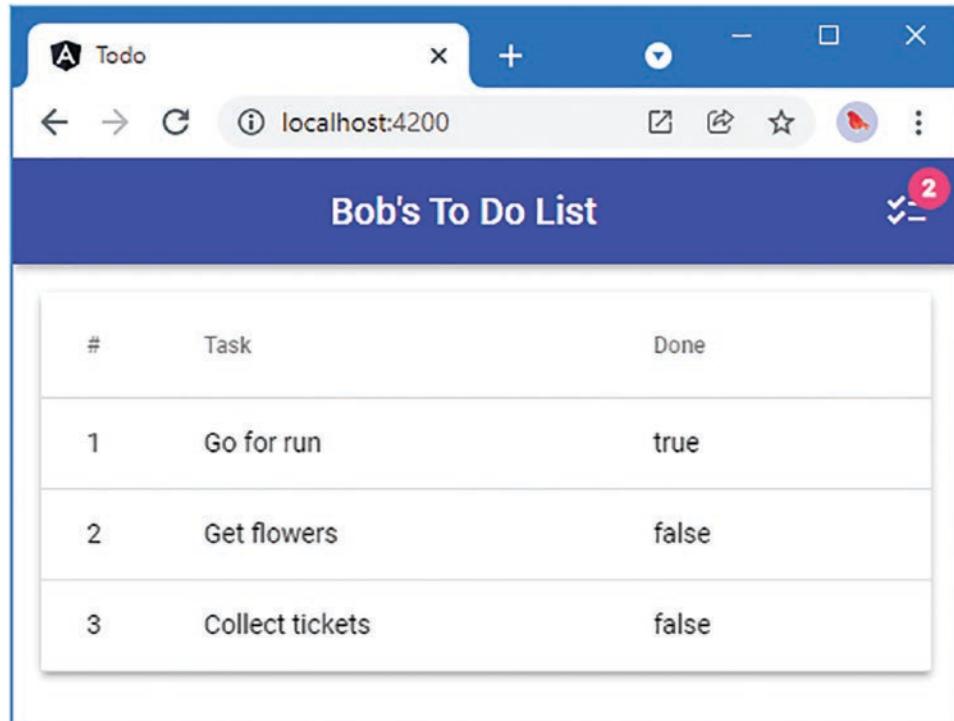


Figure 2-7. Displaying the list of to-do items

Creating a Two-Way Data Binding

At the moment, the template contains only *one-way data bindings*, which means they are used to display a data value but are unable to change it. Angular also supports *two-way data bindings*, which can be used to display a data value and change it, too. Two-way bindings are used with HTML form elements, and Listing 2-19 adds an Angular Material checkbox to the template that allows users to mark a to-do item as complete.

Listing 2-19. Adding a Checkbox in the app.component.html File in the src/app Folder

```

<mat-toolbar color="primary" class="mat-elevation-z3">
  <span class="spacer"></span>
  <span>{{ username }}'s To Do List</span>
  <span class="spacer"></span>
  <mat-icon matBadge="{{ itemCount }}" matBadgeColor="accent">checklist</mat-icon>
</mat-toolbar>

<div class="tableContainer">
  <table mat-table [dataSource]="items" class="mat-elevation-z3 fullWidth">

    <ng-container matColumnDef="id">
      <th mat-header-cell *matHeaderCellDef>#</th>
      <td mat-cell *matCellDef="let i = index"> {{ i + 1 }} </td>
    </ng-container>

    <ng-container matColumnDef="task">
      <th mat-header-cell *matHeaderCellDef>Task</th>
      <td mat-cell *matCellDef="let item"> {{ item.task }} </td>
    </ng-container>

    <ng-container matColumnDef="done">
      <th mat-header-cell *matHeaderCellDef>Done</th>
      <td mat-cell *matCellDef="let item">
        <mat-checkbox [(ngModel)]="item.complete" color="primary">
          {{ item.complete }}
        </mat-checkbox>
      </td>
    </ng-container>

    <tr mat-header-row *matHeaderRowDef="['id', 'task', 'done']"></tr>
    <tr mat-row *matRowDef="let row; columns: ['id', 'task', 'done'];"></tr>
  </table>
</div>

```

The `mat-checkbox` element applies the Angular Material checkbox component. The two-way binding is expressed using a special attribute:

```

...
<mat-checkbox [(ngModel)]="item.complete" color="primary">
...

```

The combination of brackets is known as the *banana-in-a-box* because the round brackets look like a banana contained in a box made by the square brackets. These brackets denote a two-way data binding, and `ngModel` is an Angular feature and is used to set up two-way bindings on form elements, such as checkboxes.

Save the changes to the file, and the Angular development tools will recompile the project and reload the browser to display the content shown in Figure 2-8. The effect is that the `complete` property of each to-do item is used to set a checkbox when it is displayed to the user. The appropriate `complete` property will also be updated when the user toggles the checkbox.

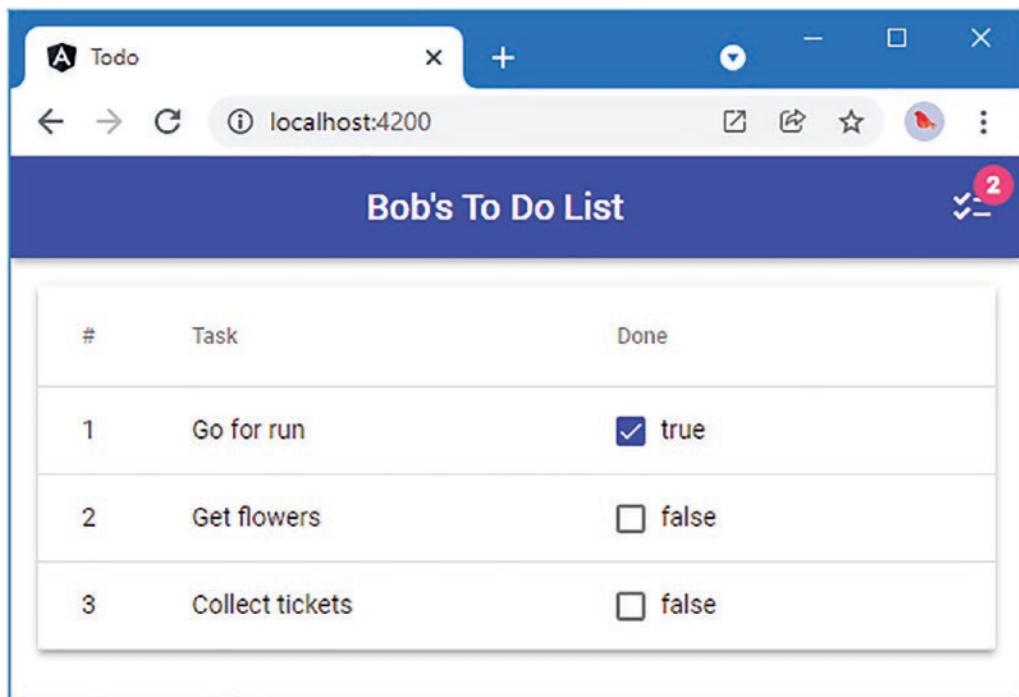


Figure 2-8. Using two-way bindings

I left the true/false values in the output to demonstrate an important aspect of how Angular deals with changes. Each time you toggle a checkbox, the corresponding text value changes and so does the counter displayed by the badge, as shown in Figure 2-9.

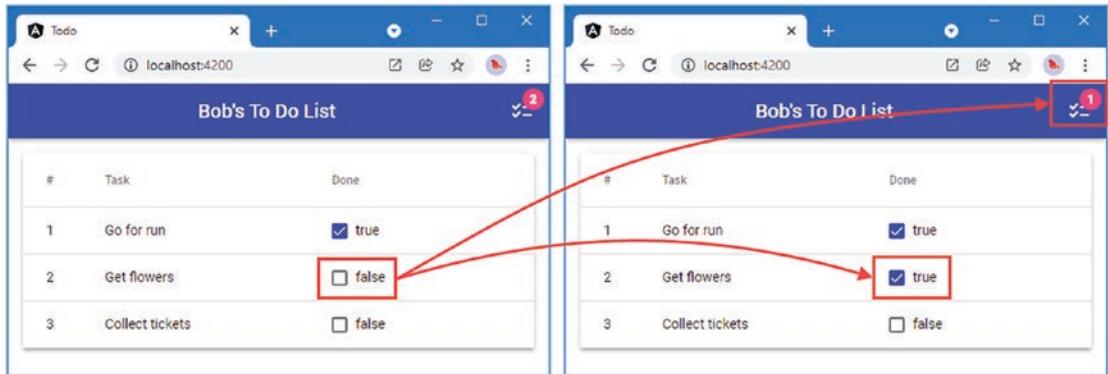


Figure 2-9. Toggling a checkbox

This behavior reveals an important Angular feature: the data model is *live*. This means data bindings—even one-way data bindings—are updated when the data model is changed. This simplifies web application development because it means you don't have to worry about ensuring that you display updates when the application state changes.

It can be easy to forget that underneath the templates and components and the live data model, Angular is using the browser's JavaScript API to create and display regular HTML elements. Right-click one of the checkboxes in the browser window and select Inspect or Inspect Element from the pop-up menu (the exact menu item will depend on your chosen browser). The browser's developer tools will open, and you can explore the HTML content displayed by the browser. You may have to dig around a little by expanding elements to see their contents, but you will see that the effect of applying an Angular Material checkbox in the template is a regular HTML checkbox, like this:

```
...
<input type="checkbox" class="mat-checkbox-input cdk-visually-hidden"
   id="mat-checkbox-1-input" tabindex="0" aria-checked="false">
...

```

If you find yourself confused by the way an Angular application behaves, then a good place to start is to examine the elements displayed by the browser, which reveals the effects created by your templates and components. There are other diagnostic tools available, as I explain in Chapter 9, but this is a simple and effective way to understand what an application is doing.

Filtering Completed To-Do Items

The checkboxes allow the data model to be updated, and the next step is to remove to-do items once they have been marked as done. Listing 2-20 changes the component's `items` property so that it filters out any items that have been completed.

Listing 2-20. Filtering To-Do Items in the app.component.ts File in the src/app Folder

```
import { Component } from '@angular/core';
import { TodoList } from './todoList';
import { TodoItem } from './todoItem';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  private list = new TodoList("Bob", [
    new TodoItem("Go for run", true),
    new TodoItem("Get flowers"),
    new TodoItem("Collect tickets"),
  ]);

  get username(): string {
    return this.list.user;
  }

  get itemCount(): number {
    return this.list.items.filter(item => !item.complete).length;
  }
}
```

```

get items(): readonly TodoItem[] {
  return this.list.items.filter(item => !item.complete);
}
}

```

The filter method is a standard JavaScript array feature. This is the same expression I used previously in the itemCount property. I could rework this property to avoid duplication, but I will add support for choosing whether completed tasks should be shown later in the chapter, which will require the items and itemCount properties to process the list of to-do items differently.

Since the data model is live and changes are reflected in data bindings immediately, checking the checkbox for an item removes it from view, as shown in Figure 2-10.

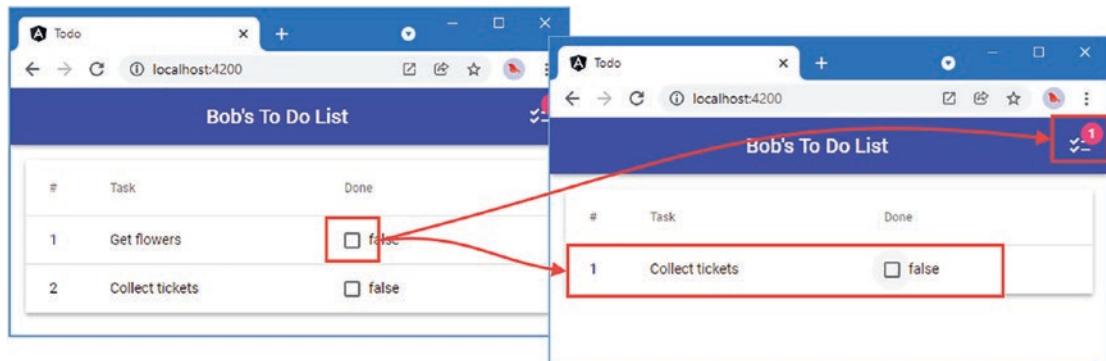


Figure 2-10. Filtering the to-do items

Adding To-Do Items

A to-do application isn't much use without the ability to add new items to the list. Listing 2-21 uses Angular Material components to present the user with an input element, into which a task description can be entered, and with a button that will use the description to create a new to-do item.

Listing 2-21. Adding Elements in the app.component.html File in the src/app Folder

```

<mat-toolbar color="primary" class="mat-elevation-z3">
  <span class="spacer"></span>
  <span>{{ username }}'s To Do List</span>
  <span class="spacer"></span>
  <mat-icon matBadge="{{ itemCount }}" matBadgeColor="accent">checklist</mat-icon>
</mat-toolbar>

<div class="inputContainer">
  <mat-form-field class="fullWidth">
    <mat-label style="padding-left: 5px;">New To Do</mat-label>
    <input matInput placeholder="Enter to-do description" #todoText>
    <button matSuffix mat-raised-button color="accent" class="addButton"
      (click)="addItem(todoText.value); todoText.value = ''">

```

```

Add
</button>
</mat-form-field>
</div>

<div class="tableContainer">
  <table mat-table [dataSource]="items" class="mat-elevation-z3 fullWidth">

    <ng-container matColumnDef="id">
      <th mat-header-cell *matHeaderCellDef>#</th>
      <td mat-cell *matCellDef="let i = index"> {{ i + 1 }} </td>
    </ng-container>

    <ng-container matColumnDef="task">
      <th mat-header-cell *matHeaderCellDef>Task</th>
      <td mat-cell *matCellDef="let item"> {{ item.task }} </td>
    </ng-container>

    <ng-container matColumnDef="done">
      <th mat-header-cell *matHeaderCellDef>Done</th>
      <td mat-cell *matCellDef="let item">
        <mat-checkbox [(ngModel)]="item.complete" color="primary">
          {{ item.complete }}
        </mat-checkbox>
      </td>
    </ng-container>

    <tr mat-header-row *matHeaderRowDef="['id', 'task', 'done']"></tr>
    <tr mat-row *matRowDef="let row; columns: ['id', 'task', 'done'];"></tr>
  </table>
</div>

```

The new elements display an `input` element and a `button` element. The `mat-form-field` element and the `mat*` attributes on the other elements configure the Angular Material styling.

The `input` element has an attribute whose name starts with the `#` character, which is used to define a variable to refer to the element in the template's data bindings:

```

...
<input matInput placeholder="Enter to-do description" #todoText

```

The name of the variable is `todoText`, and it is used by the binding that has been applied to the `button` element.

```

...
<button matSuffix mat-raised-button color="accent" class="addButton"
  (click)="addItem(todoText.value); todoText.value = ''">
...

```

This is an example of an *event binding*, and it tells Angular to invoke a component method called `addItem`, using the `value` property of the `input` element as the method argument, and then to clear the `input` element by setting its `value` property to the empty string.

Custom CSS styles are required to manage the layout of the new elements, as shown in Listing 2-22.

Listing 2-22. Defining Styles in the app.component.css File in the src/app Folder

```
.spacer { flex: 1 1 auto }
.tableContainer { padding: 15px }
.fullWidth { width: 100% }
.inputContainer { margin: 15px 15px 5px }
.addButton { margin: 5px }
```

Listing 2-23 adds the method called by the event binding to the component.

Tip Don't worry about telling the bindings apart for now. I explain the different types of binding that Angular supports in Part 2 and the meaning of the different types of brackets or parentheses that each requires. They are not as complicated as they first appear, especially once you have seen how they fit into the rest of the Angular framework.

Listing 2-23. Adding a Method in the app.component.ts File in the src/app Folder

```
import { Component } from '@angular/core';
import { TodoList } from './todoList';
import { TodoItem } from './todoItem';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  private list = new TodoList("Bob", [
    new TodoItem("Go for run", true),
    new TodoItem("Get flowers"),
    new TodoItem("Collect tickets"),
  ]);

  get username(): string {
    return this.list.user;
  }

  get itemCount(): number {
    return this.list.items.filter(item => !item.complete).length;
  }

  get items(): readonly TodoItem[] {
    return this.list.items.filter(item => !item.complete);
  }
}
```

```

addItem(newItem: string) {
  if (newItem != "") {
    this.list.addItem(newItem);
  }
}
}

```

The `addItem` method receives the text sent by the event binding in the template and uses it to add a new item to the to-do list. The result of these changes is that you can create new to-do items by entering text in the input element and clicking the Add button, as shown in Figure 2-11.

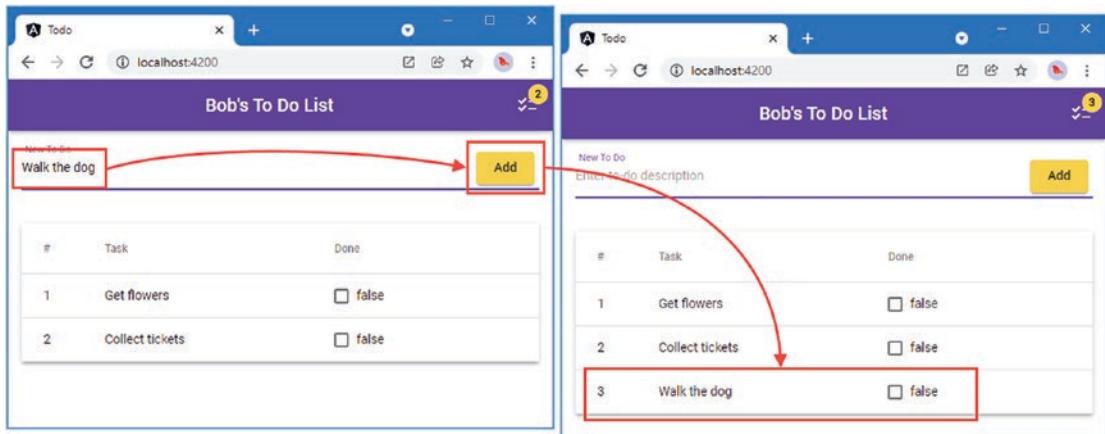


Figure 2-11. Creating a to-do item

Finishing Up

The basic features are in place, and now it is time to wrap up the project. In Listing 2-24, I removed the true/false text from the Done column in the table from the template and added an option to show completed tasks.

Listing 2-24. Modifying the Template in the app.component.html File in the src/app Folder

```

<mat-toolbar color="primary" class="mat-elevation-z3">
  <span class="spacer"></span>
  <span>{{ username }}'s To Do List</span>
  <span class="spacer"></span>
  <mat-icon matBadge="{{ itemCount }}" matBadgeColor="accent">checklist</mat-icon>
</mat-toolbar>

<div class="inputContainer">
  <mat-form-field class="fullWidth">
    <mat-label style="padding-left: 5px;">New To Do</mat-label>
    <input matInput placeholder="Enter to-do description" #todoText>

```

```

<button matSuffix mat-raised-button color="accent" class="addButton"
        (click)="addItem(todoText.value); todoText.value = ''">
    Add
</button>
</mat-form-field>
</div>

<div class="tableContainer">
    <table mat-table [dataSource]="items" class="mat-elevation-z3 fullWidth">

        <ng-container matColumnDef="id">
            <th mat-header-cell *matHeaderCellDef>#</th>
            <td mat-cell *matCellDef="let i = index"> {{ i + 1 }} </td>
        </ng-container>

        <ng-container matColumnDef="task">
            <th mat-header-cell *matHeaderCellDef>Task</th>
            <td mat-cell *matCellDef="let item"> {{ item.task }} </td>
        </ng-container>

        <ng-container matColumnDef="done">
            <th mat-header-cell *matHeaderCellDef>Done</th>
            <td mat-cell *matCellDef="let item">
                <mat-checkbox [(ngModel)]="item.complete" color="primary">
                    <!-- {{ item.complete }} -->
                </mat-checkbox>
            </td>
        </ng-container>

        <tr mat-header-row *matHeaderRowDef="['id', 'task', 'done']"></tr>
        <tr mat-row *matRowDef="let row; columns: ['id', 'task', 'done'];"></tr>
    </table>
</div>

<div class="toggleContainer">
    <span class="spacer"></span>
    <mat-slide-toggle [(ngModel)]="showComplete">
        Show Completed Items
    </mat-slide-toggle>
    <span class="spacer"></span>
</div>

```

The new elements present a toggle switch that has a two-way data binding for a property named `showComplete`. Listing 2-25 adds the definition for the `showComplete` property to the component and uses its value to determine whether completed tasks are displayed to the user.

Listing 2-25. Showing Completed Tasks in the app.component.ts File in the src/app Folder

```

import { Component } from '@angular/core';
import { TodoList } from './todoList';
import { TodoItem } from './todoItem';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  private list = new TodoList("Bob", [
    new TodoItem("Go for run", true),
    new TodoItem("Get flowers"),
    new TodoItem("Collect tickets"),
  ]);

  get username(): string {
    return this.list.user;
  }

  get itemCount(): number {
    return this.list.items.filter(item => !item.complete).length;
  }

  get items(): readonly TodoItem[] {
    return this.list.items.filter(item => this.showComplete || !item.complete);
  }

  addItem(newItem: string) {
    if (newItem != "") {
      this.list.addItem(newItem);
    }
  }

  showComplete: boolean = false;
}

```

Additional CSS styles are required to lay out the toggle switch, as shown in Listing 2-26.

Listing 2-26. Adding Styles in the app.component.css File in the src/app Folder

```

.spacer { flex: 1 1 auto }
.tableContainer { padding: 15px }
.fullWidth { width: 100% }
.inputContainer { margin: 15px 15px 5px }
.addButton { margin: 5px }
.toggleContainer { margin: 15px; display: flex }

```

The result is that the user can decide whether to see completed tasks, as shown in Figure 2-12.

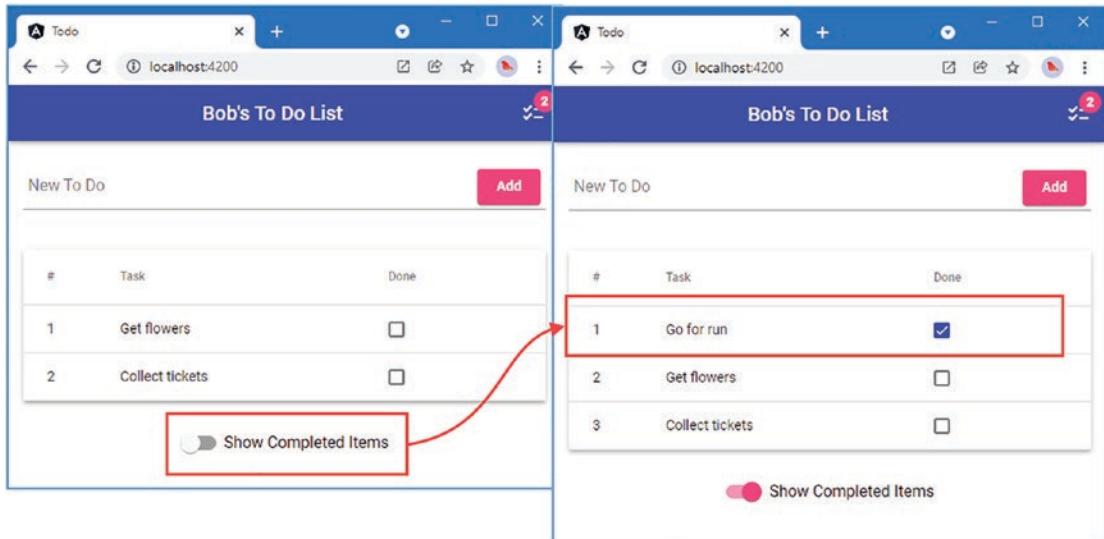


Figure 2-12. Showing completed tasks

Summary

In this chapter, I showed you how to create your first simple Angular app, which lets the user create new to-do items and mark existing items as complete. Don't worry if not everything in this chapter makes sense. What's important to understand at this stage is the general shape of an Angular application, which is built around a data model, components, and templates. If you keep these three key building blocks in mind and remember that the result is standard HTML elements, then you will have a solid foundation for everything that follows. In the next chapter, I put Angular in context and describe the structure of this book.

CHAPTER 3



Primer, Part 1

Developers come to the world of web app development via many paths and are not always grounded in the basic technologies that web apps rely on. In this chapter, I provide a brief overview of HTML, introduce the basics of JavaScript and TypeScript, and give you the foundation you need to understand the examples in the rest of the book, continuing with more advanced features in Chapter 4. If you are already familiar with HTML and TypeScript, you can jump right to Chapter 5, where I use Angular to create a more complex and realistic application.

Preparing the Example Project

To create the example project for this chapter, open a new command prompt, navigate to a convenient location, and run the command shown in Listing 3-1.

Tip You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/Apress/pro-angular-5ed>. See Chapter 1 for how to get help if you have problems running the examples.

Listing 3-1. Creating the Example Project

```
ng new Primer --routing false --style css --skip-git --skip-tests
```

This command creates a project called `Primer` that is set up for Angular development. I don't do any Angular development in this chapter, but I am going to use the Angular development tools as a convenient way to demonstrate different HTML, JavaScript, and TypeScript features.

Next, run the command shown in Listing 3-2 in the `Primer` folder to add the Bootstrap CSS package to the project. This is the package that I use to manage the appearance of content throughout this book.

Listing 3-2. Installing the Bootstrap CSS Package

```
npm install bootstrap@5.1.3
```

Run the command shown in Listing 3-3 to integrate Bootstrap into the application, taking care to enter the command as it is shown, without any extra spaces or quotes.

Listing 3-3. Changing the Application Configuration

```
ng config projects.example.architect.build.options.styles `  
['"src/styles.css"',  
'"node_modules/bootstrap/dist/css/bootstrap.min.css"]'
```

If you are using Windows, then use a PowerShell prompt to run the command shown in Listing 3-4 in the example folder.

Listing 3-4. Changing the Application Configuration Using PowerShell

```
ng config projects.Primer.architect.build.options.styles `  
['"src/styles.css"',  
'"node_modules/bootstrap/dist/css/bootstrap.min.css"]'
```

Run the command shown in Listing 3-5 in the Primer folder to start the Angular development compiler and HTTP server.

Listing 3-5. Starting the Development Tools

```
ng serve --open
```

After an initial build process, the Angular tools will open a browser window, which displays placeholder content added to the project when it was created, as shown in Figure 3-1.

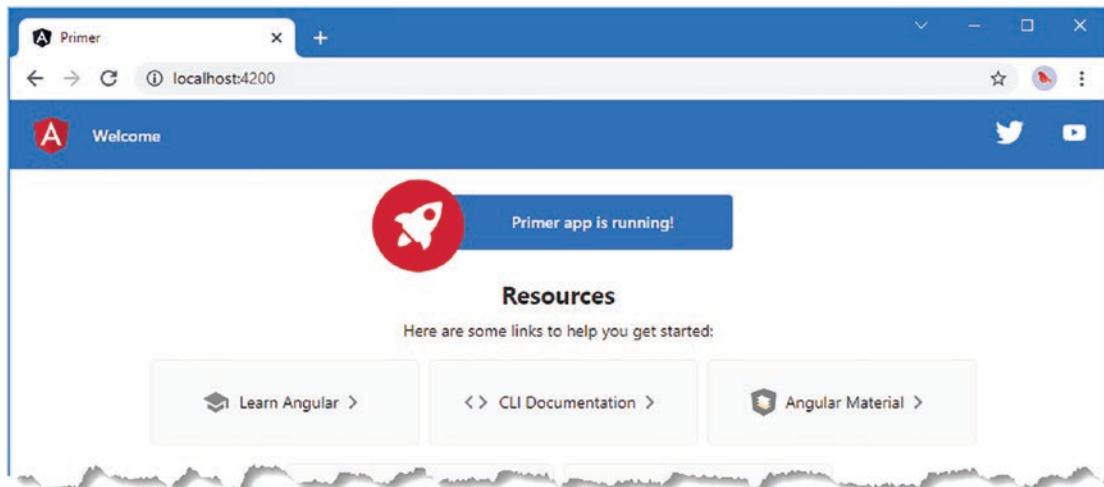


Figure 3-1. Running the example application

Understanding HTML

Use your code editor to open the Primer folder and replace the contents of `index.html` in the `src` folder with the content shown in Listing 3-6.

Listing 3-6. Replacing the Contents of the `index.html` File in the `src` Folder

```
<!DOCTYPE html>
<html>
<head>
    <title>ToDo</title>
    <meta charset="utf-8" />
</head>
<body class="m-1">
    <h3 class="bg-primary text-white p-3">Adam's To Do List</h3>
    <div class="my-1">
        <input class="form-control" />
        <button class="btn btn-primary mt-1">Add</button>
    </div>
    <table class="table table-striped table-bordered">
        <thead>
            <tr>
                <th>Description</th>
                <th>Done</th>
            </tr>
        </thead>
        <tbody>
            <tr><td>Buy Flowers</td><td>No</td></tr>
            <tr><td>Get Shoes</td><td>No</td></tr>
            <tr><td>Collect Tickets</td><td>Yes</td></tr>
            <tr><td>Call Joe</td><td>No</td></tr>
        </tbody>
    </table>
</body>
</html>
```

Reload the browser and you will see the content shown in Figure 3-2. You will see some errors in the browser's JavaScript console if you have it open, but these can be ignored and will be resolved later in the chapter.

The screenshot shows a web browser window titled "ToDo" at "localhost:4200". The page title is "Adam's To Do List". There is a blue "Add" button. Below it is a table with two columns: "Description" and "Done". The table contains four rows:

Description	Done
Buy Flowers	No
Get Shoes	No
Collect Tickets	Yes
Call Joe	No

Figure 3-2. Understanding HTML

At the heart of HTML is the *element*, which tells the browser what kind of content each part of an HTML document represents. Here is an element from the example HTML document:

```
...
<td>Buy Flowers</td>
...
```

As illustrated in Figure 3-3, this element has three parts: the start tag, the end tag, and the content.

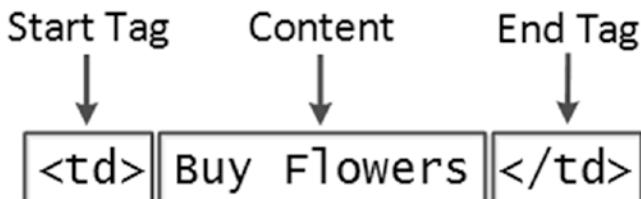


Figure 3-3. The anatomy of a simple HTML element

The *name* of this element (also referred to as the *tag name* or just the *tag*) is *td*, and it tells the browser that the content between the tags should be treated as a table cell. You start an element by placing the tag name in angle brackets (the < and > characters) and end an element by similarly using the tag, except that you also add a / character after the left-angle bracket (<). Whatever appears between the tags is the element's content, which can be text (such as *Buy Flowers* in this case) or other HTML elements.

Understanding Void Elements

The HTML specification includes elements that are not permitted to contain content. These are called *void* or *self-closing* elements, and they are written without a separate end tag, like this:

```
...
<input />
...
```

A void element is defined in a single tag, and you add a / character before the last angle bracket (the > character). The `input` element is the most used void element, and its purpose is to allow the user to provide input, through a text field, radio button, or checkbox. You will see lots of examples of working with this element in later chapters.

Understanding Attributes

You can provide additional information to the browser by adding *attributes* to your elements. Here is an element with an attribute from the example document:

```
...
<meta charset="utf-8" />
...
```

This is a `meta` element, and it describes the HTML document. There is one attribute, which I have emphasized so it is easier to see. Attributes are always defined as part of the start tag, and these attributes have a *name* and a *value*.

The name of the attribute in this example is `charset`. For the `meta` element, the `charset` attribute specifies the character encoding, which is UTF-8 in this case.

Applying Attributes Without Values

Not all attributes are applied with a value; just adding them to an element tells the browser that you want a certain kind of behavior. Here is an example of an element with such an attribute (not from the example document; I just made up this example element):

```
...
<input class="form-control" required />
...
```

This element has two attributes. The first is `class`, which is assigned a value just like the previous example. The other attribute is just the word `required`. This is an example of an attribute that doesn't need a value.

Quoting Literal Values in Attributes

Angular relies on HTML element attributes to apply a lot of its functionality. Most of the time, the values of attributes are evaluated as JavaScript expressions, such as with this element, taken from Chapter 2:

```
...
<td [ngSwitch]="item.complete">
...
```

The attribute applied to the `td` element tells Angular to read the value of a property called `complete` on an object that has been assigned to a variable called `item`. There will be occasions when you need to provide a specific value rather than have Angular read a value from the data model, and this requires additional quoting to tell Angular that it is dealing with a literal value, like this:

```
...
<td [ngSwitch]="'Apples'">
...
```

The attribute value contains the string `Apples`, which is quoted in both single and double quotes. When Angular evaluates the attribute value, it will see the single quotes and process the value as a literal string.

Understanding Element Content

Elements can contain text, but they can also contain other elements, like this:

```
...
<thead>
  <tr>
    <th>Description</th>
    <th>Done</th>
  </tr>
</thead>
...
```

The elements in an HTML document form a hierarchy. The `html` element contains the `body` element, which contains content elements, each of which can contain other elements, and so on. In the listing, the `thead` element contains `tr` elements that, in turn, contain `th` elements. Arranging elements is a key concept in HTML because it imparts the significance of the outer element to those contained within.

Understanding the Document Structure

There are some key elements that define the basic structure of an HTML document: the DOCTYPE, `html`, `head`, and `body` elements. Here is the relationship between these elements with the rest of the content removed:

```
<!DOCTYPE html>
<html>
<head>
  ...head content...
</head>
```

```
<body>
  ...body content...
</body>
</html>
```

Each of these elements has a specific role to play in an HTML document. The DOCTYPE element tells the browser that this is an HTML document and, more specifically, that this is an *HTML5* document. Earlier versions of HTML required additional information. For example, here is the DOCTYPE element for an HTML4 document:

```
...
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
  "http://www.w3.org/TR/html4/strict.dtd">
...
```

The `html` element denotes the region of the document that contains the HTML content. This element always contains the other two key structural elements: `head` and `body`. As I explained at the start of the chapter, I am not going to cover the individual HTML elements. There are too many of them, and describing HTML5 completely took me more than 1,000 pages in my HTML book. That said, Table 3-1 provides brief descriptions of the elements I used in the `index.html` file in Listing 3-2 to help you understand how elements tell the browser what kind of content they represent.

UNDERSTANDING THE DOCUMENT OBJECT MODEL

When the browser loads and processes an HTML document, it creates the *Document Object Model* (DOM). The DOM is a model in which JavaScript objects are used to represent each element in the document, and the DOM is the mechanism by which you can programmatically engage with the content of an HTML document.

You rarely work directly with the DOM in Angular, but it is important to understand that the browser maintains a live model of the HTML document represented by JavaScript objects. When Angular modifies these objects, the browser updates the content it displays to reflect the modifications. This is one of the key foundations of web applications. If we were not able to modify the DOM, we would not be able to create client-side web apps.

Table 3-1. HTML Elements Used in the Example Document

Element	Description
DOCTYPE	Indicates the type of content in the document
body	Denotes the region of the document that contains content elements
button	Denotes a button; often used to submit a form to the server
div	A generic element; often used to add structure to a document for presentation purposes
h3	Denotes a header
head	Denotes the region of the document that contains metadata
html	Denotes the region of the document that contains HTML (which is usually the entire document)
input	Denotes a field used to gather a single data item from the user
link	Imports content into the HTML document
meta	Provides descriptive data about the document, such as the character encoding
table	Denotes a table, used to organize content into rows and columns
tbody	Denotes the body of the table (as opposed to the header or footer)
td	Denotes a content cell in a table row
th	Denotes a header cell in a table row
thead	Denotes the header of a table
title	Denotes the title of the document; used by the browser to set the title of the window or tab
tr	Denotes a row in a table

Understanding CSS and the Bootstrap Framework

HTML elements tell the browser what kind of content they represent, but they don't provide any information about how that content should be displayed. The information about how to display elements is provided using *Cascading Style Sheets* (CSS). CSS consists of *properties* that can be used to configure every aspect of an element's appearance and *selectors* that allow those properties to be applied.

CSS is flexible and powerful, but it requires time and close attention to detail to get good, consistent results, especially as some legacy browsers implement features inconsistently. CSS frameworks provide a set of styles that can be easily applied to produce consistent effects throughout a project.

Throughout this book, I use the Bootstrap CSS framework, which consists of CSS classes that can be applied to elements to style them consistently, and JavaScript code that performs additional enhancements. I use the Bootstrap CSS styles in this book because they let me style my examples without having to define custom styles in each chapter. I don't use the Bootstrap JavaScript features at all in this book since the interactive parts of the examples are provided using Angular.

I don't go into detail about Bootstrap because it isn't the topic of this book, but you will see that many of the HTML elements used in examples throughout this book are assigned to classes like this:

```
...
<h3 class="bg-primary text-white p-3">Adam's To Do List</h3>
...
```

The `bg-primary`, `text-white`, and `p-3` classes all apply styles defined by the Bootstrap framework, setting the background color, text color, and padding, respectively. Unless noted in the description of an example, you can ignore the classes to which elements are assigned. See <https://getbootstrap.com> for details of the Bootstrap framework.

Understanding TypeScript/JavaScript

Angular applications are written in TypeScript, which is a superset of JavaScript that adds support for static types. In this section, I describe the relationship between TypeScript and JavaScript and introduce the basic features that you will need to understand to begin Angular development, continuing in Chapter 4. This is not a comprehensive guide to TypeScript or JavaScript, but it addresses the basics, and it will give you the knowledge you need to get started.

Understanding the TypeScript Workflow

Angular projects are set up with the TypeScript compiler, which is used to generate the JavaScript code that will be sent to the browser. There is no Angular development in this chapter, but I am going to take advantage of the TypeScript support to demonstrate important language features. The key file for this process is named `main.ts` and is found in the `src` folder. Replace the contents of the `main.ts` file with the statements shown in Listing 3-7.

Listing 3-7. Replacing the Contents of the main.ts File in the src Folder

```
console.log("Hello");
```

The basic JavaScript building block is the *statement*. Each statement represents a single command, and statements are usually terminated by a semicolon (`;`). The semicolon is optional, but using them makes your code easier to read and allows for multiple statements on a single line.

When you save the file, the change is detected, and the Angular tools rebuild the project, sending the results to the browser to execute. The statement in Listing 3-7 calls the `console.log` function, which writes a message to the browser's JavaScript console. Open your browser's F12 developer tools (which is typically done by pressing the F12 key) and select the Console; you will see the output shown in Figure 3-4.

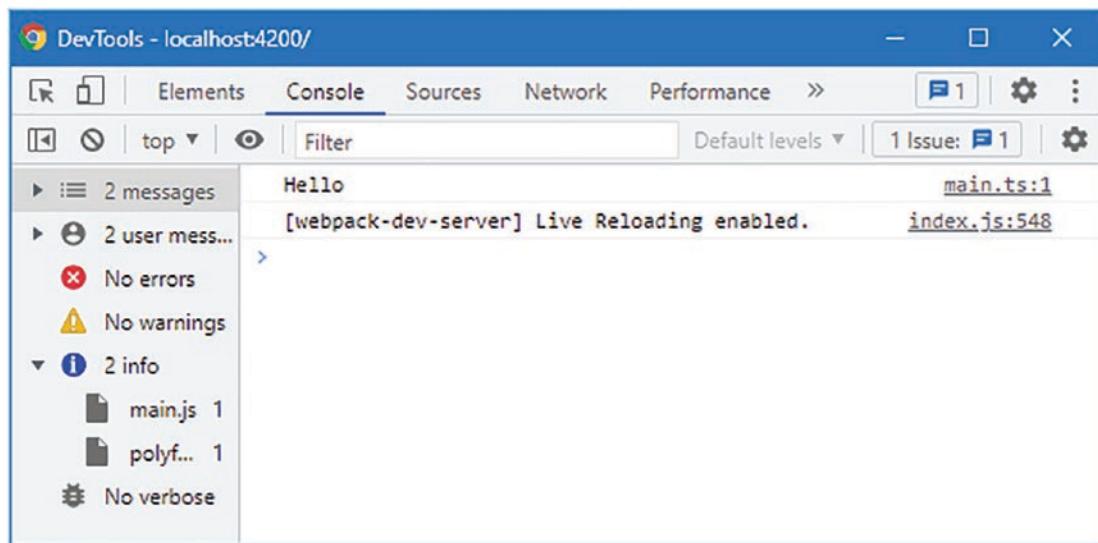


Figure 3-4. A message in the JavaScript console

The output from the statement in the `main.ts` file is displayed, along with additional messages generated by the automatic reloading process, which will automatically update the browser when a change is detected. Listing 3-8 adds another statement to the `main.ts` file.

Listing 3-8. Adding a Statement in the `main.ts` File in the `src` Folder

```
console.log("Hello");
console.log("Hello, World");
```

When you save the file, the project will be recompiled, and the browser will automatically reload, producing the following output in the JavaScript console:

```
Hello
Hello, World
```

Understanding JavaScript vs. TypeScript

JavaScript has an unusual approach to data types, which means that, for example, any variable can be assigned any value, regardless of type. As a simple demonstration, I am going to work outside of the Angular tools for a moment. Open a new command prompt, navigate to a convenient location, and create a file named `example.js` with the content shown in Listing 3-9. It doesn't matter where you put this file, as long as it isn't in the Primer project folder.

Listing 3-9. The Contents of the `example.js` File

```
function myFunction(param) {
  let result = param + 100;
  console.log("My result: " + result);
}
```

This listing defines a JavaScript function, which receives a value as a parameter, uses the addition operator to add 10 to the value, and then writes out the result to the JavaScript console. Notice that there are no data types specified in this code. The function, which is named `myFunction`, can receive any data type, as shown in Listing 3-10.

Listing 3-10. Invoking the Function in the example.js File

```
function myFunction(param) {
    let result = param + 100;
    console.log("My result: " + result);
}

myFunction(1);
myFunction("London");
```

The first new statement invokes `myFunction` with a number, 10. The second new statement invokes `myFunction` with a string, London. Using the command prompt, execute the JavaScript code by running the command shown in Listing 3-11 in the folder in which you created the `example.js` file.

Listing 3-11. Executing the JavaScript Code

```
node example.js
```

This command will produce the following output as the JavaScript statements are executed:

```
My result: 101
My result: London100
```

When the function received a number, the addition operator combined one number, 1, with another number, 100, and produced the result 101. But when the function received a string, the addition operator was asked to combine values with two different data types. It produced its result by converting the number 100 into a string and concatenating it with the parameter value to produce the result London100. JavaScript does provide the means to check whether a value is of a specific type, as shown in Listing 3-12.

Listing 3-12. Checking a Type in the example.js File

```
function myFunction(param) {
    if (typeof(param) == "number") {
        let result = param + 100;
        console.log("My result: " + result);
    } else {
        throw ("Expected a number: " + param)
    }
}

myFunction(1);
myFunction("London");
```

The `typeof` function is used to check that the parameter is a number value, and the `throw` keyword is used to create an error if it is not, which you can see by running the command in Listing 3-11 again, which produces the following output:

```
My result: 101
C:\javascript.js:6
    throw ("Expected a number: " + param)
    ^
Expected a number: London
(Use `node --trace-uncought ...` to show where the exception was thrown)
```

The behavior of the function has changed so that it only accepts numbers, but this change is enforced at runtime, and there is no way for a programmer calling the function to know what it expects without reading the source code.

Compiling the Function with TypeScript

TypeScript is a superset of JavaScript that requires types to be specified so they can be checked by a compiler. Returning to the Angular project, replace the contents of the `main.ts` file with those shown in Listing 3-13.

Listing 3-13. Replacing the Contents of the `main.ts` File in the `src` Folder

```
function myFunction(param) {
    if (typeof(param) == "number") {
        let result = param + 100;
        console.log("My result: " + result);
    } else {
        throw ("Expected a number: " + param)
    }
}

myFunction(1);
myFunction("London");
```

This is the same code that I used in the JavaScript file in the previous section. When the file is saved, the Angular development tools detect the change and rebuild the project, which includes using the TypeScript compiler to compile files with the `.ts` extension. The compiler reports the following error:

```
Error: src/main.ts:1:21 - error TS7006: Parameter 'param' implicitly has an 'any' type.
1 function myFunction(param) {  
~~~~~
```

TypeScript is a layer on top of JavaScript but doesn't change the way that JavaScript works. So, TypeScript functions are allowed to accept any data type because that is how JavaScript functions work. The difference is that TypeScript requires the developer to explicitly declare that is the behavior that is required, as shown in Listing 3-14.

Listing 3-14. Specifying the Function Parameter Type in the main.ts File in the src Folder

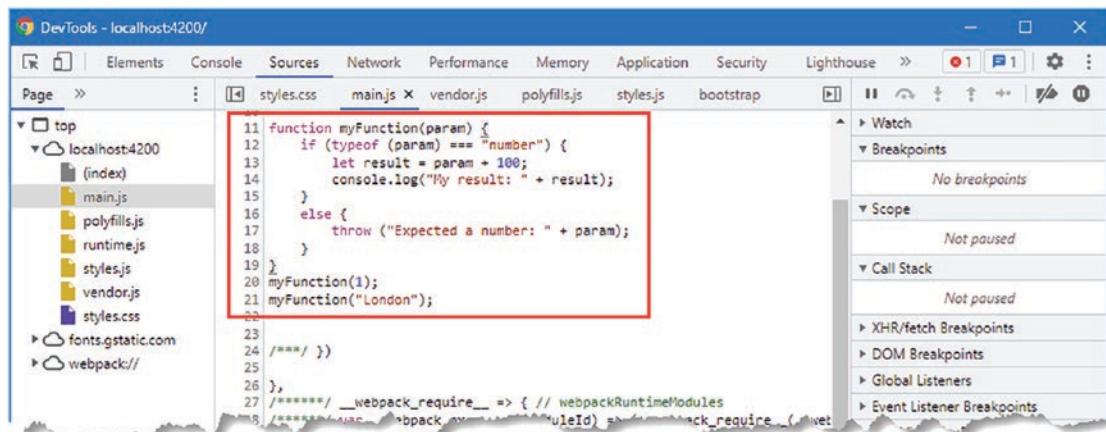
```
function myFunction(param: any) {
    if (typeof(param) == "number") {
        let result = param + 100;
        console.log("My result: " + result);
    } else {
        throw ("Expected a number: " + param)
    }
}

myFunction(1);
myFunction("London");
```

The type for the parameter is specified after the name, separated by a colon, which is known as a *type annotation*. The type specified in this listing is `any`, which indicates that the function can accept any data type. The behavior of the function hasn't changed, but the `any` keyword satisfies the TypeScript compiler. When the file is saved, the code will be compiled, sent to the browser, and executed, producing the following output in the browser's JavaScript console:

```
My result: 101
Uncaught Expected a number: London
```

TypeScript features are erased during the compilation process so that pure JavaScript remains. Most F12 developer tools allow you to inspect the JavaScript code received by the browser, which reveals that the compilation process has removed the `any` keyword, as shown in Figure 3-5.

**Figure 3-5.** Examining the compiled code

Using a More Specific Type

TypeScript requires the `any` keyword to make sure that you really want the default JavaScript behavior. Most of the time, however, TypeScript code is written with more specific data types, as shown in Listing 3-15.

Listing 3-15. Specifying a Single Type in the main.ts File in the src Folder

```
function myFunction(param: number) {
    if (typeof(param) == "number") {
        let result = param + 100;
        console.log("My result: " + result);
    } else {
        throw ("Expected a number: " + param)
    }
}

myFunction(1);
myFunction("London");
```

JavaScript defines five core primitive types: `string`, `number`, `boolean`, `undefined`, and `null`. In Listing 3-15, I have changed the type annotation to replace any with the JavaScript primitive type `number`, which tells TypeScript that the function only expects to receive `number` values.

Tip There are also `bignum` and `symbol` primitive types, but I don't use them in this book. The `bignum` type is used to represent numbers expressed in arbitrary precision format, and the `symbol` type is used to represent unique token values.

The TypeScript compiler will report the following error when the code is compiled:

```
Error: src/main.ts:11:12 - error TS2345: Argument of type 'string' is not assignable to
parameter of type 'number'.
11 myFunction("London");
~~~~~
```

The TypeScript compiler has inspected the types of the arguments used to invoke the function and determined that one of them doesn't match the `number` type annotation. The type annotation also allows me to simplify the function code, as shown in Listing 3-16, because I can rely on the TypeScript compiler to check types, rather than do so at runtime.

Listing 3-16. Simplifying the Function in the main.ts File in the src Folder

```
function myFunction(param: number) {
    //if (typeof(param) == "number") {
    //    let result = param + 100;
    //    console.log("My result: " + result);
    //}
    //else {
    //    throw ("Expected a number: " + param)
    //}
}

myFunction(1);
//myFunction("London");
```

When the file is saved and compiled, the following output will be displayed in the browser's JavaScript console:

```
My result: 101
```

Using a Type Union

TypeScript is a compile-time gatekeeper that helps you make your use of types explicit so that problems that would otherwise cause runtime errors can be detected. And, since TypeScript compiles into pure JavaScript and doesn't change the way that JavaScript works, everything that can be done in JavaScript can be described in TypeScript. This is important because many developers assume that TypeScript is similar to languages such as C# or Java. That's not the case—TypeScript is just a layer, albeit a useful one, that allows the programmer to annotate JavaScript code to explain to the compiler what types are expected in a given section of code so that the compiler can warn the programmer when different types are used.

As an example, earlier examples in this section have covered two extreme situations: that `myFunction` can accept all parameter types (denoted with the `any` keyword) and `myFunction` can accept only `number` parameters (denoted with the `number` type). But it is possible to write JavaScript functions so they can deal with combinations of types, as shown in Listing 3-17.

Listing 3-17. Supporting Multiple Types in the main.ts File in the src Folder

```
function myFunction(param: number) {
    if (typeof(param) == "number" || typeof(param) == "string") {
        let result = param + 100;
        console.log("My result: " + result);
    } else {
        throw ("Expected a number or a string: " + param)
    }
}

myFunction(1);
//myFunction("London");
```

There is now a mismatch between the code in the function and its parameter type annotation. To describe situations where multiple types are acceptable, TypeScript supports type unions, as shown in Listing 3-18.

Listing 3-18. Using a Type Union in the main.ts File in the src Folder

```
function myFunction(param: number | string) {
    if (typeof(param) == "number" || typeof(param) == "string") {
        let result = param + 100;
        console.log("My result: " + result);
    } else {
        throw ("Expected a number or a string: " + param)
    }
}

myFunction(1);
myFunction("London");
```

Type unions combine multiple types with the | character so that the type annotation `number | string` tells the compiler that `myFunction` will accept both `number` and `string` values. But the TypeScript compiler is clever, and it knows that JavaScript will do different things when it applies the addition operator to two `number` values or a `string` and a `number`, which means that this statement produces an ambiguous result:

```
...
let result = param + 100;
...
```

TypeScript is designed to avoid ambiguity, and the compiler will generate the following error when compiling the code:

```
...
Error: src/main.ts:3:22 - error TS2365: Operator '+' cannot be applied to types 'string | number' and 'number'.
...
```

Remember that the purpose of TypeScript is only to highlight potential problems, not to enforce any particular solution to a problem. There are several ways to resolve this ambiguity, but the one that I want to illustrate in this section is shown in Listing 3-19, which is to tell the TypeScript compiler that everything is going to be alright.

Listing 3-19. Addressing the Ambiguity in the `main.ts` File in the `src` Folder

```
function myFunction(param: number | string) {
    if (typeof(param) == "number" || typeof(param) == "string") {
        let result = (param as any) + 100;
        console.log("My result: " + result);
    } else {
        throw ("Expected a number or a string: " + param)
    }
}

myFunction(1);
myFunction("London");
```

The `as` keyword tells the TypeScript compiler that its knowledge of the `param` value is incomplete and that it should treat it as a type that I specify. In this case, I have specified the `any` type, which has the effect of telling the TypeScript that the ambiguity is expected and prevents it from producing an error. This code produces the following output in the browser's JavaScript console:

```
My result: 101
My result: London100
```

Using the `as` keyword should be done with caution because the TypeScript compiler is sophisticated and usually has a pretty solid understanding of how data types are being used. Equally, using the `any` type can be dangerous because it essentially stops the TypeScript compiler from checking types. And, it should go without saying, when you tell the TypeScript compiler that you know more about the code, then you need to make sure that you are right; otherwise, you will return to the runtime-error issue that led to the introduction of TypeScript in the first place.

Accessing Type Features

Unions are a useful way to describe combinations of types, but TypeScript will only allow the use of features that are shared by all of the types in the union. So, for example, for a value whose type is the union `number | string`, the TypeScript compiler will only allow the use of features that are defined by both the `number` and `string` types. To demonstrate, Listing 3-20 attempts to use the `toFixed` method, which is defined by the `number` type and which is not defined by the `string` type.

Listing 3-20. Accessing a Type Feature in the main.ts File in the src Folder

```
function myFunction(param: number | string) {
    if (typeof(param) == "number" || typeof(param) == "string") {
        let fixed = param.toFixed(2);
        console.log("My result: " + fixed);
    } else {
        throw ("Expected a number or a string: " + param)
    }
}

myFunction(1);
myFunction("London");
```

The TypeScript compiler is guarding against ambiguity again. It knows that the `param` value will be either a `number` or a `string` and that calling the `toFixed` method when the value is a `string` will cause an error. The compiler produces the following error when the code is compiled:

```
Error: src/main.ts:3:27 - error TS2339: Property 'toFixed' does not exist on type 'string | number'.
```

To resolve this issue, either I can use only features that are available for both `number` and `string` values or I can check the type `param` value within the function to eliminate the ambiguity, as shown in Listing 3-21.

Listing 3-21. Checking a Type in the main.ts File in the src Folder

```
function myFunction(param: number | string) {
    if (typeof(param) == "number") {
        let numberResult = param.toFixed(2);
        console.log("My result: " + numberResult);
    } else {
        let stringResult = param + 100;
        console.log("My result: " + stringResult);
    }
}

myFunction(1);
myFunction("London");
```

I need to remove the ambiguity about the parameter value's type so that I call the `toFixed` method only when the function receives a `number`. This code produces the following output in the browser's JavaScript console when it is compiled and executed:

```
My result: 1.00
My result: London100
```

Understanding the Basic TypeScript/JavaScript Features

Now that you understand the relationship between TypeScript and JavaScript, it is time to describe the basic language features you will need to follow the examples in this book. This is not a comprehensive guide to either TypeScript or JavaScript, but it should be enough to get you started as you learn how the features provided by Angular fit together.

Defining Variables and Constants

The `let` keyword is used to define variables, and the `const` keyword is used to define a constant value that will not change, as shown in Listing 3-22.

Listing 3-22. Defining Variables and Constants in the main.ts File in the src Folder

```
let condition = true;
let person = "Bob";
const age = 40;
```

The TypeScript compiler infers the type of each variable or constant from the value it is assigned and will generate an error if a value of a different type is assigned. Types can be specified explicitly, as shown in Listing 3-23.

Listing 3-23. Specifying Types in the main.ts File in the src Folder

```
let condition: boolean = true;
let person: string = "Bob";
const age: number = 40;
```

Dealing with Unassigned and Null Values

In JavaScript, variables that have been defined but not assigned a value are assigned the special value `undefined`, whose type is `undefined`, as shown in Listing 3-24.

Listing 3-24. Defining a Variable Without a Value in the main.ts File in the src Folder

```
let condition: boolean = true;
let person: string = "Bob";
const age: number = 40;
```

```
let place;
console.log("Place value: " + place + " Type: " + typeof(place));
place = "London";
console.log("Place value: " + place + " Type: " + typeof(place));
```

This code produces the following output in the browser's JavaScript console:

```
Place value: undefined Type: undefined
Place value: London Type: string
```

This behavior may seem nonsensical in isolation, but it is consistent with the rest of JavaScript, where values have types, and any value can be assigned to a variable. JavaScript also defines a separate special value, `null`, which can be assigned to variables to indicate no value or result, as shown in Listing 3-25.

Listing 3-25. Assigning Null in the main.ts File in the src Folder

```
let condition: boolean = true;
let person: string = "Bob";
const age: number = 40;

let place;
console.log("Place value: " + place + " Type: " + typeof(place));
place = "London";
console.log("Place value: " + place + " Type: " + typeof(place));
place = null;
console.log("Place value: " + place + " Type: " + typeof(place));
```

I can generally provide a robust defense of the way that JavaScript features work, but there is an oddity of the `null` value that makes little sense, which can be seen in the output this code produces in the browser's JavaScript console:

```
Place value: undefined Type: undefined
Place value: London Type: string
Place value: null Type: object
```

The oddity is that the type of the special `null` value is `object`. I introduce the JavaScript support for objects in Chapter 4, but this JavaScript quirk dates back to the first version of JavaScript and hasn't been addressed because so much code has been written that depends on it.

Leaving aside this inconsistency, when the TypeScript compiler processes the code in Listing 3-25, it determines that values of different types are assigned to the `place` variable and infers the variable's type as `any`.

As I explained, the `any` type allows values of any type to be used, which effectively disables the TypeScript compiler's type checks. A type union can be used to restrict the values that can be used, while still allowing `undefined` and `null` to be used, as shown in Listing 3-26.

Listing 3-26. Using a Type Union in the main.ts File in the src Folder

```
let condition: boolean = true;
let person: string = "Bob";
const age: number = 40;

let place: string | undefined | null;
console.log("Place value: " + place + " Type: " + typeof(place));
place = "London";
console.log("Place value: " + place + " Type: " + typeof(place));
place = null;
console.log("Place value: " + place + " Type: " + typeof(place));
```

This type union allows the place variable to be assigned string values or undefined or null. Notice that null is specified by value in the type union. This listing produces the same output in the JavaScript console as Listing 3-26.

Using the JavaScript Primitive Types

As noted earlier, JavaScript defines a small set of primitive types: string, number, boolean, undefined, and null. This may seem like a short list, but JavaScript manages to fit a lot of flexibility into these three types.

Working with Booleans

The boolean type has two values: true and false. Listing 3-27 shows both values being used, but this type is most useful when used in conditional statements, such as an if statement. There is no console output from this listing.

Listing 3-27. Defining boolean Values in the main.ts File in the src Folder

```
let firstBool = true;
let secondBool = false;
```

Working with Strings

You define string values using either the double or single quote characters, as shown in Listing 3-28.

Listing 3-28. Defining string Variables in the main.ts File in the src Folder

```
let firstString = "This is a string";
let secondString = 'And so is this';
```

The quote characters you use must match. You can't start a string with a single quote and finish with a double quote, for example. There is no output from this listing.

JavaScript provides string objects with a basic set of properties and methods, the most useful of which are described in Table 3-2.

Table 3-2. Useful string Properties and Methods

Name	Description
length	This property returns the number of characters in the string.
charAt(index)	This method returns a string containing the character at the specified index.
concat(string)	This method returns a new string that concatenates the string on which the method is called and the string provided as an argument.
indexOf(term, start)	This method returns the first index at which term appears in the string or -1 if there is no match. The optional start argument specifies the start index for the search.
replace(term, newTerm)	This method returns a new string in which all instances of term are replaced with newTerm.
slice(start, end)	This method returns a substring containing the characters between the start and end indices.
split(term)	This method splits up a string into an array of values that were separated by term.
toUpperCase()	These methods return new strings in which all the characters are uppercase or lowercase.
toLowerCase()	
trim()	This method returns a new string from which all the leading and trailing whitespace characters have been removed.

Using Template Strings

A common programming task is to combine static content with data values to produce a string that can be presented to the user. The traditional way to do this is through string concatenation, which is the approach I have been using in the examples so far in this chapter, as follows:

```
...
console.log("Place value: " + place + " Type: " + typeof(place));
...
```

JavaScript also supports *template strings*, which allow data values to be specified inline, which can help reduce errors and result in a more natural development experience. Listing 3-29 shows the use of a template string.

Listing 3-29. Using a Template String in the main.ts File in the src Folder

```
let place: string | undefined | null;
console.log(`Place value: ${place} Type: ${typeof(place)}`);
```

Template strings begin and end with backticks (the ` character), and data values are denoted by curly braces preceded by a dollar sign. This string, for example, incorporates the value of the place variable and its type into the template string:

```
...
console.log(`Place value: ${place} Type: ${typeof(place)}`);
...
```

This example produces the following output:

```
Place value: undefined Type: undefined
```

Working with Numbers

The number type is used to represent both *integer* and *floating-point* numbers (also known as *real numbers*). Listing 3-30 provides a demonstration.

Listing 3-30. Defining number Values in the main.ts File in the src Folder

```
let daysInWeek = 7;
let pi = 3.14;
let hexValue = 0xFFFF;
```

You don't have to specify which kind of number you are using. You just express the value you require, and JavaScript will act accordingly. In the listing, I have defined an integer value, defined a floating-point value, and prefixed a value with 0x to denote a hexadecimal value. Listing 3-30 doesn't produce any output.

Working with Null and Undefined Values

The null and undefined values have no features, such as properties or methods, but the unusual approach taken by JavaScript means that you can only assign these values to variables whose type is a union that includes null or undefined, as shown in Listing 3-31.

Listing 3-31. Assigning Null and Undefined Values in the main.ts File in the src Folder

```
let person1 = "Alice";
let person2: string | undefined = "Bob";
```

The TypeScript compiler will infer the type of the person1 variable as `string` because that is the type of the value assigned to it. This variable cannot be assigned the `null` or `undefined` value.

The person2 variable is defined with a type annotation that specifies `string` or `undefined` values. This variable can be assigned `undefined` but not `null`, since `null` is not part of the type union.

Using the JavaScript Operators

JavaScript defines a largely standard set of operators. I've summarized the most useful in Table 3-3.

Table 3-3. Useful JavaScript Operators

Operator	Description
<code>++</code> , <code>--</code>	Pre- or post-increment and decrement
<code>+, -, *, /, %</code>	Addition, subtraction, multiplication, division, remainder
<code><, <=, >, >=</code>	Less than, less than or equal to, more than, more than or equal to
<code>==, !=</code>	Equality and inequality tests
<code>== ==, != !=</code>	Identity and nonidentity tests
<code>&&, </code>	Logical AND and OR
<code> , ??</code>	Null and null-ish coalescing operators
<code>?</code>	Optional chaining operator
<code>=</code>	Assignment
<code>+</code>	String concatenation
<code>?:</code>	Three-operand conditional statement

Using Conditional Statements

Many of the JavaScript operators are used in conjunction with conditional statements. In this book, I tend to use the `if/else` and `switch` statements. Listing 3-32 shows the use of both, which will be familiar if you have worked with pretty much any programming language.

Listing 3-32. Using the if/else and switch Conditional Statements in the main.ts File in the src Folder

```
let firstName = "Adam";

if (firstName == "Adam") {
    console.log("firstName is Adam");
} else if (firstName == "Jacqui") {
    console.log("firstName is Jacqui");
} else {
    console.log("firstName is neither Adam or Jacqui");
}

switch (firstName) {
    case "Adam":
        console.log("firstName is Adam");
        break;
    case "Jacqui":
        console.log("firstName is Jacqui");
        break;
    default:
        console.log("firstName is neither Adam or Jacqui");
        break;
}
```

The results from the listing are as follows:

```
firstName is Adam
firstName is Adam
```

The Equality Operator vs. the Identity Operator

In JavaScript, the equality operator (==) will attempt to coerce (convert) operands to the same type to assess equality. This can be a useful feature, but it is widely misunderstood and often leads to unexpected results. Listing 3-33 shows the equality operator in action.

Listing 3-33. Using the Equality Operator in the main.ts File in the src Folder

```
let firstVal: any = 5;
let secondVal: any = "5";

if (firstVal == secondVal) {
    console.log("They are the same");
} else {
    console.log("They are NOT the same");
}
```

The output from this script is as follows:

```
They are the same
```

JavaScript is converting the two operands into the same type and comparing them. In essence, the equality operator tests that values are the same irrespective of their type.

If you want to test to ensure that the values *and* the types are the same, then you need to use the identity operator (====, three equal signs, rather than the two of the equality operator), as shown in Listing 3-34.

Listing 3-34. Using the Identity Operator in the main.ts File in the src Folder

```
let firstVal: any = 5;
let secondVal: any = "5";

if (firstVal === secondVal) {
    console.log("They are the same");
} else {
    console.log("They are NOT the same");
}
```

In this example, the identity operator will consider the two variables to be different. This operator doesn't coerce types. The result from this script is as follows:

```
They are NOT the same
```

To demonstrate how JavaScript works, I had to use the `any` type when declaring the `firstVal` and `secondVal` variables, because TypeScript restricts the use of the equality operator so that it can be used only on two values of the same type. Listing 3-35 removes the variable type annotations and allows TypeScript to infer the types from the assigned values.

Listing 3-35. Removing the Type Annotations in the main.ts File in the src Folder

```
let firstVal = 5;
let secondVal = "5";

if (firstVal === secondVal) {
    console.log("They are the same");
} else {
    console.log("They are NOT the same");
}
```

The TypeScript compiler detects that the variable types are not the same and generates the following error:

```
Error: src/main.ts:4:5 - error TS2367: This condition will always return 'false' since the
types 'number' and 'string' have no overlap.
```

UNDERSTANDING TRUTHY AND FALSY

The comparison operator presents another pitfall for the unwary, which is that expressions can be *truthy* or *falsy*. The following results are always falsy:

- The `false` (`boolean`) value
- The `0` (`number`) value
- The empty string (`""`)
- `null`
- `undefined`
- `Nan` (a special number value)

All other values are truthy, which can be confusing. For example, `"false"` (a string whose content is the word `false`) is truthy. The best way to avoid confusion is to only use expressions that evaluate to the boolean values `true` and `false`.

Explicitly Converting Types

The string concatenation operator (`+`) has higher precedence than the addition operator (also `+`), which means JavaScript will concatenate variables in preference to adding. This can confuse because JavaScript will also convert types freely to produce a result—and not always the result that is expected, as shown in Listing 3-36.

Listing 3-36. String Concatenation Operator Precedence in the main.ts File in the src Folder

```
let myData1 = 5 + 5;
let myData2 = 5 + "5";

console.log(`Result 1: ${myData1}, Type: ${typeof(myData1)}`);
console.log(`Result 2: ${myData2}, Type: ${typeof(myData2)}`);
```

This code produces the following output in the browser's JavaScript console:

```
Result 1: 10, Type: number
Result 2: 55, Type: string
```

The second result is the kind that confuses. What might be intended to be an addition operation is interpreted as string concatenation through a combination of operator precedence and type conversion. The TypeScript compiler understands the way the JavaScript operators behave and correctly infers the data types it produces, but, unlike the equally confusing equality operator, TypeScript doesn't prevent the type conversion.

To avoid this, you can explicitly convert the types of values to ensure you perform the right kind of operation, as described in the following sections.

Converting Numbers to Strings

If you are working with multiple number variables and want to concatenate them as strings, then you can convert the numbers to strings with the `toString` method, as shown in Listing 3-37.

Listing 3-37. Using the number.`toString` Method in the main.ts File in the src Folder

```
let myData1 = (5).toString() + String(5);
let myData2 = 5 + "5";

console.log(`Result 1: ${myData1}, Type: ${typeof(myData1)}`);
console.log(`Result 2: ${myData2}, Type: ${typeof(myData2)}`);
```

Notice that I placed the numeric value in parentheses, and then I called the `toString` method. This is because you have to allow JavaScript to convert the literal value into a number before you can call the methods that the number type defines. I have also shown an alternative approach to achieve the same effect, which is to call the `String` function and pass in the numeric value as an argument. Both of these techniques have the same effect, which is to convert a number to a string, meaning that the `+` operator is used for string concatenation and not addition. The output from this script is as follows:

```
Result 1: 55, Type: string
Result 2: 55, Type: string
```

Other methods allow you to exert more control over how a number is represented as a string. I briefly describe these methods in Table 3-4. All of the methods shown in the table are defined by the `number` type.

Table 3-4. Useful Number-to-String Methods

Method	Description
<code>toString()</code>	This method returns a string that represents a number in base 10.
<code>toString(2)</code>	This method returns a string that represents a number in binary, octal, or hexadecimal notation.
<code>toString(8)</code>	
<code>toString(16)</code>	
<code>toFixed(n)</code>	This method returns a string representing a real number with the n digits after the decimal point.
<code>toExponential(n)</code>	This method returns a string that represents a number using exponential notation with one digit before the decimal point and n digits after.
<code>toPrecision(n)</code>	This method returns a string that represents a number with n significant digits, using exponential notation if required.

Converting Strings to Numbers

The complementary technique is to convert strings to numbers so that you can perform addition rather than concatenation, as shown in Listing 3-38.

Listing 3-38. Converting Strings to Numbers in the main.ts File in the src Folder

```
let myData1 = (5).toString() + String(5);
let myData2 = Number("5") + parseInt("5");

console.log(`Result 1: ${myData1}, Type: ${typeof(myData1)} );
console.log(`Result 2: ${myData2}, Type: ${typeof(myData2)} );
```

The output from this script is as follows:

```
Result 1: 55, Type: string
Result 2: 10, Type: number
```

The `Number` function is strict in the way that it parses string values, but there are two other functions you can use that are more flexible and will ignore trailing non-number characters. These functions are `parseInt` and `parseFloat`. I have described all three methods in Table 3-5.

Table 3-5. Useful String to Number Methods

Method	Description
<code>Number(str)</code>	This method parses the specified string to create an integer or real value.
<code>parseInt(str)</code>	This method parses the specified string to create an integer value.
<code>parseFloat(str)</code>	This method parses the specified string to create an integer or real value.

Using the Null and Nullish Coalescing Operators

The logical OR operator (`||`) has been traditionally used as a null coalescing operator in JavaScript, allowing a fallback value to be used in place of `null` or `undefined` values, as shown in Listing 3-39.

Note If you move the mouse pointer over the variable in code editors such as Visual Studio Code, you will see that the TypeScript compiler is smart enough to infer when the variables in the next few examples are `null` or `undefined`. This is because all of the statements in the `main.ts` file are executed in sequence, allowing the compiler to use a more specific combination of types than have been used in the type annotations. This doesn't happen in real projects, where code is defined in functions or methods.

Listing 3-39. Using the Null Coalescing Operator in the `main.ts` File in the `src` Folder

```
let val1: string | undefined;
let val2: string | undefined = "London";

let coalesced1 = val1 || "fallback value";
let coalesced2 = val2 || "fallback value";

console.log(`Result 1: ${coalesced1}`);
console.log(`Result 2: ${coalesced2}`);
```

The `||` operator returns the left-hand operand if it evaluates as truthy and otherwise returns the right-hand operand. When the operator is applied to `val1`, the right-hand operand is returned because no value has been assigned to the variable, meaning that it is `undefined`. When the operator is applied to `val2`, the left-hand operand is returned because the variable has been assigned the string `London`, which evaluates as truthy. This code produces the following output in the browser's JavaScript console:

```
Result 1: fallback value
Result 2: London
```

The problem with using the `||` operator this way is that truthy and falsy values can produce unexpected results, as shown in Listing 3-40.

Listing 3-40. An Unexpected Null Coalescing Result in the `main.ts` File in the `src` Folder

```
let val1: string | undefined;
let val2: string | undefined = "London";
let val3: number | undefined = 0;

let coalesced1 = val1 || "fallback value";
let coalesced2 = val2 || "fallback value";
let coalesced3 = val3 || 100;

console.log(`Result 1: ${coalesced1}`);
console.log(`Result 2: ${coalesced2}`);
console.log(`Result 3: ${coalesced3}`);
```

The new coalescing operation returns the fallback value, even though the `val3` variable is neither `null` nor `undefined`, because `0` evaluates as falsy. The code produces the following results in the browser's JavaScript console:

```
Result 1: fallback value
Result 2: London
Result 3: 100
```

The nullish-coalescing operator (`??`) addresses this issue by returning the right-hand operand only if the left-hand operand is `null` or `undefined`, as shown in Listing 3-41.

Listing 3-41. Using the Nullish-Coalescing Operator in the main.ts File in the src Folder

```
let val1: string | undefined;
let val2: string | undefined = "London";
let val3: number | undefined = 0;

let coalesced1 = val1 ?? "fallback value";
let coalesced2 = val2 ?? "fallback value";
let coalesced3 = val3 ?? 100;

console.log(`Result 1: ${coalesced1}`);
console.log(`Result 2: ${coalesced2}`);
console.log(`Result 3: ${coalesced3}`);
```

The nullish operator doesn't consider truthy and falsy outcomes and looks only for the `null` and `undefined` values. This code produces the following output in the browser's JavaScript console:

```
Result 1: fallback value
Result 2: London
Result 3: 0
```

Using the Optional Chaining Operator

As explainer earlier, TypeScript won't let `null` or `undefined` to be assigned to variables unless they have been defined with a suitable type union. Further, TypeScript will only allow methods and properties defined by all of the types in the union to be used. This combination of features means that you have to guard against `null` or `undefined` values before you can use the features provided by any other type in a union, as demonstrated in Listing 3-42.

Listing 3-42. Guarding Against Null or Undefined Values in the main.ts File in the src Folder

```
let count: number | undefined | null = 100;
if (count != null && count != undefined) {
    let result1: string = count.toFixed(2);
    console.log(`Result 1: ${result1}`);
}
```

To invoke the `toFixed` method, I have to make sure that the `count` variable hasn't been assigned `null` or `undefined`. The TypeScript compiler understands the meaning of the expressions in the `if` statement and knows that excluding `null` and `undefined` values means that the value assigned to `count` must be a number, meaning that the `toFixed` method can be used safely. This code produces the following output in the browser's JavaScript console:

Result 1: 100.00

The optional chaining operator (the `?` character) simplifies the guarding process, as shown in Listing 3-43.

Listing 3-43. Using the Optional Chaining Operator in the main.ts File in the src Folder

```
let count: number | undefined | null = 100;
if (count != null && count != undefined) {
    let result1: string = count.toFixed(2);
    console.log(`Result 1: ${result1}`);
}

let result2: string | undefined = count?.toFixed(2);
console.log(`Result 2: ${result2}`);
```

The operator is applied between the variable and the method call and will return `undefined` if the value is `null` or `undefined`, preventing the method from being invoked:

```
...
let result2: string | undefined = count?.toFixed(2);
...
```

If the value isn't `null` or `undefined`, then the method call will proceed as normal. The result from an expression that includes the optional chaining operator is a type union of `undefined` and the result from the method. In this case, the union will be `string | undefined` because the `toFixed` method returns a `string`. The code in Listing 3-43 produces the following output in the browser's JavaScript console:

Result 1: 100.00
Result 2: 100.00

Summary

In this chapter, I described some of the basic features of the foundation on which Angular is built. I described the basic structure of HTML elements and explained the relationship between JavaScript and TypeScript, before introducing the basic JavaScript/TypeScript features. In the next chapter, I continue to describe useful JavaScript and TypeScript features and provide a brief overview of an important JavaScript library that you will encounter in Angular development.

CHAPTER 4



Primer, Part 2

In this chapter, I continue to describe the basic features of TypeScript and JavaScript that are required for Angular development and briefly touch on the RxJS package, which is used extensively by Angular and is required for some advanced features.

Preparing for This Chapter

This chapter uses the Primer project created in Chapter 4. No changes are required for this chapter. Open a new command prompt, navigate to the `Primer` folder, and run the command shown in Listing 4-1 to start the Angular development tools.

Tip You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/Apress/pro-angular-5ed>. See Chapter 1 for how to get help if you have problems running the examples.

Listing 4-1. Starting the Development Tools

```
ng serve --open
```

After an initial build process, the Angular tools will open a browser window and display the content shown in Figure 4-1.

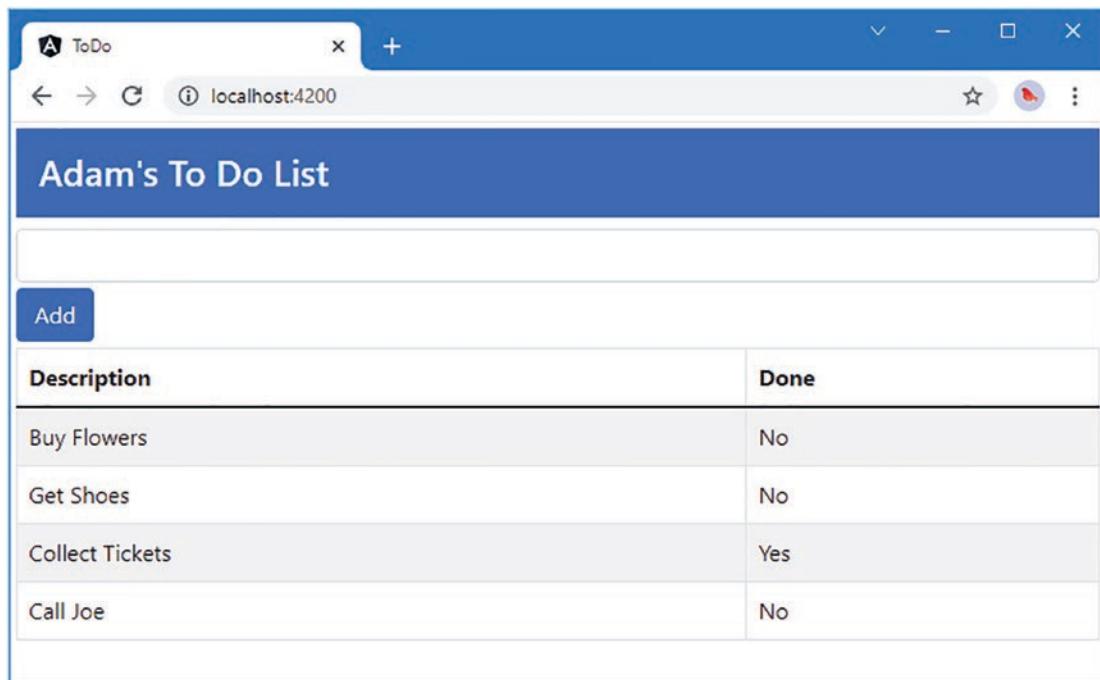


Figure 4-1. Running the example application

This chapter continues to use the browser’s JavaScript console. Press F12 to open the browser’s developers tools and switch to the console; you will see the following results (you may have to reload the browser):

```
Result 1: 100.00
Result 2: 100.00
```

Defining and Using Functions

When the browser receives JavaScript code, it executes the statements it contains in the order in which they have been defined. In common with most languages, JavaScript allows statements to be grouped into a function, which won’t be executed until a statement that invokes the function is executed, as shown in Listing 4-2.

Listing 4-2. Defining a Function in the main.ts File in the src Folder

```
function writeValue(val: string | null) {
    console.log(`Value: ${val ?? "Fallback value"}`)
}

writeValue("London");
writeValue(null);
```

Functions are defined with the `function` keyword and are given a name. If a function defines parameters, then TypeScript requires type annotations, which are used to enforce consistency in the use of the function. The function in Listing 4-2 is named `writeValue`, and it defines a parameter that will accept `string` or `null` values. The statement inside of the function isn't executed until the browser reaches a statement that invokes the function. The code in Listing 4-2 produces the following output in the browser's JavaScript console:

```
Value: London
Value: Fallback value
```

Defining Optional Function Parameters

By default, TypeScript will allow functions to be invoked only when the number of arguments matches the number of parameters the function defines. This may seem obvious if you are used to other mainstream languages, but a function can be called with any number of arguments in pure JavaScript, regardless of how many parameters have been defined. The `?` character is used to denote an optional parameter, as shown in Listing 4-3.

Listing 4-3. Defining an Optional Parameter in the main.ts File in the src Folder

```
function writeValue(val?: string) {
    console.log(`Value: ${val ?? "Fallback value"}`)
}

writeValue("London");
writeValue();
```

The `?` operator has been applied to the `val` parameter, which means that the function can be invoked with zero or one argument. Within the function, the parameter type is `string | undefined`, because the value will be `undefined` if the function is invoked without an argument.

Note Don't confuse `val?: string`, which is an optional parameter, with `val: string | undefined`, which is a type union of `string` and `undefined`. The type union requires the function to be invoked with an argument, which may be the value `undefined`, whereas the optional parameter allows the function to be invoked without an argument.

The code in Listing 4-3 produces the following output in the browser's JavaScript console:

```
Value: London
Value: Fallback value
```

Defining Default Parameter Values

Parameters can be defined with a default value, which will be used when the function is invoked without a corresponding argument. This can be a useful way to avoid dealing with `undefined` values, as shown in Listing 4-4.

Listing 4-4. Defining a Default Parameter Value in the main.ts File in the src Folder

```
function writeValue(val: string = "default value") {
    console.log(`Value: ${val}`)
}

writeValue("London");
writeValue();
```

The default value will be used when the function is invoked without an argument. This means that the type of the parameter in the example will always be `string`, so I don't have to check for `undefined` values. The code in Listing 4-4 produces the following output in the browser's JavaScript console:

```
Value: London
Value: default value
```

Defining Rest Parameters

Rest parameters are used to capture any additional arguments when a function is invoked with additional arguments, as shown in Listing 4-5.

Listing 4-5. Using a Rest Parameter in the main.ts File in the src Folder

```
function writeValue(val: string, ...extraInfo: string[]) {
    console.log(`Value: ${val}, Extras: ${extraInfo}`)
}

writeValue("London", "Raining", "Cold");
writeValue("Paris", "Sunny");
writeValue("New York");
```

The rest parameter must be the last parameter defined by the function, and its name is prefixed with an ellipsis (three periods, `...`). The rest parameter is an array to which any extra arguments will be assigned. In the listing, the function prints out each extra argument to the console, producing the following results:

```
Value: London, Extras: Raining,Cold
Value: Paris, Extras: Sunny
Value: New York, Extras:
```

Defining Functions That Return Results

You can return results from functions by declaring the return data type and using the `return` keyword within the function body, as shown in Listing 4-6.

Listing 4-6. Returning a Result in the main.ts File in the src Folder

```
function composeString(val: string) : string {
    return `Composed string: ${val}`;
}

function writeValue(val?: string) {
    console.log(composeString(val ?? "Fallback value"));
}

writeValue("London");
writeValue();
```

The new function defines one parameter, which is a `string`, and returns a result, which is also a `string`. The type of the result is defined using a type annotation after the parameters:

```
...
function composeString(val: string) : string {
```

TypeScript will check the use of the `return` keyword to ensure that the function returns a result and that the result is of the expected type. This code produces the following output in the browser's JavaScript console:

```
Composed string: London
Composed string: Fallback value
```

Using Functions as Arguments to Other Functions

JavaScript functions are values, which means you can use one function as the argument to another, as demonstrated in Listing 4-7.

Listing 4-7. Using a Function as an Argument to Another Function in the main.ts File in the src Folder

```
function getUKCapital() : string {
    return "London";
}

function writeCity(f: () => string) {
    console.log(`City: ${f()}`);
}

writeCity(getUKCapital);
```

The `writeCity` function defines a parameter called `f`, which is a function that it invokes to get the value to insert into the string that it writes out. TypeScript requires the function parameter to be described so that the types of its parameters and results are declared:

```
...
function writeCity(f: () => string) {
...
}
```

This is the arrow syntax, also known as fat arrow syntax or the lambda expression syntax. There are three parts to an arrow function: the input parameters surrounded by parentheses, then an equal sign and a greater-than sign (the “arrow”), and finally the function result. The parameter function doesn’t define any parameters, so the parentheses are empty. This means that the type of the parameter `f` is a function that accepts no parameters and returns a `string` result. The parameter function is invoked within a template string:

```
...
console.log(`City: ${f()}`)
...
```

Only functions with the specified combination of parameters and result can be used as an argument to `writeCity`. The `getUKCapital` function has the correct characteristics:

```
...
writeCity(getUKCapital);
...
```

Notice that only the name of the function is used as the argument. If you follow the function name with parentheses, `writeCity(getUKCapital())`, then you are telling JavaScript to invoke the `getUKCapital` function and pass the result to the `writeCity` function. TypeScript will detect that the result from the `getUKCapital` function doesn’t match the parameter type defined by the `writeCity` function and will produce an error when the code is compiled. The code in Listing 4-7 produces the following output in the browser’s JavaScript console:

City: London

Defining Functions Using the Arrow Syntax

The arrow syntax can also be used to define functions, not just to describe them, and this is a useful way to define functions inline, as shown in Listing 4-8.

Listing 4-8. Defining an Arrow Function in the main.ts File in the src Folder

```
function getUKCapital(): string {
    return "London";
}

function writeCity(f: () => string) {
    console.log(`City: ${f()}`)
}
```

```
writeCity(getUKCapital);
writeCity(() => "Paris");
```

This inline function receives no parameters and returns the literal string value Paris, allowing me to define a function that can be used as an argument to the writeCity function. The code in Listing 4-8 produces the following output in the browser's JavaScript console:

```
City: London
City: Paris
```

Understanding Value Closure

Functions can access values that are defined in the surrounding code, using a feature called *closure*, as demonstrated in Listing 4-9.

Listing 4-9. Using a Closure in the main.ts File in the src Folder

```
function getUKCapital(): string {
    return "London";
}

function writeCity(f: () => string) {
    console.log(`City: ${f()}`)
}

writeCity(getUKCapital);
writeCity(() => "Paris");
let myCity = "Rome";
writeCity(() => myCity);
```

The new arrow function returns the value of the variable named myCity, which is defined in the surrounding code. This is a powerful feature that means you don't have to define parameters on functions to pass around data values, but caution is required because it is easy to get unexpected results when using common variable names like counter or index, where you may not realize that you are reusing a variable name from the surrounding code. This example produces the following output in the browser's JavaScript console:

```
City: London
City: Paris
City: Rome
```

Working with Arrays

JavaScript arrays work like arrays in most other programming languages. Listing 4-10 demonstrates how to create and populate an array.

Listing 4-10. Creating an Populating an Array in the main.ts File in the src Folder

```
let myArray = [];
myArray[0] = 100;
myArray[1] = "Adam";
myArray[2] = true;
```

I have created a new and empty array using the literal syntax, which uses square brackets, and assigned the array to a variable named `myArray`. In the subsequent statements, I assign values to various index positions in the array. (There is no console output from this listing.)

There are a couple of things to note in this example. First, I didn't need to declare the number of items in the array when I created it. JavaScript arrays will resize themselves to hold any number of items. The second point is that I didn't have to declare the data types that the array will hold. Any JavaScript array can hold any mix of data types. In the example, I have assigned three items to the array: a number, a string, and a boolean. The TypeScript compiler infers the type of the array as `any[]`, denoting any array that can hold values of all types. The example can be written with the type annotation shown in Listing 4-11.

Listing 4-11. Using a Type Annotation in the main.ts File in the src Folder

```
let myArray: any[] = [];
myArray[0] = 100;
myArray[1] = "Adam";
myArray[2] = true;
```

Arrays can be restricted to hold values of specific types, as shown in Listing 4-12.

Listing 4-12. Restricting Array Value Types in the main.ts File in the src Folder

```
let myArray: (number | string | boolean)[] = [];
myArray[0] = 100;
myArray[1] = "Adam";
myArray[2] = true;
```

The type union restricts the array so that it can hold only `number`, `string`, and `boolean` values. Notice that I have put the type union in parentheses because the union `number | string | boolean[]` denotes a value that can be assigned a `number`, a `string`, or an array of `boolean` values, which is not what is intended.

Arrays can be defined and populated in a single statement, as shown in Listing 4-13.

Listing 4-13. Populating a New Array in the main.ts File in the src Folder

```
let myArray: (number | string | boolean)[] = [100, "Adam", true];
```

If you omit the type annotation, TypeScript will infer the array type from the values used to populate the array. You should rely on this feature with caution for arrays that are intended to hold multiple types because it requires that the full range of types is used when creating the array.

Reading and Modifying the Contents of an Array

You read the value at a given index using square braces ([and]), placing the index you require between the braces, as shown in Listing 4-14.

Listing 4-14. Reading the Data from an Array Index in the main.ts File in the src Folder

```
let myArray: (number | string | boolean)[] = [100, "Adam", true];

let val = myArray[0];
console.log(`Value: ${val}`);
```

The TypeScript compiler infers the type of values in the array so that the type of the `val` variable in Listing 4-14 is `number | string | boolean`. This code produces the following output in the browser's JavaScript console:

Value: 100

You can modify the data held in any position in a JavaScript array simply by assigning a new value to the index, as shown in Listing 4-15. The TypeScript compiler will check that the type of the value you assign matches the array element type.

Listing 4-15. Modifying the Contents of an Array in the main.ts File in the src Folder

```
let myArray: (number | string | boolean)[] = [100, "Adam", true];

myArray[0] = "Tuesday";

let val = myArray[0];
console.log(`Value: ${val}`);
```

In this example, I have assigned a `string` to position 0 in the array, a position that was previously held by a `number`. This code produces the following output in the browser's JavaScript console:

Value: Tuesday

Enumerating the Contents of an Array

You enumerate the content of an array using a `for` loop or using the `forEach` method, which receives a function that is called to process each element in the array. Listing 4-16 shows both approaches.

Listing 4-16. Enumerating the Contents of an Array in the main.ts File in the src Folder

```
let myArray: (number | string | boolean)[] = [100, "Adam", true];

for (let i = 0; i < myArray.length; i++) {
    console.log("Index " + i + ": " + myArray[i]);
}

console.log("----");

myArray.forEach((value, index) => console.log("Index " + index + ": " + value));
```

The JavaScript for loop works just the same way as loops in many other languages. You determine how many elements there are in the array using its `length` property.

The function passed to the `forEach` method is given two arguments: the value of the current item to be processed and the position of that item in the array. In this listing, I have used an arrow function as the argument to the `forEach` method, which is the kind of use for which they excel (and you will see used throughout this book). The output from the listing is as follows:

```
Index 0: 100
Index 1: Adam
Index 2: true
---
Index 0: 100
Index 1: Adam
Index 2: true
```

Using the Spread Operator

The spread operator is used to expand an array so that its contents can be used as function arguments or combined with other arrays. In Listing 4-17, I used the spread operator to expand an array so that its items can be combined into another array.

Listing 4-17. Using the Spread Operator in the main.ts File in the src Folder

```
let myArray: (number | string | boolean)[] = [100, "Adam", true];
let otherArray = [...myArray, 200, "Bob", false];

// for (let i = 0; i < myArray.length; i++) {
//   console.log("Index " + i + ": " + myArray[i]);
// }

// console.log("---");

otherArray.forEach((value, index) => console.log("Index " + index + ": " + value));
```

The spread operator is an ellipsis (a sequence of three periods), and it causes the array to be unpacked.

```
...
let otherArray = [...myArray, 200, "Bob", false];
...
```

Using the spread operator, I can specify `myArray` as an item when I define `otherArray`, with the result that the contents of the first array will be unpacked and added as items to the second array. This example produces the following results:

```
Index 0: 100
Index 1: Adam
Index 2: true
Index 3: 200
Index 4: Bob
Index 5: false
```

Using the Built-in Array Methods

JavaScript arrays define a number of methods, the most useful of which are described in Table 4-1.

Table 4-1. Useful Array Methods

Method	Description
<code>concat(otherArray)</code>	This method returns a new array that concatenates the array on which it has been called with the array specified as the argument. Multiple arrays can be specified.
<code>join(separator)</code>	This method joins all the elements in the array to form a string. The argument specifies the character used to delimit the items.
<code>pop()</code>	This method removes and returns the last item in the array.
<code>shift()</code>	This method removes and returns the first element in the array.
<code>push(item)</code>	This method appends the specified item to the end of the array.
<code>unshift(item)</code>	This method inserts a new item at the start of the array.
<code>reverse()</code>	This method returns a new array that contains the items in reverse order.
<code>slice(start,end)</code>	This method returns a section of the array.
<code>sort()</code>	This method sorts the array. An optional comparison function can be used to perform custom comparisons.
<code>splice(index, count)</code>	This method removes count items from the array, starting at the specified index. The removed items are returned as the result of the method.
<code>unshift(item)</code>	This method inserts a new item at the start of the array.
<code>every(test)</code>	This method calls the test function for each item in the array and returns true if the function returns true for all of them and false otherwise.
<code>some(test)</code>	This method returns true if calling the test function for each item in the array returns true at least once.
<code>filter(test)</code>	This method returns a new array containing the items for which the test function returns true.
<code>find(test)</code>	This method returns the first item in the array for which the test function returns true.
<code>findIndex(test)</code>	This method returns the index of the first item in the array for which the test function returns true.
<code>foreach(callback)</code>	This method invokes the callback function for each item in the array, as described in the previous section.
<code>includes(value)</code>	This method returns true if the array contains the specified value.
<code>map(callback)</code>	This method returns a new array containing the result of invoking the callback function for every item in the array.
<code>reduce(callback)</code>	This method returns the accumulated value produced by invoking the callback function for every item in the array.

Since many of the methods in Table 4-1 return a new array, these methods can be chained together to process a filtered data array, as shown in Listing 4-18.

Listing 4-18. Processing a Data Array in the main.ts File in the src Folder

```
let myArray: (number | string | boolean)[] = [100, "Adam", true];
let otherArray = [...myArray, 200, "Bob", false];

let sum: number = otherArray
  .filter(val => typeof(val) == "number")
  .reduce((total: number, val) => total + (val as number), 0)

console.log(`Sum: ${sum}`);
```

I use the filter method to select the number in the array and use the reduce method to determine the total, producing the following output in the browser's JavaScript console:

Sum: 300

Notice that I have had to give the TypeScript compiler some help with a type annotation and the as keyword. The compiler is sophisticated but doesn't always correctly infer the types in this type of operation.

Working with Objects

JavaScript objects are a collection of properties, each of which has a name and value. The simplest way to create an object is to use the literal syntax, as shown in Listing 4-19.

Listing 4-19. Creating an Object in the main.ts File in the src Folder

```
let hat = {
  name: "Hat",
  price: 100
};

let boots = {
  name: "Boots",
  price: 100
}

console.log(`Name: ${hat.name}, Price: ${hat.price}`);
console.log(`Name: ${boots.name}, Price: ${boots.price}`);
```

The literal syntax uses braces to contain a list of property names and values. Names are separated from their values with colons and from other properties with commas. Two objects are defined in Listing 4-19 and assigned to variables named hat and boots. The properties defined by the object can be accessed through the variable name, as shown in this statement:

```
...
console.log(`Name: ${hat.name}, Price: ${hat.price}`);
...
```

The code in Listing 4-19 produces the following output:

```
Name: Hat, Price: 100
Name: Boots, Price: 100
```

Understanding Literal Object Types

When the TypeScript encounters a literal object, it infers its type, using the combination of property names and the values to which they are assigned. This combination can be used in type annotations, allowing the shape of objects to be described as, for example, function parameters, as shown in Listing 4-20.

Tip In Angular development, you will most frequently create objects using classes, which are described in the “Defining Classes” section.

Listing 4-20. Describing an Object Type in the main.ts File in the src Folder

```
let hat = {
    name: "Hat",
    price: 100
};

let boots = {
    name: "Boots",
    price: 100
}

function printDetails(product : { name: string, price: number}) {
    console.log(`Name: ${product.name}, Price: ${product.price}`);
}

printDetails(hat);
printDetails(boots);
```

The type annotation specifies that the `product` parameter can accept objects that define a `string` property called `name`, and a `number` property named `price`. This example produces the same output as Listing 4-19.

A type annotation that describes a combination of property names and types just sets out a minimum threshold for objects, which can define additional properties and still be conform to the type, as shown in Listing 4-21.

Listing 4-21. Adding a Property in the main.ts File in the src Folder

```
let hat = {
    name: "Hat",
    price: 100
};
```

```

let boots = {
    name: "Boots",
    price: 100,
    category: "Snow Gear"
}

function printDetails(product : { name: string, price: number}) {
    console.log(`Name: ${product.name}, Price: ${product.price}`);
}

printDetails(hat);
printDetails(boots);

```

The listing adds a new property to the objects assigned to the `boots` variable, but since the object defines the properties described in the type annotation, this object can still be used as an argument to the `printDetails` function. This example produces the same output as Listing 4-19.

Defining Optional Properties in a Type Annotation

A question mark can be used to denote an optional property, as shown in Listing 4-22, allowing objects that don't define the property to still conform to the type.

Listing 4-22. Defining an Optional Property in the main.ts File in the src Folder

```

let hat = {
    name: "Hat",
    price: 100
};

let boots = {
    name: "Boots",
    price: 100,
    category: "Snow Gear"
}

function printDetails(product : { name: string, price: number, category?: string}) {
    if (product.category != undefined) {
        console.log(`Name: ${product.name}, Price: ${product.price}, ` +
            `Category: ${product.category}`);
    } else {
        console.log(`Name: ${product.name}, Price: ${product.price}`);
    }
}

printDetails(hat);
printDetails(boots);

```

The type annotation adds an optional `category` property, which is marked as optional. This means that the type of the property is `string | undefined`, and the function can test to see if a `category` value has been provided using the language features described in Chapter 3. This code produces the following output in the browser's JavaScript console:

Name: Hat, Price: 100
Boots, Price: 100, Category: Snow Gear

Defining Classes

Classes are templates used to create objects, providing an alternative to the literal syntax. Support for classes is a recent addition to the JavaScript specification and is intended to make working with JavaScript more consistent with other mainstream programming languages. Listing 4-23 defines a class and uses it to create objects.

Listing 4-23. Defining a Class in the main.ts File in the src Folder

```
class Product {

    constructor(name: string, price: number, category?: string) {
        this.name = name;
        this.price = price;
        this.category = category;
    }

    name: string
    price: number
    category?: string
}

let hat = new Product("Hat", 100);

let boots = new Product("Boots", 100, "Snow Gear");

function printDetails(product : { name: string, price: number, category?: string}) {
    if (product.category != undefined) {
        console.log(`Name: ${product.name}, Price: ${product.price}, ` +
            `Category: ${product.category}`);
    } else {
        console.log(`Name: ${product.name}, Price: ${product.price}`);
    }
}

printDetails(hat);
printDetails(boots);
```

JavaScript classes will be familiar if you have used another mainstream language such as Java or C#. The `class` keyword is used to declare a class, followed by the name of the class, which is `Product` in this example.

The `constructor` function is invoked when a new object is created using the class, and it provides an opportunity to receive data values and do any initial setup that the class requires. In the example, the `constructor` defines `name`, `price`, and `category` parameters that are used to assign values to properties defined with the same names.

The new keyword is used to create an object from a class, like this:

```
...
let hat = new Product("Hat", 100);
...
```

This statement creates a new object using the `Product` class as its template. `Product` is used as a function in this situation, and the arguments passed to it will be received by the constructor function defined by the class. The result of this expression is a new object that is assigned to a variable called `hat`.

Notice that the objects created from the class can still be used as arguments to the `printDetails` function. Introducing a class has changed the way that objects are created, but those objects have the same combination of property names and types and still match the type annotation for the function parameters. The code in Listing 4-23 produces the following output in the browser's JavaScript console:

```
Name: Hat, Price: 100
Name: Boots, Price: 100, Category: Snow Gear
```

Adding Methods to a Class

I can simplify the code in the example by moving the functionality defined by the `printDetails` function into a method defined by the `Product` class, as shown in Listing 4-24.

Listing 4-24. Defining a Method in the main.ts File in the src Folder

```
class Product {

    constructor(name: string, price: number, category?: string) {
        this.name = name;
        this.price = price;
        this.category = category;
    }

    name: string
    price: number
    category?: string

    printDetails() {
        if (this.category != undefined) {
            console.log(`Name: ${this.name}, Price: ${this.price}, ` +
                `Category: ${this.category}`);
        } else {
            console.log(`Name: ${this.name}, Price: ${this.price}`);
        }
    }
}
```

```

let hat = new Product("Hat", 100);

let boots = new Product("Boots", 100, "Snow Gear");

// function printDetails(product : { name: string, price: number, category?: string }) {
//   if (product.category != undefined) {
//     console.log(`Name: ${product.name}, Price: ${product.price}, ` +
//     `Category: ${product.category}`);
//   } else {
//     console.log(`Name: ${product.name}, Price: ${product.price}`);
//   }
// }

hat.printDetails();
boots.printDetails();

```

Methods are invoked through the object, like this:

```

...
hat.printDetails();
...

```

The method accesses the properties defined by the object through the `this` keyword:

```

...
console.log(`Name: ${this.name}, Price: ${this.price}`);
...

```

This example produces the following output in the browser's JavaScript console:

```

Name: Hat, Price: 100
Name: Boots, Price: 100, Category: Snow Gear

```

Access Controls and Simplified Constructors

TypeScript provides support for access controls using the `public`, `private`, and `protected` keywords. The `public` class gives unrestricted access to the properties and methods defined by a class, meaning they can be accessed by any other part of the application. The `private` keyword restricts access to features so they can be accessed only within the class that defines them. The `protected` keyword restricts access so that features can be accessed within the class or a subclass.

By default, the features defined by a class are accessible by any part of the application, as though the `public` keyword has been applied. You won't see the access control keywords applied to methods and properties in this book because access controls are not essential in an Angular application. But there is a related feature that I use often, which allows classes to be simplified by applying the access control keyword to the constructor parameters, as shown in Listing 4-25.

Listing 4-25. Simplifying the Class in the main.ts File in the src Folder

```
class Product {

    constructor(public name: string, public price: number, public category?: string) {
        // this.name = name;
        // this.price = price;
        // this.category = category;
    }

    // name: string
    // price: number
    // category?: string

    printDetails() {
        if (this.category != undefined) {
            console.log(`Name: ${this.name}, Price: ${this.price}, ` +
                `Category: ${this.category}`);
        } else {
            console.log(`Name: ${this.name}, Price: ${this.price}`);
        }
    }
}

let hat = new Product("Hat", 100);

let boots = new Product("Boots", 100, "Snow Gear");

hat.printDetails();
boots.printDetails();
```

Adding one of the access control keywords to a constructor parameter has the effect of creating a property with the same name, type, and access level. So, adding the `public` keyword to the `price` parameter, for example, creates a `public` property named `price`, which can be assigned `number` values. The value received through the constructor is used to initialize the property. This is a useful feature that eliminates the need to copy parameter values to initialize properties, and it is a feature that I wish other languages would adopt. The code in Listing 4-25 produces the same output as Listing 4-24, and only the way that the `name`, `price`, and `category` properties are defined has changed.

Using Class Inheritance

Classes can inherit behavior from other classes using the `extends` keyword, as shown in Listing Listing 4-26.

Listing 4-26. Using Class Inheritance in the main.ts File in the src Folder

```
class Product {

    constructor(public name: string, public price: number, public category?: string) {
```

```

printDetails() {
    if (this.category != undefined) {
        console.log(`Name: ${this.name}, Price: ${this.price}, ` +
            `Category: ${this.category}`);
    } else {
        console.log(`Name: ${this.name}, Price: ${this.price}`);
    }
}

class DiscountProduct extends Product {

    constructor(name: string, price: number, private discount: number) {
        super(name, price - discount);
    }
}

let hat = new DiscountProduct("Hat", 100, 10);

let boots = new Product("Boots", 100, "Snow Gear");

hat.printDetails();
boots.printDetails();

```

The `extends` keyword is used to declare the class that will be inherited from, known as the *superclass* or *base class*. In the listing, `DiscountProduct` inherits from `Product`. The `super` keyword is used to invoke the superclass's constructor and methods. The `DiscountProduct` builds on the `Product` functionality to add support for a price reduction, producing the following results in the browser's JavaScript console:

```
Name: Hat, Price: 90
Name: Boots, Price: 100, Category: Snow Gear
```

Checking Object Types

When applied to an object, the `typeof` function will return `object`. To determine whether an object has been derived from a class, the `instanceof` keyword can be used, as shown in Listing 4-27.

Listing 4-27. Checking an Object Type in the main.ts File in the src Folder

```

class Product {

    constructor(public name: string, public price: number, public category?: string) {}

    printDetails() {
        if (this.category != undefined) {
            console.log(`Name: ${this.name}, Price: ${this.price}, ` +
                `Category: ${this.category}`);
        }
    }
}

```

```

        } else {
            console.log(`Name: ${this.name}, Price: ${this.price}`);
        }
    }

class DiscountProduct extends Product {

    constructor(name: string, price: number, private discount: number) {
        super(name, price - discount);
    }
}

let hat = new DiscountProduct("Hat", 100, 10);

let boots = new Product("Boots", 100, "Snow Gear");

// hat.printDetails();
// boots.printDetails();

console.log(`Hat is a Product? ${hat instanceof Product}`);
console.log(`Hat is a DiscountProduct? ${hat instanceof DiscountProduct}`);
console.log(`Boots is a Product? ${boots instanceof Product}`);
console.log(`Boots is a DiscountProduct? ${boots instanceof DiscountProduct}`);

```

The `instanceof` keyword is used with an object value and a class, and the expression returns true if the object was created from the class or a superclass. The code in Listing 4-27 produces the following output in the browser's JavaScript console:

```

Hat is a Product? True
Hat is a DiscountProduct? True
Boots is a Product? True
Boots is a DiscountProduct? false

```

Working with JavaScript Modules

JavaScript modules are used to manage the dependencies in a web application, which means you don't need to manage a large set of individual code files to ensure that the browser downloads all the code for the application. Instead, during the compilation process, all of the JavaScript files that the application requires are combined into a larger file, known as a *bundle*, and it is this that is downloaded by the browser.

Creating and Using Modules

Each TypeScript or JavaScript file that you add to a project is treated as a module. To demonstrate, I created a folder called `modules` in the `src` folder, added to it a file called `NameAndWeather.ts`, and added the code shown in Listing 4-28.

Listing 4-28. The Contents of the NameAndWeather.ts File in the src/modules Folder

```
export class Name {
    constructor(public first: string, public second: string) {}

    get nameMessage() {
        return `Hello ${this.first} ${this.second}`;
    }
}

export class WeatherLocation {
    constructor(public weather: string, public city: string) {}

    get weatherMessage() {
        return `It is ${this.weather} in ${this.city}`;
    }
}
```

The classes, functions, and variables defined in a JavaScript or TypeScript file can be accessed only within that file by default. The `export` keyword is used to make features accessible outside of the file so that they can be used by other parts of the application. In the example, I have applied the `export` keyword to the `Name` and `WeatherLocation` classes, which means they are available to be used outside of the module.

Tip I have defined two classes in the `NameAndWeather.ts` file, which has the effect of creating a module that contains two classes. The convention in Angular applications is to put each class into its own file, which means that each class is defined in its own module.

The `import` keyword is used to declare a dependency on the features that a module provides. In Listing 4-29, I have used the `Name` and `WeatherLocation` classes in the `main.ts` file, and that means I have to use the `import` keyword to declare a dependency on them and the module they come from.

Listing 4-29. Importing Specific Types in the main.ts File in the src Folder

```
import { Name, WeatherLocation } from "./modules/NameAndWeather";

let name = new Name("Adam", "Freeman");
let loc = new WeatherLocation("raining", "London");

console.log(name.nameMessage);
console.log(loc.weatherMessage);
```

This is the way that I use the `import` keyword in most of the examples in this book. The keyword is followed by curly braces that contain a comma-separated list of the features that the code in the current file depends on, followed by the `from` keyword, followed by the module name. In this case, I have imported the `Name` and `WeatherLocation` classes from the `NameAndWeather` module in the `modules` folder. Notice that the file extension is not included when specifying the module.

When the `main.ts` file is compiled, the Angular development tools detect the dependency on the code in the `NameAndWeather.ts` file. This dependency ensures that the `Name` and `WeatherLocation` classes are included in the JavaScript bundle file, and you will see the following output in the browser's JavaScript console, showing that code in the module was used to produce the result:

```
Hello Adam Freeman  
It is raining in London
```

Notice that I didn't have to include the `NameAndWeather.ts` file in a list of files to be sent to the browser. Just using the `import` keyword is enough to declare the dependency and ensure that the code required by the application is included in the JavaScript file sent to the browser.

UNDERSTANDING MODULE RESOLUTION

You will see two different ways of specifying modules in the `import` statements in this book. The first is a relative module, in which the name of the module is prefixed with `./`, like this example from Listing 4-29:

```
...  
import { Name, WeatherLocation } from "./modules/NameAndWeather";  
...
```

This statement specifies a module located relative to the file that contains the `import` statement. In this case, the `NameAndWeather.ts` file is in the `modules` directory, which is in the same directory as the `main.ts` file. The other type of import is nonrelative. Here is an example of a nonrelative import from Chapter 2 and one that you will see throughout this book:

```
...  
import { Component } from "@angular/core";  
...
```

The module in this `import` statement doesn't start with `./`, and the build tools resolve the dependency by looking for a package in the `node_modules` folder. In this case, the dependency is on a feature provided by the `@angular/core` package, which is added to the project when it is created by the `ng new` command.

Working with Reactive Extensions

Angular relies on a package named RxJS, also known as *Reactive Extensions*, or *ReactiveX*. The Reactive Extensions library is useful in Angular applications because it provides a simple and unambiguous system for sending and receiving notifications. It doesn't sound like a huge achievement, but it underpins most of the built-in Angular functionality, which needs to change the HTML content displayed to the user when the state of the application changes.

You won't often need to work with RxJS directly, but there are some features described in this book that rely on RxJS, and a basic knowledge of the building blocks RxJS provides can be useful. (See <https://rxjs.dev> for a full description of the features provided by the RxJS package.)

Understanding Observables

The key Reactive Extensions building block is an `Observable<T>`, which represents an observable sequence of events that occur over a period of time. This is most often encountered when using the Angular support for making HTTP requests, described in Chapter 23, where the outcome of the request is presented through an `Observable<T>` object. The generic type argument `<T>` denotes the type of event that the observable produces so that an `Observable<string>` will produce a series of `string` values, for example.

An object can subscribe to an `Observable` and receive a notification each time an event occurs, allowing it to respond only when the event has been observed. In the case of an HTTP request, for example, the use of the `Observable` allows the response to be handled when it arrives, without the handler code needing to periodically check whether the request has completed.

Note If you are familiar with JavaScript, you may wonder if an `Observable` is the same as a `Promise`, which is the typical way of dealing with asynchronous operations. The key difference is that an `Observable` represents a series of events, rather than a single result, which better suits the way that Angular works. HTTP requests can be handled equally well with an `Observable` or a `Promise`, but other Angular features require ongoing notifications, which is where RxJS excels.

The basic method provided by an `Observable` is `subscribe`, which accepts an object whose properties are set to functions that respond to the sequence of events. The property names and the purpose of the functions are described in Table 4-2. If you only need to specify a function that receives events, then you can pass that function as the argument to the `subscribe` method.

Table 4-2. The `Observable` `subscribe` Argument Properties

Name	Description
<code>next</code>	This function is invoked when a new event occurs.
<code>error</code>	This function is invoked when an error occurs.
<code>complete</code>	This function is invoked when the sequence of events ends.

Listing 4-30 defines a function that receives an `Observable<string>` and writes out the sequence of `string` values that are received.

Listing 4-30. Using an `Observable` in the main.ts File in the src Folder

```
import { Observable } from "rxjs";

function receiveEvents(observable: Observable<string>) {
  observable.subscribe({
    next: str => {
      console.log(`Event received: ${str}`);
    },
    complete: () => console.log("Sequence ended")
  });
}
```

The RxJS package is added to the project when it is created. The `recieveEvents` function defines an `Observable<string>` parameter and calls the `subscribe` method, providing functions that write out messages when an event is received and when the event sequence ends.

Understanding Observers

The Reactive Extensions `Observer<T>` class provides the mechanism by which updates are created, using the methods described in Table 4-3.

Table 4-3. The Observer Methods

Name	Description
<code>next(value)</code>	This method creates a new event using the specified value.
<code>error(errorObject)</code>	This method reports an error, described using the argument, which can be any object.
<code>complete()</code>	This method ends the sequence, indicating that no further events will be sent.

Listing 4-31 defines a function that receives an `Observer<string>` and uses it to send a series of events before calling the `complete` method.

Listing 4-31. Using an Observer in the main.ts File in the src Folder

```
import { Observable, Observer } from "rxjs";

function recieveEvents(observable: Observable<string>) {
  observable.subscribe({
    next: str => {
      console.log(`Event received: ${str}`);
    },
    complete: () => console.log("Sequence ended")
  });
}

function sendEvents(observer: Observer<string>) {
  let count = 5;
  for (let i = 0; i < count; i++) {
    observer.next(`${i + 1} of ${count}`);
  }
  observer.complete();
}
```

Understanding Subjects

The Reactive Extensions library provides the `Subject<T>` class, which implements both the `Observer` and `Observable` functionality. A `Subject` is useful when you are working with RxJS in your own code, rather than using an `Observer` or `Observable` provided through the Angular API. In Listing 4-32, I have created a `Subject<string>` and used it as the argument to invoke the functions defined in the previous sections.

THE DIFFERENT TYPES OF SUBJECT

Listing 4-32 uses the `Subject` class, which is the simplest way to create an object that is both an `Observer` and an `Observable`. Its main limitation is that when a new subscriber is created using the `subscribe` method, it won't receive an event until the next time the `next` method is called. This can be unhelpful if you are creating instances of components or directives dynamically and you want them to have some context data as soon as they are created.

The Reactive Extensions library includes some specialized implementations of the `Subject` class that can be used to work around this problem. The `BehaviorSubject` class keeps track of the last event it processed and sends it to new subscribers as soon as they call the `subscribe` method. The `ReplaySubject` class does something similar, except that it keeps track of all of its events and sends them all to new subscribers, allowing them to catch up with any events that were sent before they subscribed.

Listing 4-32. Using a Subject in the main.ts File in the src Folder

```
import { Observable, Observer, Subject } from "rxjs";

function recieveEvents(observable: Observable<string>) {
    observable.subscribe({
        next: str => {
            console.log(`Event received: ${str}`);
        },
        complete: () => console.log("Sequence ended")
    });
}

function sendEvents(observer: Observer<string>) {
    let count = 5;
    for (let i = 0; i < count; i++) {
        observer.next(`${i + 1} of ${count}`);
    }
    observer.complete();
}

let subject = new Subject<string>();
recieveEvents(subject);
sendEvents(subject);
```

The new statements connect together the functions, and the `Subject<string>` acts as the conduit that carries the events between functions, producing the following output in the browser's JavaScript console:

```
Event received: 1 of 5
Event received: 2 of 5
Event received: 3 of 5
Event received: 4 of 5
Event received: 5 of 5
Sequence ended
```

Summary

In this chapter, I continued to describe the key features provided by TypeScript and JavaScript, including functions, arrays, objects, and modules. I also briefly described the RxJS package, which is used for some of the advanced features described later in the book.

CHAPTER 5



SportsStore: A Real Application

In Chapter 2, I built a quick and simple Angular application. Small and focused examples allow me to demonstrate specific Angular features, but they can lack context. To help overcome this problem, I am going to create a simple but realistic e-commerce application.

My application, called SportsStore, will follow the classic approach taken by online stores everywhere. I will create an online product catalog that customers can browse by category and page, a shopping cart where users can add and remove products, and a checkout where customers can enter their shipping details and place their orders. I will also create an administration area that includes create, read, update, and delete (CRUD) facilities for managing the catalog—and I will protect it so that only logged-in administrators can make changes. Finally, I show you how to prepare and deploy an Angular application.

My goal in this chapter and those that follow is to give you a sense of what real Angular development is like by creating as realistic an example as possible. I want to focus on Angular, of course, and so I have simplified the integration with external systems, such as the data store, and omitted others entirely, such as payment processing.

The SportsStore example is one that I use in a few of my books, not least because it demonstrates how different frameworks, languages, and development styles can be used to achieve the same result. You don't need to have read any of my other books to follow this chapter, but you will find the contrasts interesting if you already own my *Pro ASP.NET Core 3* book, for example.

The Angular features that I use in the SportsStore application are covered in-depth in later chapters. Rather than duplicate everything here, I tell you just enough to make sense of the example application and refer you to other chapters for in-depth information. You can either read the SportsStore chapters from end to end to get a sense of how Angular works or jump to and from the detailed chapters to get into the depth of each feature. Either way, don't expect to understand everything right away—Angular has lots of moving parts, and the SportsStore application is intended to show you how they fit together without diving too deeply into the details that I spend the rest of the book describing.

Preparing the Project

To create the SportsStore project, open a command prompt, navigate to a convenient location, and run the following command:

```
ng new SportsStore --routing false --style css --skip-git --skip-tests
```

The angular-cli package will create a new project for Angular development, with configuration files, placeholder content, and development tools. The project setup process can take some time since there are many NPM packages to download and install.

Tip You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/Apress/pro-angular-5ed>. See Chapter 1 for how to get help if you have problems running the examples.

Installing the Additional NPM Packages

Additional packages are required for the SportsStore project, in addition to the core Angular packages and build tools set up by the `ng new` command. Run the following commands to navigate to the `SportsStore` folder and add the required packages:

```
cd SportsStore
npm install bootstrap@5.1.3
npm install @fortawesome/fontawesome-free@6.0.0
npm install --save-dev json-server@0.17.0
npm install --save-dev jsonwebtoken@8.5.1
```

It is important to use the version numbers shown in the listing. You may see warnings about unmet peer dependencies as you add the packages, but you can ignore them. Some of the packages are installed using the `--save-dev` argument, which indicates they are used during development and will not be part of the `SportsStore` application.

Adding the CSS Style Sheets to the Application

Once the packages have been installed, run the command shown in Listing 5-1 in the `SportsStore` folder to add the Bootstrap CSS stylesheet to the project.

Listing 5-1. Changing the Application Configuration

```
ng config projects.SportsStore.architect.build.options.styles \
['"src/styles.css"', \
'"node_modules/@fortawesome/fontawesome-free/css/all.min.css"', \
'"node_modules/bootstrap/dist/css/bootstrap.min.css"]'
```

If you are using Windows, then use a PowerShell prompt to run the command shown in Listing 5-2 in the example folder.

Listing 5-2. Changing the Application Configuration Using PowerShell

```
ng config projects.SportsStore.architect.build.options.styles ^
['"src/styles.css"',
'"node_modules/@fortawesome/fontawesome-free/css/all.min.css"',
'"node_modules/bootstrap/dist/css/bootstrap.min.css"]'
```

Run the command shown in Listing 5-3 in the `SportsStore` folder to ensure the configuration changes have been applied correctly.

Listing 5-3. Checking the Configuration Changes

```
ng config projects.SportsStore.architect.build.options.styles
```

The output from this command should contain the three files listed in Listing 5-1 and Listing 5-2, like this:

```
[  
  "src/styles.css",  
  "node_modules/@fortawesome/fontawesome-free/css/all.min.css",  
  "node_modules/bootstrap/dist/css/bootstrap.min.css"  
]
```

Preparing the RESTful Web Service

The SportsStore application will use asynchronous HTTP requests to get model data provided by a RESTful web service. As I describe in Chapter 23, REST is an approach to designing web services that uses the HTTP method or verb to specify an operation and the URL to select the data objects that the operation applies to.

I added the json-server package to the project in the previous section. This is an excellent package for creating web services from JSON data or JavaScript code. Add the statement shown in Listing 5-4 to the scripts section of the package.json file so that the json-server package can be started from the command line.

Listing 5-4. Adding a Script in the package.json File in the SportsStore Folder

```
...  
  "scripts": {  
    "ng": "ng",  
    "start": "ng serve",  
    "build": "ng build",  
    "watch": "ng build --watch --configuration development",  
    "test": "ng test",  
    "json": "json-server data.js -p 3500 -m authMiddleware.js"  
  },  
  ...
```

To provide the json-server package with data to work with, I added a file called `data.js` in the `SportsStore` folder and added the code shown Listing 5-5, which will ensure that the same data is available whenever the json-server package is started so that I have a fixed point of reference during development.

Tip It is important to pay attention to the filenames when creating the configuration files. Some have the `.json` extension, which means they contain static data formatted as JSON. Other files have the `.js` extension, which means they contain JavaScript code. Each tool required for Angular development has expectations about its configuration file.

Listing 5-5. The Contents of the data.js File in the SportsStore Folder

```
module.exports = function () {
    return {
        products: [
            { id: 1, name: "Kayak", category: "Watersports",
                description: "A boat for one person", price: 275 },
            { id: 2, name: "Lifejacket", category: "Watersports",
                description: "Protective and fashionable", price: 48.95 },
            { id: 3, name: "Soccer Ball", category: "Soccer",
                description: "FIFA-approved size and weight", price: 19.50 },
            { id: 4, name: "Corner Flags", category: "Soccer",
                description: "Give your playing field a professional touch",
                price: 34.95 },
            { id: 5, name: "Stadium", category: "Soccer",
                description: "Flat-packed 35,000-seat stadium", price: 79500 },
            { id: 6, name: "Thinking Cap", category: "Chess",
                description: "Improve brain efficiency by 75%", price: 16 },
            { id: 7, name: "Unsteady Chair", category: "Chess",
                description: "Secretly give your opponent a disadvantage",
                price: 29.95 },
            { id: 8, name: "Human Chess Board", category: "Chess",
                description: "A fun game for the family", price: 75 },
            { id: 9, name: "Bling King", category: "Chess",
                description: "Gold-plated, diamond-studded King", price: 1200 }
        ],
        orders: []
    }
}
```

This code defines two data collections that will be presented by the RESTful web service. The `products` collection contains the products for sale to the customer, while the `orders` collection will contain the orders that customers have placed (but which is currently empty).

The data stored by the RESTful web service needs to be protected so that ordinary users can't modify the products or change the status of orders. The `json-server` package doesn't include any built-in authentication features, so I created a file called `authMiddleware.js` in the `SportsStore` folder and added the code shown in Listing 5-6.

Listing 5-6. The Contents of the authMiddleware.js File in the SportsStore Folder

```
const jwt = require("jsonwebtoken");

const APP_SECRET = "myappsecret";
const USERNAME = "admin";
const PASSWORD = "secret";

const mappings = {
    get: ["/api/orders", "/orders"],
    post: ["/api/products", "/products", "/api/categories", "/categories"]
}
```

```

function requiresAuth(method, url) {
  return (mappings[method.toLowerCase()] || [])
    .find(p => url.startsWith(p)) !== undefined;
}

module.exports = function (req, res, next) {
  if (req.url.endsWith("/login") && req.method == "POST") {
    if (req.body && req.body.name == USERNAME && req.body.password == PASSWORD) {
      let token = jwt.sign({ data: USERNAME, expiresIn: "1h" }, APP_SECRET);
      res.json({ success: true, token: token });
    } else {
      res.json({ success: false });
    }
    res.end();
    return;
  } else if (requiresAuth(req.method, req.url)) {
    let token = req.headers["authorization"] || "";
    if (token.startsWith("Bearer")) {
      token = token.substring(7, token.length - 1);
      try {
        jwt.verify(token, APP_SECRET);
        next();
        return;
      } catch (err) { }
    }
    res.statusCode = 401;
    res.end();
    return;
  }
  next();
}

```

This code inspects HTTP requests sent to the RESTful web service and implements some basic security features. This is server-side code that is not directly related to Angular development, so don't worry if its purpose isn't immediately obvious. I explain the authentication and authorization process in Chapter 7, including how to authenticate users with Angular.

Caution Don't use the code in Listing 5-6 other than for the SportsStore application. It contains weak passwords that are hardwired into the code. This is fine for the SportsStore project because the emphasis is on client-side development with Angular, but this is not suitable for real projects.

Preparing the HTML File

Every Angular web application relies on an HTML file that is loaded by the browser and that loads and starts the application. Edit the `index.html` file in the `SportsStore/src` folder to remove the placeholder content and to add the elements shown in Listing 5-7.

Listing 5-7. Preparing the index.html File in the src Folder

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>SportsStore</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
</head>
<body class="p-2">
  <app>SportsStore Will Go Here</app>
</body>
</html>
```

The HTML document includes an app element, which is the placeholder for the SportsStore functionality. There is also a base element, which is required by the Angular URL routing features, which I add to the SportsStore project in Chapter 6.

Creating the Folder Structure

An important part of setting up an Angular application is to create the folder structure. The ng new command sets up a project that puts all of the application's files in the src folder, with the Angular files in the src/app folder. To add some structure to the project, create the additional folders shown in Table 5-1.

Table 5-1. The Additional Folders Required for the SportsStore Project

Folder	Description
SportsStore/src/app/model	This folder will contain the code for the data model.
SportsStore/src/app/store	This folder will contain the functionality for basic shopping.
SportsStore/src/app/admin	This folder will contain the functionality for administration.

Running the Example Application

Make sure that all the changes have been saved, and run the following command in the SportsStore folder:

```
ng serve --open
```

This command will start the development tools set up by the ng new command, which will automatically compile and package the code and content files in the src folder whenever a change is detected. A new browser window will open and show the content illustrated in Figure 5-1.

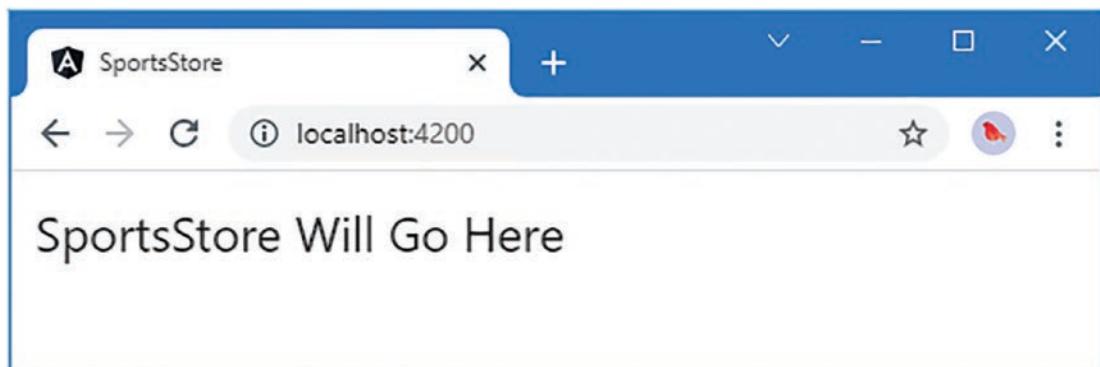


Figure 5-1. Running the example application

The development web server will start on port 4200, so the URL for the application will be `http://localhost:4200`. You don't have to include the name of the HTML document because `index.html` is the default file that the server responds with. (You will see errors in the browser's JavaScript console, which can be ignored for the moment.)

Starting the RESTful Web Service

To start the RESTful web service, open a new command prompt, navigate to the `SportsStore` folder, and run the following command:

```
npm run json
```

The RESTful web service is configured to run on port 3500. To test the web service request, use the browser to request the URL `http://localhost:3500/products/1`. The browser will display a JSON representation of one of the products defined in Listing 5-5, as follows:

```
{  
  "id": 1,  
  "name": "Kayak",  
  "category": "Watersports",  
  "description": "A boat for one person",  
  "price": 275  
}
```

Preparing the Angular Project Features

Every Angular project requires some basic preparation. In the sections that follow, I replace the placeholder content to build the foundation for the SportsStore application.

Updating the Root Component

The root component is the Angular building block that will manage the contents of the app element in the HTML document from Listing 5-7. An application can contain many components, but there is always a root component that takes responsibility for the top-level content presented to the user. I edited the file called `app.component.ts` in the SportsStore/src/app folder and replaced the existing code with the statements shown in Listing 5-8.

Listing 5-8. Replacing the Contents of the `app.component.ts` File in the `src/app` Folder

```
import { Component } from "@angular/core";

@Component({
  selector: "app",
  template: `<div class="bg-success p-2 text-center text-white">
    This is SportsStore
  </div>`
})
export class AppComponent { }
```

The `@Component` decorator tells Angular that the `AppComponent` class is a component, and its properties configure how the component is applied. All the component properties are described in Chapter 15, but the properties shown in the listing are the most basic and most frequently used. The `selector` property tells Angular how to apply the component in the HTML document, and the `template` property defines the HTML content the component will display. Components can define inline templates, like this one, or they use external HTML files, which can make managing complex content easier.

There is no code in the `AppComponent` class because the root component in an Angular project exists just to manage the content shown to the user. Initially, I'll manage the content displayed by the root component manually, but in Chapter 6, I use a feature called *URL routing* to adapt the content automatically based on user actions.

Inspecting the Root Module

There are two types of Angular modules: feature modules and the root module. Feature modules are used to group related application functionality to make the application easier to manage. I create feature modules for each major functional area of the application, including the data model, the store interface presented to users, and the administration interface.

The root module is used to describe the application to Angular. The description includes which feature modules are required to run the application, which custom features should be loaded, and the name of the root component. The conventional name of the root module file is `app.module.ts`, which is created in the SportsStore/src/app folder. No changes are required to this file for the moment; Listing 5-9 shows its initial content.

Listing 5-9. The Initial Contents of the `app.module.ts` File in the `src/app` Folder

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';

@NgModule({
```

```

declarations: [AppComponent],
imports: [BrowserModule],
providers: [],
bootstrap: [AppComponent]
})
export class AppModule { }

```

Similar to the root component, there is no code in the root module's class. That's because the root module only really exists to provide information through the `@NgModule` decorator. The `imports` property tells Angular that it should load the `BrowserModule` feature module, which contains the core Angular features required for a web application.

The `declarations` property tells Angular that it should load the root component, the `providers` property tells Angular about the shared objects used by the application, and the `bootstrap` property tells Angular that the root component is the `AppComponent` class. I'll add information to this decorator's properties as I add features to the SportsStore application, but this basic configuration is enough to start the application.

Inspecting the Bootstrap File

The next piece of plumbing is the bootstrap file, which starts the application. This book is focused on using Angular to create applications that work in web browsers, but the Angular platform can be ported to different environments. The bootstrap file uses the Angular browser platform to load the root module and start the application. No changes are required for the contents of the `main.ts` file, which is in the `SportsStore/src` folder, as shown in Listing 5-10.

Listing 5-10. The Contents of the `main.ts` File in the `src` Folder

```

import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';
import { environment } from './environments/environment';

if (environment.production) {
  enableProdMode();
}

platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.error(err));

```

The development tools detect the changes to the project's file, compile the code files, and automatically reload the browser, producing the content shown in Figure 5-2.

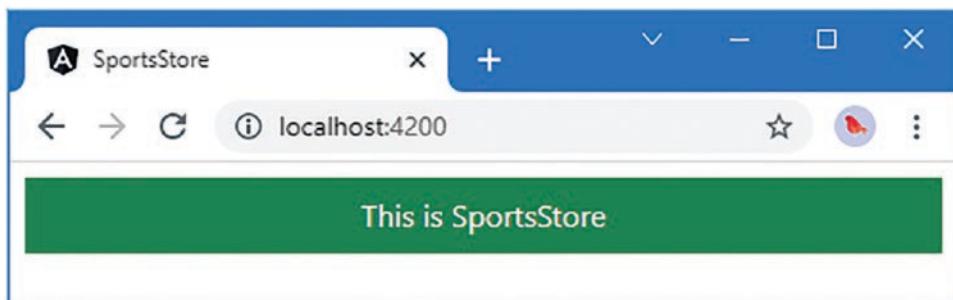


Figure 5-2. Starting the SportsStore application

Starting the Data Model

The best place to start any new project is the data model. I want to get to the point where you can see some Angular features at work, so rather than define the data model from end to end, I am going to put some basic functionality in place using dummy data. I'll use this data to create user-facing features and then return to the data model to wire it up to the RESTful web service in Chapter 6.

Creating the Model Classes

Every data model needs classes that describe the types of data that will be contained in the data model. For the SportsStore application, this means classes that describe the products sold in the store and the orders that are received from customers.

Being able to describe products will be enough to get started with the SportsStore application, and I'll create other model classes to support features as I implement them. I created a file called `product.model.ts` in the `SportsStore/src/app/model` folder and added the code shown in Listing 5-11.

Listing 5-11. The Contents of the `product.model.ts` File in the `src/app/model` Folder

```
export class Product {

    constructor(
        public id?: number,
        public name?: string,
        public category?: string,
        public description?: string,
        public price?: number) { }

}
```

The `Product` class defines a constructor that accepts `id`, `name`, `category`, `description`, and `price` properties, which correspond to the structure of the data used to populate the RESTful web service. The question marks (the `?` characters) that follow the parameter names indicate that these are optional parameters that can be omitted when creating new objects using the `Product` class, which can be useful when writing applications where model object properties will be populated using HTML forms.

Creating the Dummy Data Source

To prepare for the transition from dummy to real data, I am going to feed the application data using a data source. The rest of the application won't know where the data is coming from, which will make the switch to getting data using HTTP requests seamless.

I added a file called `static.datasource.ts` to the `SportsStore/src/app/model` folder and defined the class shown in Listing 5-12.

Listing 5-12. The Contents of the `static.datasource.ts` File in the `src/app/model` Folder

```
import { Injectable } from "@angular/core";
import { Product } from "./product.model";
import { Observable, from } from "rxjs";

@Injectable()
export class StaticDataSource {
    private products: Product[] = [
        new Product(1, "Product 1", "Category 1", "Product 1 (Category 1)", 100),
        new Product(2, "Product 2", "Category 1", "Product 2 (Category 1)", 100),
        new Product(3, "Product 3", "Category 1", "Product 3 (Category 1)", 100),
        new Product(4, "Product 4", "Category 1", "Product 4 (Category 1)", 100),
        new Product(5, "Product 5", "Category 1", "Product 5 (Category 1)", 100),
        new Product(6, "Product 6", "Category 2", "Product 6 (Category 2)", 100),
        new Product(7, "Product 7", "Category 2", "Product 7 (Category 2)", 100),
        new Product(8, "Product 8", "Category 2", "Product 8 (Category 2)", 100),
        new Product(9, "Product 9", "Category 2", "Product 9 (Category 2)", 100),
        new Product(10, "Product 10", "Category 2", "Product 10 (Category 2)", 100),
        new Product(11, "Product 11", "Category 3", "Product 11 (Category 3)", 100),
        new Product(12, "Product 12", "Category 3", "Product 12 (Category 3)", 100),
        new Product(13, "Product 13", "Category 3", "Product 13 (Category 3)", 100),
        new Product(14, "Product 14", "Category 3", "Product 14 (Category 3)", 100),
        new Product(15, "Product 15", "Category 3", "Product 15 (Category 3)", 100),
    ];

    getProducts(): Observable<Product[]> {
        return from([this.products]);
    }
}
```

The `StaticDataSource` class defines a method called `getProducts`, which returns the dummy data. The result of calling the `getProducts` method is an `Observable<Product[]>`, which is an `Observable` that produces arrays of `Product` objects.

The `Observable` class is provided by the Reactive Extensions package, which is used by Angular to handle state changes in applications, as described in Chapter 4. An `Observable` object represents an asynchronous task that will produce a result at some point in the future. Angular exposes its use of `Observable` objects for some features, including making HTTP requests, and this is why the `getProducts` method returns an `Observable<Product[]>` rather than simply returning the data synchronously.

The `@Injectable` decorator has been applied to the `StaticDataSource` class. This decorator is used to tell Angular that this class will be used as a service, which allows other classes to access its functionality through a feature called *dependency injection*, which is described in Chapters 17 and 18. You'll see how services work as the application takes shape.

Tip Notice that I have to import `Injectable` from the `@angular/core` JavaScript module so that I can apply the `@Injectable` decorator. I won't highlight all the different Angular classes that I import for the SportsStore example, but you can get full details in the chapters that describe the features they relate to.

Creating the Model Repository

The data source is responsible for providing the application with the data it requires, but access to that data is typically mediated by a *repository*, which is responsible for distributing that data to individual application building blocks so that the details of how the data has been obtained are kept hidden. I added a file called `product.repository.ts` in the `SportsStore/src/app/model` folder and defined the class shown in Listing 5-13.

Listing 5-13. The Contents of the `product.repository.ts` File in the `src/app/model` Folder

```
import { Injectable } from "@angular/core";
import { Product } from "./product.model";
import { StaticDataSource } from "./static.datasource";

@Injectable()
export class ProductRepository {
    private products: Product[] = [];
    private categories: string[] = [];

    constructor(private dataSource: StaticDataSource) {
        dataSource.getProducts().subscribe(data => {
            this.products = data;
            this.categories = data.map(p => p.category ?? "(None)")
                .filter((c, index, array) => array.indexOf(c) == index).sort();
        });
    }

    getProducts(category?: string): Product[] {
        return this.products
            .filter(p => category == undefined || category == p.category);
    }

    getProduct(id: number): Product | undefined {
        return this.products.find(p => p.id == id);
    }

    getCategories(): string[] {
        return this.categories;
    }
}
```

When Angular needs to create a new instance of the repository, it will inspect the class and see that it needs a `StaticDataSource` object to invoke the `ProductRepository` constructor and create a new object. The repository constructor calls the data source's `getProducts` method and then uses the `subscribe` method on the `Observable` object that is returned to receive the product data.

Creating the Feature Module

I am going to define an Angular feature model that will allow the data model functionality to be easily used elsewhere in the application. I added a file called `model.module.ts` in the `SportsStore/src/app/model` folder and defined the class shown in Listing 5-14.

Tip Don't worry if all the filenames seem similar and confusing. You will get used to the way that Angular applications are structured as you work through the other chapters in the book, and you will soon be able to look at the files in an Angular project and know what they are all intended to do.

Listing 5-14. The Contents of the `model.module.ts` File in the `src/app/model` Folder

```
import { NgModule } from "@angular/core";
import { ProductRepository } from "./product.repository";
import { StaticDataSource } from "./static.datasource";

@NgModule({
    providers: [ProductRepository, StaticDataSource]
})
export class ModelModule { }
```

The `@NgModule` decorator is used to create feature modules, and its properties tell Angular how the module should be used. There is only one property in this module, `providers`, and it tells Angular which classes should be used as services for the dependency injection feature, which is described in Chapters 17 and 18. Feature modules—and the `@NgModule` decorator—are described in Chapter 19.

Starting the Store

Now that the data model is in place, I can start to build out the store functionality, which will let the user see the products for sale and place orders for them. The basic structure of the store will be a two-column layout, with category buttons that allow the list of products to be filtered and a table that contains the list of products, as illustrated by Figure 5-3.



Figure 5-3. The basic structure of the store

In the sections that follow, I'll use Angular features and the data in the model to create the layout shown in the figure.

Creating the Store Component and Template

As you become familiar with Angular, you will learn that features can be combined to solve the same problem in different ways. I try to introduce some variety into the SportsStore project to showcase some important Angular features, but I am going to keep things simple for the moment in the interest of being able to get the project started quickly.

With this in mind, the starting point for the store functionality will be a new component, which is a class that provides data and logic to an HTML template, which contains data bindings that generate content dynamically. I created a file called `store.component.ts` in the `SportsStore/src/app/store` folder and defined the class shown in Listing 5-15.

Listing 5-15. The Contents of the `store.component.ts` File in the `src/app/store` Folder

```
import { Component } from "@angular/core";
import { Product } from "../model/product.model";
import { ProductRepository } from "../model/product.repository";

@Component({
  selector: "store",
  templateUrl: "store.component.html"
})
export class StoreComponent {

  constructor(private repository: ProductRepository) { }

  get products(): Product[] {
    return this.repository.getProducts();
  }
}
```

```

    get categories(): string[] {
        return this.repository.getCategories();
    }
}
}

```

The `@Component` decorator has been applied to the `StoreComponent` class, which tells Angular that it is a component. The decorator's properties tell Angular how to apply the component to HTML content (using an element called `store`) and how to find the component's template (in a file called `store.component.html`).

The `StoreComponent` class provides the logic that will support the template content. The constructor receives a `ProductRepository` object as an argument, provided through the dependency injection feature described in Chapters 17 and 18. The component defines `products` and `categories` properties that will be used to generate HTML content in the template, using data obtained from the repository. To provide the component with its template, I created a file called `store.component.html` in the `SportsStore/src/app/store` folder and added the HTML content shown in Listing 5-16.

Listing 5-16. The Contents of the `store.component.html` File in the `src/app/store` Folder

```

<div class="container-fluid">
    <div class="row">
        <div class="bg-dark text-white p-2">
            <span class="navbar-brand ml-2">SPORTS STORE</span>
        </div>
    </div>
    <div class="row text-white">
        <div class="col-3 bg-info p-2">
            {{categories.length}} Categories
        </div>
        <div class="col-9 bg-success p-2">
            {{products.length}} Products
        </div>
    </div>
</div>

```

The template is simple, just to get started. Most of the elements provide the structure for the store layout and apply some Bootstrap CSS classes. There are only two Angular data bindings at the moment, which are denoted by the `{{` and `}}` characters. These are *string interpolation* bindings, and they tell Angular to evaluate the binding expression and insert the result into the element. The expressions in these bindings display the number of products and categories provided by the store component.

Creating the Store Feature Module

There isn't much store functionality in place yet, but even so, some additional work is required to wire it up to the rest of the application. To create the Angular feature module for the store functionality, I created a file called `store.module.ts` in the `SportsStore/src/app/store` folder and added the code shown in Listing 5-17.

Listing 5-17. The Contents of the `store.module.ts` File in the `src/app/store` Folder

```

import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule } from "@angular/forms";

```

```

import { ModelModule } from "../model/model.module";
import { StoreComponent } from "./store.component";

@NgModule({
  imports: [ModelModule, BrowserModule, FormsModule],
  declarations: [StoreComponent],
  exports: [StoreComponent]
})
export class StoreModule { }

```

The `@NgModule` decorator configures the module, using the `imports` property to tell Angular that the store module depends on the model module as well as `BrowserModule` and `FormsModule`, which contain the standard Angular features for web applications and for working with HTML form elements. The decorator uses the `declarations` property to tell Angular about the `StoreComponent` class, and the `exports` property tells Angular the class can be also used in other parts of the application, which is important because it will be used by the root module.

Updating the Root Component and Root Module

Applying the basic model and store functionality requires updating the application's root module to import the two feature modules and also requires updating the root module's template to add the HTML element to which the component in the store module will be applied. Listing 5-18 shows the change to the root component's template.

Listing 5-18. Adding an Element in the app.component.ts File in the src/app Folder

```

import { Component } from "@angular/core";

@Component({
  selector: "app",
  template: "<store></store>"
})
export class AppComponent { }

```

The `store` element replaces the previous content in the root component's template and corresponds to the value of the `selector` property of the `@Component` decorator in Listing 5-15. Listing 5-19 shows the change required to the root module so that Angular loads the feature module that contains the store functionality.

Listing 5-19. Importing Feature Modules in the app.module.ts File in the src/app Folder

```

import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';
import { StoreModule } from "./store/store.module";

@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule, StoreModule],
  providers: []
})

```

```
bootstrap: [AppComponent]
})
export class AppModule { }
```

When you save the changes to the root module, Angular will have all the details it needs to load the application and display the content from the store module, as shown in Figure 5-4.

If you don't see the expected result, then stop the Angular development tools and use the `ng serve` command to start them again. This will repeat the build process for the project and should reflect the changes you have made.

All the building blocks created in the previous section work together to display the—admittedly simple—content, which shows how many products there are and how many categories they fit in to.

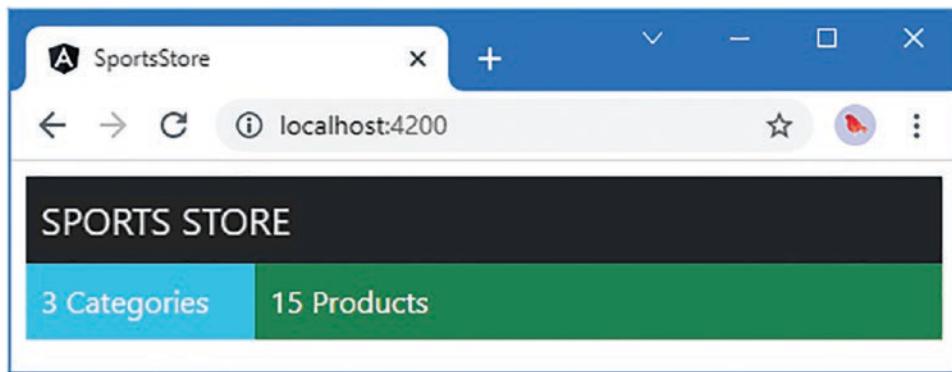


Figure 5-4. Basic features in the SportsStore application

Adding Store Features the Product Details

The nature of Angular development begins with a slow start as the foundation of the project is put in place and the basic building blocks are created. But once that's done, new features can be created relatively easily. In the sections that follow, I add features to the store so that the user can see the products on offer.

Displaying the Product Details

The obvious place to start is to display details for the products so that the customer can see what's on offer. Listing 5-20 adds HTML elements to the store component's template with data bindings that generate content for each product provided by the component.

Listing 5-20. Adding Elements in the store.component.html File in the src/app/store Folder

```
<div class="container-fluid">
    <div class="row">
        <div class="bg-dark text-white p-2">
            <span class="navbar-brand ml-2">SPORTS STORE</span>
        </div>
    </div>
    <div class="row text-white">
        <div class="col-3 bg-info p-2">
```

```

    {{categories.length}} Categories
</div>
<div class="col-9 p-2 text-dark">
  <div *ngFor="let product of products" class="card m-1 p-1 bg-light">
    <h4>
      {{product.name}}
      <span class="badge rounded-pill bg-primary" style="float:right">
        {{ product.price | currency:"USD": "symbol": "2.2-2" }}
      </span>
    </h4>
    <div class="card-text bg-white p-1">{{product.description}}</div>
  </div>
</div>
</div>

```

Most of the elements control the layout and appearance of the content. The most important change is the addition of an Angular data binding expression.

```

...
<div *ngFor="let product of products" class="card m-1 p-1 bg-light">
...

```

This is an example of a directive, which transforms the HTML element it is applied to. This specific directive is called `ngFor`, and it transforms the `div` element by duplicating it for each object returned by the component's `products` property. Angular includes a range of built-in directives that perform the most commonly required tasks, as described in Chapter 11.

As it duplicates the `div` element, the current object is assigned to a variable called `product`, which allows it to be referred to in other data bindings, such as this one, which inserts the value of the current product's `name` `description` property as the content of the `div` element:

```

...
<div class="card-text p-1 bg-white">{{product.description}}</div>
...

```

Not all data in an application's data model can be displayed directly to the user. Angular includes a feature called *pipes*, which are classes used to transform or prepare a data value for its use in a data binding. There are several built-in pipes included with Angular, including the `currency` pipe, which formats number values as currencies, like this:

```

...
{{ product.price | currency:"USD": "symbol": "2.2-2" }}
...

```

The syntax for applying pipes can be a little awkward, but the expression in this binding tells Angular to format the `price` property of the current product using the `currency` pipe, with the currency conventions from the United States. Save the changes to the template, and you will see a list of the products in the data model displayed as a long list, as illustrated in Figure 5-5.

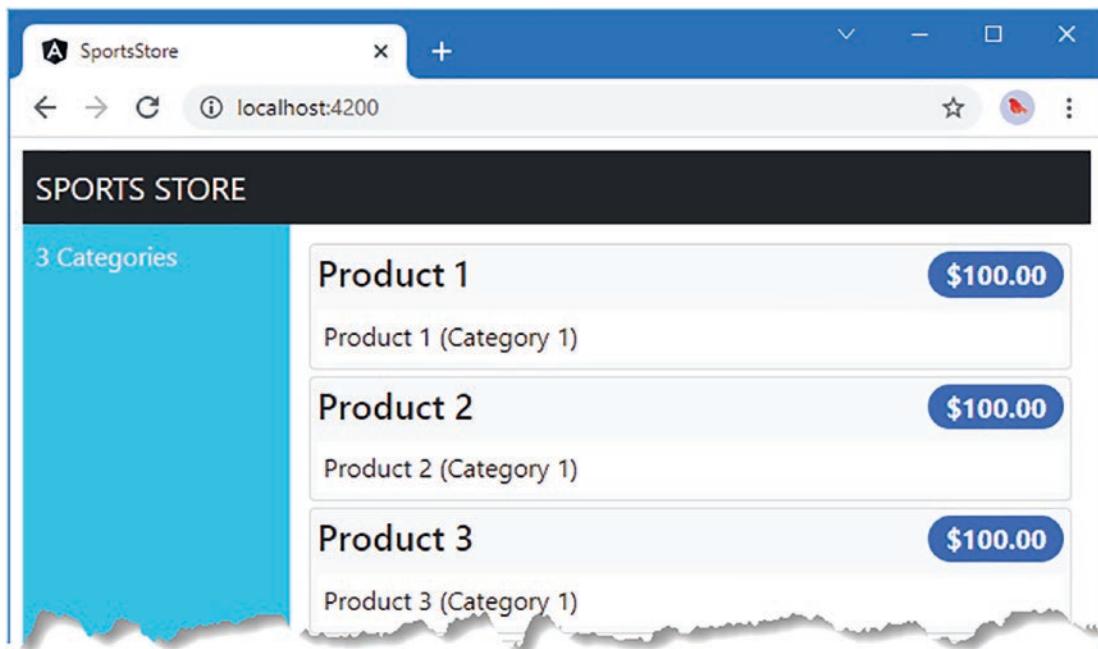


Figure 5-5. Displaying product information

Adding Category Selection

Adding support for filtering the list of products by category requires preparing the store component so that it keeps track of which category the user wants to display and requires changing the way that data is retrieved to use that category, as shown in Listing 5-21.

Listing 5-21. Adding Category Filtering in the store.component.ts File in the src/app/store Folder

```
import { Component } from "@angular/core";
import { Product } from "../model/product.model";
import { ProductRepository } from "../model/product.repository";

@Component({
  selector: "store",
  templateUrl: "store.component.html"
})
export class StoreComponent {
  selectedCategory: string | undefined;

  constructor(private repository: ProductRepository) { }

  get products(): Product[] {
    return this.repository.getProducts(this.selectedCategory);
  }
}
```

```

get categories(): string[] {
    return this.repository.getCategories();
}

changeCategory(newCategory?: string) {
    this.selectedCategory = newCategory;
}

}

```

The changes are simple because they build on the foundation that took so long to create at the start of the chapter. The `selectedCategory` property is assigned the user's choice of category (where `undefined` means all categories) and is used in the `updateData` method as an argument to the `getProducts` method, delegating the filtering to the data source. The `changeCategory` method brings these two members together in a method that can be invoked when the user makes a category selection.

[Listing 5-22](#) shows the corresponding changes to the component's template to provide the user with the set of buttons that change the selected category and show which category has been picked.

Listing 5-22. Adding Category Buttons in the store.component.html File in the src/app/store Folder

```

<div class="container-fluid">
    <div class="row">
        <div class="bg-dark text-white p-2">
            <span class="navbar-brand ml-2">SPORTS STORE</span>
        </div>
    </div>
    <div class="row text-white">
        <div class="col-3 p-2">
            <div class="d-grid gap-2">
                <button class="btn btn-outline-primary" (click)="changeCategory()">
                    Home
                </button>
                <button *ngFor="let cat of categories"
                        class="btn btn-outline-primary"
                        [class.active]="cat == selectedCategory"
                        (click)="changeCategory(cat)">
                    {{cat}}
                </button>
            </div>
        </div>
        <div class="col-9 p-2 text-dark">
            <div *ngFor="let product of products" class="card m-1 p-1 bg-light">
                <h4>
                    {{product.name}}
                    <span class="badge rounded-pill bg-primary" style="float:right">
                        {{ product.price | currency:"USD":"symbol":2.2-2" }}
                    </span>
                </h4>
                <div class="card-text bg-white p-1">{{product.description}}</div>
            </div>
        </div>
    </div>
</div>

```

There are two new button elements in the template. The first is a Home button, and it has an event binding that invokes the component's `changeCategory` method when the button is clicked. No argument is provided to the method, which has the effect of setting the category to null and selecting all the products.

The `ngFor` binding has been applied to the other button element, with an expression that will repeat the element for each value in the array returned by the component's `categories` property. The button has a `click` event binding whose expression calls the `changeCategory` method to select the current category, which will filter the products displayed to the user. There is also a `class` binding, which adds the `button` element to the active class when the category associated with the button is the selected category. This provides the user with visual feedback when the categories are filtered, as shown in Figure 5-6.

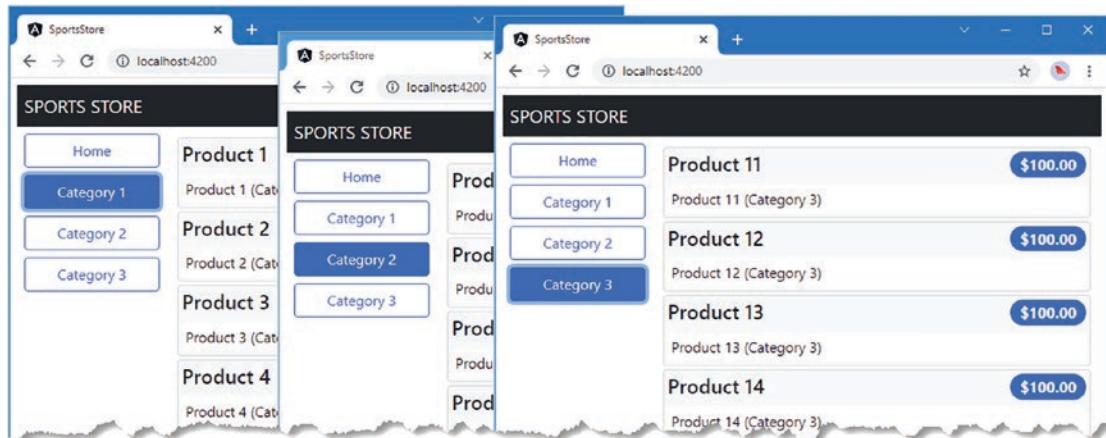


Figure 5-6. Selecting product categories

Adding Product Pagination

Filtering the products by category has helped make the product list more manageable, but a more typical approach is to break the list into smaller sections and present each of them as a page, along with navigation buttons that move between the pages. Listing 5-23 enhances the store component so that it keeps track of the current page and the number of items on a page.

Listing 5-23. Adding Pagination Support in the store.component.ts File in the src/app/store Folder

```
import { Component } from "@angular/core";
import { Product } from "../model/product.model";
import { ProductRepository } from "../model/product.repository";

@Component({
  selector: "store",
  templateUrl: "store.component.html"
})
export class StoreComponent {
  selectedCategory: string | undefined;
  productsPerPage = 4;
  selectedPage = 1;
```

```

constructor(private repository: ProductRepository) { }

get products(): Product[] {
    let pageIndex = (this.selectedPage - 1) * this.productsPerPage
    return this.repository.getProducts(this.selectedCategory)
        .slice(pageIndex, pageIndex + this.productsPerPage);
}

get categories(): string[] {
    return this.repository.getCategories();
}

changeCategory(newCategory?: string) {
    this.selectedCategory = newCategory;
}

changePage(newPage: number) {
    this.selectedPage = newPage;
}

changePageSize(newSize: number) {
    this.productsPerPage = Number(newSize);
    this.changePage(1);
}

get pageNumbers(): number[] {
    return Array(Math.ceil(this.repository
        .getProducts(this.selectedCategory).length / this.productsPerPage))
        .fill(0).map((x, i) => i + 1);
}
}
}

```

There are two new features in this listing. The first is the ability to get a page of products, and the second is to change the size of the pages, allowing the number of products that each page contains to be altered.

There is an oddity that the component has to work around. There is a limitation in the built-in `ngFor` directive that Angular provides, which can generate content only for the objects in an array or a collection, rather than using a counter. Since I need to generate numbered page navigation buttons, this means I need to create an array that contains the numbers I need, like this:

```

...
return Array(Math.ceil(this.repository.getProducts(this.selectedCategory).length
    / this.productsPerPage)).fill(0).map((x, i) => i + 1);
...

```

This statement creates a new array, fills it with the value 0, and then uses the `map` method to generate a new array with the number sequence. This works well enough to implement the pagination feature, but it feels awkward, and I demonstrate a better approach in the next section. Listing 5-24 shows the changes to the store component's template to implement the pagination feature.

Listing 5-24. Adding Pagination in the store.component.html File in the src/app/store Folder

```

<div class="container-fluid">
  <div class="row">
    <div class="bg-dark text-white p-2">
      <span class="navbar-brand ml-2">SPORTS STORE</span>
    </div>
  </div>
  <div class="row text-white">
    <div class="col-3 p-2">
      <div class="d-grid gap-2">
        <button class="btn btn-outline-primary" (click)="changeCategory()">
          Home
        </button>
        <button *ngFor="let cat of categories"
                class="btn btn-outline-primary"
                [class.active]="cat == selectedCategory"
                (click)="changeCategory(cat)">
          {{cat}}
        </button>
      </div>
    </div>
    <div class="col-9 p-2 text-dark">
      <div *ngFor="let product of products" class="card m-1 p-1 bg-light">
        <h4>
          {{product.name}}
          <span class="badge rounded-pill bg-primary" style="float:right">
            {{ product.price | currency:"USD":"symbol":2.2-2" }}
          </span>
        </h4>
        <div class="card-text bg-white p-1">{{product.description}}</div>
      </div>
      <div class="form-inline float-start mr-1">
        <select class="form-control" [value]="productsPerPage"
               (change)="changePageSize($any($event).target.value)">
          <option value="3">3 per Page</option>
          <option value="4">4 per Page</option>
          <option value="6">6 per Page</option>
          <option value="8">8 per Page</option>
        </select>
      </div>
      <div class="btn-group float-end">
        <button *ngFor="let page of pageNumbers" (click)="changePage(page)"
               class="btn btn-outline-primary"
               [class.active]="page == selectedPage">
          {{page}}
        </button>
      </div>
    </div>
  </div>
</div>

```

The new elements add a select element that allows the size of the page to be changed and a set of buttons that navigate through the product pages. The new elements have data bindings to wire them up to the properties and methods provided by the component. The result is a more manageable set of products, as shown in Figure 5-7.

Tip The select element in Listing 5-24 is populated with option elements that are statically defined, rather than created using data from the component. One impact of this is that when the selected value is passed to the changePageSize method, it will be a string value, which is why the argument is parsed to a number before being used to set the page size in Listing 5-23. Care must be taken when receiving data values from HTML elements to ensure they are of the expected type. TypeScript type annotations don't help in this situation because the data binding expression is evaluated at runtime, long after the TypeScript compiler has generated JavaScript code that doesn't contain the extra type information.

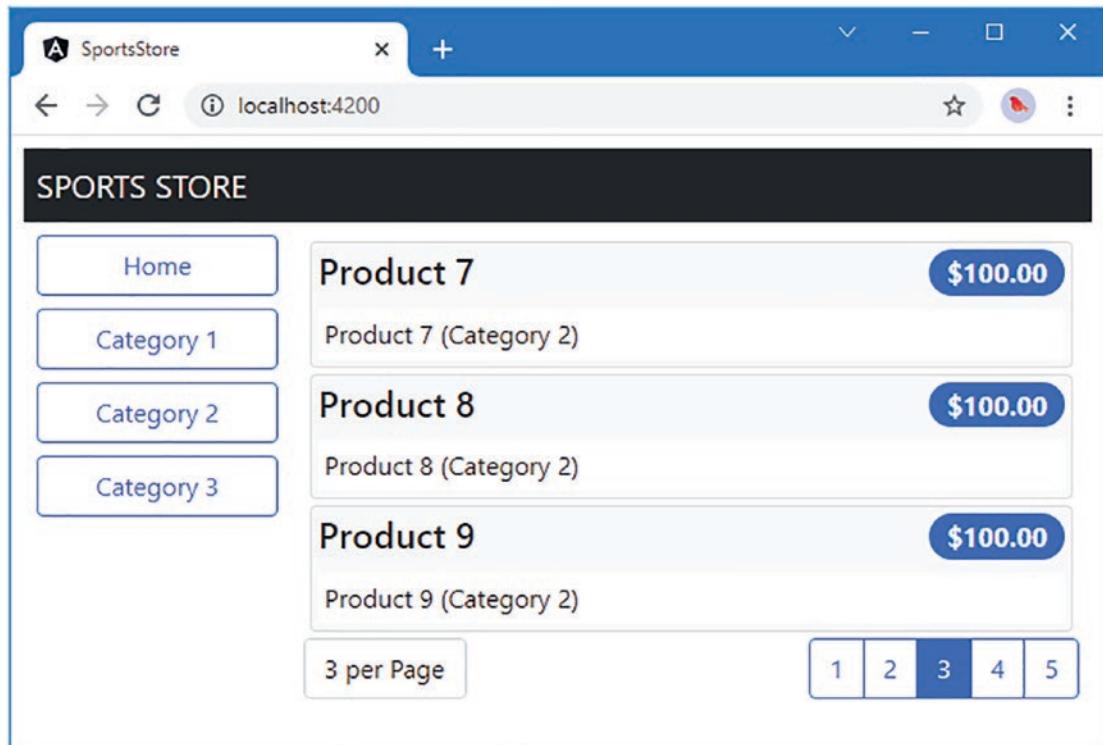


Figure 5-7. Pagination for products

Creating a Custom Directive

In this section, I am going to create a custom directive so that I don't have to generate an array full of numbers to create the page navigation buttons. Angular provides a good range of built-in directives, but it is a simple process to create your own directives to solve problems that are specific to your application or to

support features that the built-in directives don't have. I added a file called `counter.directive.ts` in the `src/app/store` folder and used it to define the class shown in Listing 5-25.

Listing 5-25. The Contents of the `counter.directive.ts` File in the `src/app/store` Folder

```
import {
  Directive, ViewContainerRef, TemplateRef, Input, SimpleChanges
} from "@angular/core";

@Directive({
  selector: "[counterOf]"
})
export class CounterDirective {

  constructor(private container: ViewContainerRef,
    private template: TemplateRef<Object>) {
  }

  @Input("counterOf")
  counter: number = 0;

  ngOnChanges(changes: SimpleChanges) {
    this.container.clear();
    for (let i = 0; i < this.counter; i++) {
      this.container.createEmbeddedView(this.template,
        new CounterDirectiveContext(i + 1));
    }
  }
}

class CounterDirectiveContext {
  constructor(public $implicit: any) { }
}
```

This is an example of a structural directive, which is described in detail in Chapter 14. This directive is applied to elements through a `counter` property and relies on special features that Angular provides for creating content repeatedly, just like the built-in `ngFor` directive. In this case, rather than yield each object in a collection, the custom directive yields a series of numbers that can be used to create the page navigation buttons.

Tip This directive deletes all the content it has created and starts again when the number of pages changes. This can be an expensive process in more complex directives, and I explain how to improve performance in Chapter 14.

To use the directive, it must be added to the `declarations` property of its feature module, as shown in Listing 5-26.

Listing 5-26. Registering the Custom Directive in the store.module.ts File in the src/app/store Folder

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule } from "@angular/forms";
import { ModelModule } from "../model/model.module";
import { StoreComponent } from "./store.component";
import { CounterDirective } from "./counter.directive";

@NgModule({
  imports: [ModelModule, BrowserModule, FormsModule],
  declarations: [StoreComponent, CounterDirective],
  exports: [StoreComponent]
})
export class StoreModule { }
```

Now that the directive has been registered, it can be used in the store component's template to replace the `ngFor` directive, as shown in Listing 5-27.

Listing 5-27. Replacing the Built-in Directive in the store.component.html File in the src/app/store Folder

```
<div class="container-fluid">
  <div class="row">
    <div class="bg-dark text-white p-2">
      <span class="navbar-brand ml-2">SPORTS STORE</span>
    </div>
  </div>
  <div class="row text-white">
    <div class="col-3 p-2">
      <div class="d-grid gap-2">
        <button class="btn btn-outline-primary" (click)="changeCategory()">
          Home
        </button>
        <button *ngFor="let cat of categories"
          class="btn btn-outline-primary"
          [class.active]="cat == selectedCategory"
          (click)="changeCategory(cat)">
          {{cat}}
        </button>
      </div>
    </div>
    <div class="col-9 p-2 text-dark">
      <div *ngFor="let product of products" class="card m-1 p-1 bg-light">
        <h4>
          {{product.name}}
          <span class="badge rounded-pill bg-primary" style="float:right">
            {{ product.price | currency:"USD":"symbol":"2.2-2" }}
          </span>
        </h4>
        <div class="card-text bg-white p-1">{{product.description}}</div>
      </div>
    </div>
  </div>
</div>
```

```

<div class="form-inline float-start mr-1">
  <select class="form-control" [value]="productsPerPage"
    (change)="changePageSize($any($event).target.value)">
    <option value="3">3 per Page</option>
    <option value="4">4 per Page</option>
    <option value="6">6 per Page</option>
    <option value="8">8 per Page</option>
  </select>
</div>
<div class="btn-group float-end">
  <button *counter="let page of pageCount" (click)="changePage(page)"
    class="btn btn-outline-primary"
    [class.active]="page == selectedPage">
    {{page}}
  </button>
</div>
</div>
</div>

```

The new data binding relies on a property called `pageCount` to configure the custom directive. In Listing 5-28, I have replaced the array of numbers with a simple number that provides the expression value.

Listing 5-28. Supporting the Custom Directive in the store.component.ts File in the src/app/store Folder

```

import { Component } from "@angular/core";
import { Product } from "../model/product.model";
import { ProductRepository } from "../model/product.repository";

@Component({
  selector: "store",
  templateUrl: "store.component.html"
})
export class StoreComponent {
  selectedCategory: string | undefined;
  productsPerPage = 4;
  selectedPage = 1;

  constructor(private repository: ProductRepository) { }

  get products(): Product[] {
    let pageIndex = (this.selectedPage - 1) * this.productsPerPage
    return this.repository.getProducts(this.selectedCategory)
      .slice(pageIndex, pageIndex + this.productsPerPage);
  }

  get categories(): string[] {
    return this.repository.getCategories();
  }
}

```

```

changeCategory(newCategory?: string) {
    this.selectedCategory = newCategory;
}

changePage(newPage: number) {
    this.selectedPage = newPage;
}

changePageSize(newSize: number) {
    this.productsPerPage = Number(newSize);
    this.changePage(1);
}

// get pageNumbers(): number[] {
//     return Array(Math.ceil(this.repository
//         .getProducts(this.selectedCategory).length / this.productsPerPage))
//         .fill(0).map((x, i) => i + 1);
// }

get pageCount(): number {
    return Math.ceil(this.repository
        .getProducts(this.selectedCategory).length / this.productsPerPage)
}
}
}

```

There is no visual change to the SportsStore application, but this section has demonstrated that it is possible to supplement the built-in Angular functionality with custom code that is tailored to the needs of a specific project.

Summary

In this chapter, I started the SportsStore project. The early part of the chapter was spent creating the foundation for the project, including creating the root building blocks for the application and starting work on the feature modules. Once the foundation was in place, I was able to rapidly add features to display the dummy model data to the user, add pagination, and filter the products by category. I finished the chapter by creating a custom directive to demonstrate how the built-in features provided by Angular can be supplemented by custom code. In the next chapter, I continue to build the SportsStore application.

CHAPTER 6



SportsStore: Orders and Checkout

In this chapter, I continue adding features to the SportsStore application that I created in Chapter 5. I add support for a shopping cart and a checkout process and replace the dummy data with the data from the RESTful web service.

Preparing the Example Application

No preparation is required for this chapter, which continues using the SportsStore project from Chapter 5. To start the RESTful web service, open a command prompt and run the following command in the `SportsStore` folder:

```
npm run json
```

Open a second command prompt and run the following command in the `SportsStore` folder to start the development tools and HTTP server:

```
ng serve --open
```

Tip You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/Apress/pro-angular-5ed>. See Chapter 1 for how to get help if you have problems running the examples.

Creating the Cart

The user needs a cart into which products can be placed and used to start the checkout process. In the sections that follow, I'll add a cart to the application and integrate it into the store so that the user can select the products they want.

Creating the Cart Model

The starting point for the cart feature is a new model class that will be used to gather together the products that the user has selected. I added a file called `cart.model.ts` in the `src/app/model` folder and used it to define the class shown in Listing 6-1.

Listing 6-1. The Contents of the `cart.model.ts` File in the `src/app/model` Folder

```
import { Injectable } from "@angular/core";
import { Product } from "./product.model";

@Injectable()
export class Cart {
    public lines: CartLine[] = [];
    public itemCount: number = 0;
    public cartPrice: number = 0;

    addLine(product: Product, quantity: number = 1) {
        let line = this.lines.find(line => line.product.id == product.id);
        if (line != undefined) {
            line.quantity += quantity;
        } else {
            this.lines.push(new CartLine(product, quantity));
        }
        this.recalculate();
    }

    updateQuantity(product: Product, quantity: number) {
        let line = this.lines.find(line => line.product.id == product.id);
        if (line != undefined) {
            line.quantity = Number(quantity);
        }
        this.recalculate();
    }

    removeLine(id: number) {
        let index = this.lines.findIndex(line => line.product.id == id);
        this.lines.splice(index, 1);
        this.recalculate();
    }

    clear() {
        this.lines = [];
        this.itemCount = 0;
        this.cartPrice = 0;
    }

    private recalculate() {
        this.itemCount = 0;
        this.cartPrice = 0;
        for (let line of this.lines) {
            this.itemCount++;
            this.cartPrice += line.quantity * line.product.price;
        }
    }
}
```

```

        this.lines.forEach(l => {
            this.itemCount += l.quantity;
            this.cartPrice += l.lineTotal;
        })
    }
}

export class CartLine {
    constructor(public product: Product,
        public quantity: number) {}

    get lineTotal() {
        return this.quantity * (this.product.price ?? 0);
    }
}

```

Individual product selections are represented as an array of `CartLine` objects, each of which contains a `Product` object and a quantity. The `Cart` class keeps track of the total number of items that have been selected and their total cost.

There should be a single `Cart` object used throughout the entire application, ensuring that any part of the application can access the user's product selections. To achieve this, I am going to make the `Cart` a service, which means that Angular will take responsibility for creating an instance of the `Cart` class and will use it when it needs to create a component that has a `Cart` constructor argument. This is another use of the Angular dependency injection feature, which can be used to share objects throughout an application and which is described in detail in Chapters 17 and 18. The `@Injectable` decorator, which has been applied to the `Cart` class in the listing, indicates that this class will be used as a service.

Note Strictly speaking, the `@Injectable` decorator is required only when a class has its own constructor arguments to resolve, but it is a good idea to apply it anyway because it serves as a signal that the class is intended for use as a service.

Listing 6-2 registers the `Cart` class as a service in the `providers` property of the model feature module class.

Listing 6-2. Registering the `Cart` as a Service in the `model.module.ts` File in the `src/app/model` Folder

```

import { NgModule } from "@angular/core";
import { ProductRepository } from "./product.repository";
import { StaticDataSource } from "./static.datasource";
import { Cart } from "./cart.model";

@NgModule({
    providers: [ProductRepository, StaticDataSource, Cart]
})
export class ModelModule { }

```

Creating the Cart Summary Components

Components are the essential building blocks for Angular applications because they allow discrete units of code and content to be easily created. The SportsStore application will show users a summary of their product selections in the title area of the page, which I am going to implement by creating a component. I added a file called `cartSummary.component.ts` in the `src/app/store` folder and used it to define the component shown in Listing 6-3.

Listing 6-3. The Contents of the `cartSummary.component.ts` File in the `src/app/store` Folder

```
import { Component } from "@angular/core";
import { Cart } from "../model/cart.model";

@Component({
  selector: "cart-summary",
  templateUrl: "cartSummary.component.html"
})
export class CartSummaryComponent {

  constructor(public cart: Cart) { }
}
```

When Angular needs to create an instance of this component, it will have to provide a `Cart` object as a constructor argument, using the service that I configured in the previous section by adding the `Cart` class to the feature module's `providers` property. The default behavior for services means that a single `Cart` object will be created and shared throughout the application, although there are different service behaviors available (as described in Chapter 18).

To provide the component with a template, I created an HTML file called `cartSummary.component.html` in the same folder as the component class file and added the markup shown in Listing 6-4.

Listing 6-4. The Contents of the `cartSummary.component.html` File in the `src/app/store` Folder

```
<div class="float-end">
  <small class="fs-6">
    Your cart:
    <span *ngIf="cart.itemCount > 0">
      {{ cart.itemCount }} item(s)
      {{ cart.cartPrice | currency:"USD":symbol:"2.2-2" }}
    </span>
    <span *ngIf="cart.itemCount == 0">
      (empty)
    </span>
  </small>
  <button class="btn btn-sm bg-dark text-white" [disabled]="cart.itemCount == 0">
    <i class="fa fa-shopping-cart"></i>
  </button>
</div>
```

This template uses the `Cart` object provided by its component to display the number of items in the cart and the total cost. There is also a button that will start the checkout process when I add it to the application later in the chapter.

Tip The button element in Listing 6-4 is styled using classes defined by Font Awesome, which is one of the packages in the package.json file from Chapter 5. This open-source package provides excellent support for icons in web applications, including the shopping cart I need for the SportsStore application. See <http://fontawesome.io> for details.

Listing 6-5 registers the new component with the store feature module, in preparation for using it in the next section.

Listing 6-5. Registering the Component in the store.module.ts File in the src/app/store Folder

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule } from "@angular/forms";
import { ModelModule } from "../model/model.module";
import { StoreComponent } from "./store.component";
import { CounterDirective } from "./counter.directive";
import { CartSummaryComponent } from "./cartSummary.component";

@NgModule({
  imports: [ModelModule, BrowserModule, FormsModule],
  declarations: [StoreComponent, CounterDirective, CartSummaryComponent],
  exports: [StoreComponent]
})
export class StoreModule {}
```

Integrating the Cart into the Store

The store component is the key to integrating the cart and the cart widget into the application. Listing 6-6 updates the store component so that its constructor has a Cart parameter and defines a method that will add a product to the cart.

Listing 6-6. Adding Cart Support in the store.component.ts File in the src/app/store Folder

```
import { Component } from "@angular/core";
import { Product } from "../model/product.model";
import { ProductRepository } from "../model/product.repository";
import { Cart } from "../model/cart.model";

@Component({
  selector: "store",
  templateUrl: "store.component.html"
})
export class StoreComponent {
  selectedCategory: string | undefined;
  productsPerPage = 4;
  selectedPage = 1;

  constructor(private repository: ProductRepository,
    private cart: Cart) {}
```

```

get products(): Product[] {
    let pageIndex = (this.selectedPage - 1) * this.productsPerPage
    return this.repository.getProducts(this.selectedCategory)
        .slice(pageIndex, pageIndex + this.productsPerPage);
}

get categories(): string[] {
    return this.repository.getCategories();
}

changeCategory(newCategory?: string) {
    this.selectedCategory = newCategory;
}

changePage(newPage: number) {
    this.selectedPage = newPage;
}

changePageSize(newSize: number) {
    this.productsPerPage = Number(newSize);
    this.changePage(1);
}

get pageCount(): number {
    return Math.ceil(this.repository
        .getProducts(this.selectedCategory).length / this.productsPerPage)
}

addProductToCart(product: Product) {
    this.cart.addLine(product);
}
}

```

To complete the integration of the cart into the store component, Listing 6-7 adds the element that will apply the cart summary component to the store component's template and adds a button to each product description with the event binding that calls the addProductToCart method.

Listing 6-7. Applying the Component in the store.component.html File in the src/app/store Folder

```

<div class="container-fluid">
    <div class="row">
        <div class="bg-dark text-white p-2">
            <span class="navbar-brand ml-2">
                SPORTS STORE
                <cart-summary></cart-summary>
            </span>
        </div>
    </div>
    <div class="row text-white">
        <div class="col-3 p-2">
            <div class="d-grid gap-2">

```

```

<button class="btn btn-outline-primary" (click)="changeCategory()">
    Home
</button>
<button *ngFor="let cat of categories"
        class="btn btn-outline-primary"
        [class.active]="cat == selectedCategory"
        (click)="changeCategory(cat)">
    {{cat}}
</button>
</div>
</div>
<div class="col-9 p-2 text-dark">
    <div *ngFor="let product of products" class="card m-1 p-1 bg-light">
        <h4>
            {{product.name}}
            <span class="badge rounded-pill bg-primary" style="float:right">
                {{ product.price | currency:"USD":"symbol":2.2-2" }}
            </span>
        </h4>
        <div class="card-text bg-white p-1">
            {{product.description}}
            <button class="btn btn-success btn-sm float-end"
                   (click)="addProductToCart(product)">
                Add To Cart
            </button>
        </div>
    </div>
    <div class="form-inline float-start mr-1">
        <select class="form-control" [value]="productsPerPage"
               (change)="changePageSize($any($event).target.value)">
            <option value="3">3 per Page</option>
            <option value="4">4 per Page</option>
            <option value="6">6 per Page</option>
            <option value="8">8 per Page</option>
        </select>
    </div>
    <div class="btn-group float-end">
        <button *counter="let page of pageCount" (click)="changePage(page)"
               class="btn btn-outline-primary"
               [class.active]="page == selectedPage">
            {{page}}
        </button>
    </div>
</div>
</div>

```

The result is a button for each product that adds it to the cart, as shown in Figure 6-1. The full cart process isn't complete yet, but you can see the effect of each addition in the cart summary at the top of the page.

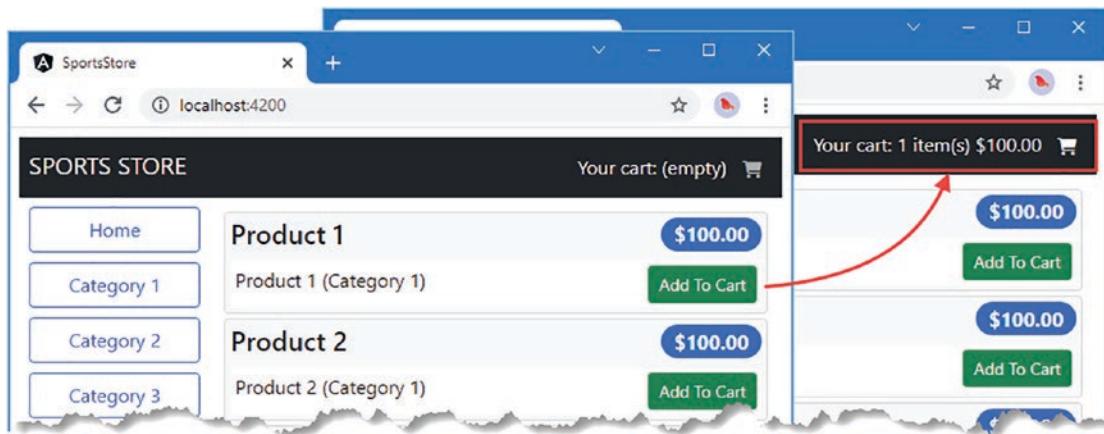


Figure 6-1. Adding cart support to the SportsStore application

Notice how clicking one of the Add To Cart buttons updates the summary component's content automatically. This happens because there is a single Cart object being shared between two components, and changes made by one component are reflected when Angular evaluates the data binding expressions in the other component.

Adding URL Routing

Most applications need to show different content to the user at different times. In the case of the SportsStore application, when the user clicks one of the Add To Cart buttons, they should be shown a detailed view of their selected products and given the chance to start the checkout process.

Angular supports a feature called *URL routing*, which uses the current URL displayed by the browser to select the components that are displayed to the user. This is an approach that makes it easy to create applications whose components are loosely coupled and easy to change without needing corresponding modifications elsewhere in the applications. URL routing also makes it easy to change the path that a user follows through an application.

For the SportsStore application, I am going to add support for three different URLs, which are described in Table 6-1. This is a simple configuration, but the routing system has a lot of features, which are described in detail in Chapters 24 to 26.

Table 6-1. The URLs Supported by the SportsStore Application

URL	Description
/store	This URL will display the list of products.
/cart	This URL will display the user's cart in detail.
/checkout	This URL will display the checkout process.

In the sections that follow, I create placeholder components for the SportsStore cart and order checkout stages and then integrate them into the application using URL routing. Once the URLs are implemented, I will return to the components and add more useful features.

Creating the Cart Detail and Checkout Components

Before adding URL routing to the application, I need to create the components that will be displayed by the /cart and /checkout URLs. I only need some basic placeholder content to get started, just to make it obvious which component is being displayed. I started by adding a file called cartDetail.component.ts in the src/app/store folder and defined the component shown in Listing 6-8.

Listing 6-8. The Contents of the cartDetail.component.ts File in the src/app/store Folder

```
import { Component } from "@angular/core";

@Component({
  template: `<div><h3 class="bg-info p-1 text-white">Cart Detail Component</h3></div>`
})
export class CartDetailComponent {}
```

Next, I added a file called checkout.component.ts in the src/app/store folder and defined the component shown in Listing 6-9.

Listing 6-9. The Contents of the checkout.component.ts File in the src/app/store Folder

```
import { Component } from "@angular/core";

@Component({
  template: `<div><h3 class="bg-info p-1 text-white">Checkout Component</h3></div>`
})
export class CheckoutComponent {}
```

This component follows the same pattern as the cart component and displays a placeholder message. Listing 6-10 registers the components in the store feature module and adds them to the exports property, which means they can be used elsewhere in the application.

Listing 6-10. Registering Components in the store.module.ts File in the src/app/store Folder

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule } from "@angular/forms";
import { ModelModule } from "../model/model.module";
import { StoreComponent } from "./store.component";
import { CounterDirective } from "./counter.directive";
import { CartSummaryComponent } from "./cartSummary.component";
import { CartDetailComponent } from "./cartDetail.component";
import { CheckoutComponent } from "./checkout.component";

@NgModule({
  imports: [ModelModule, BrowserModule, FormsModule],
  declarations: [StoreComponent, CounterDirective, CartSummaryComponent,
    CartDetailComponent, CheckoutComponent],
  exports: [StoreComponent, CartDetailComponent, CheckoutComponent]
})
export class StoreModule {}
```

Creating and Applying the Routing Configuration

Now that I have a range of components to display, the next step is to create the routing configuration that tells Angular how to map URLs into components. Each mapping of a URL to a component is known as a *URL route* or just a *route*. In Part 3, where I create more complex routing configurations, I define the routes in a separate file, but for this project, I am going to follow a simpler approach and define the routes within the `@NgModule` decorator of the application's root module, as shown in Listing 6-11.

Tip The Angular routing feature requires a `base` element in the HTML document, which provides the base URL against which routes are applied. This element was added to the `index.html` file by the `ng new` command when I created the SportsStore project in Chapter 5. If you omit the element, Angular will report an error and be unable to apply the routes.

Listing 6-11. Creating the Routing Configuration in the `app.module.ts` File in the `src/app` Folder

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';
import { StoreModule } from './store/store.module';
import { StoreComponent } from './store/store.component';
import { CheckoutComponent } from './store/checkout.component';
import { CartDetailComponent } from './store/cartDetail.component';
import { RouterModule } from '@angular/router';

@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule, StoreModule,
    RouterModule.forRoot([
      { path: "store", component: StoreComponent },
      { path: "cart", component: CartDetailComponent },
      { path: "checkout", component: CheckoutComponent },
      { path: "**", redirectTo: "/store" }
    ])
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

The `RouterModule.forRoot` method is passed a set of routes, each of which maps a URL to a component. The first three routes in the listing match the URLs from Table 6-1. The final route is a wildcard that redirects any other URL to `/store`, which will display `StoreComponent`.

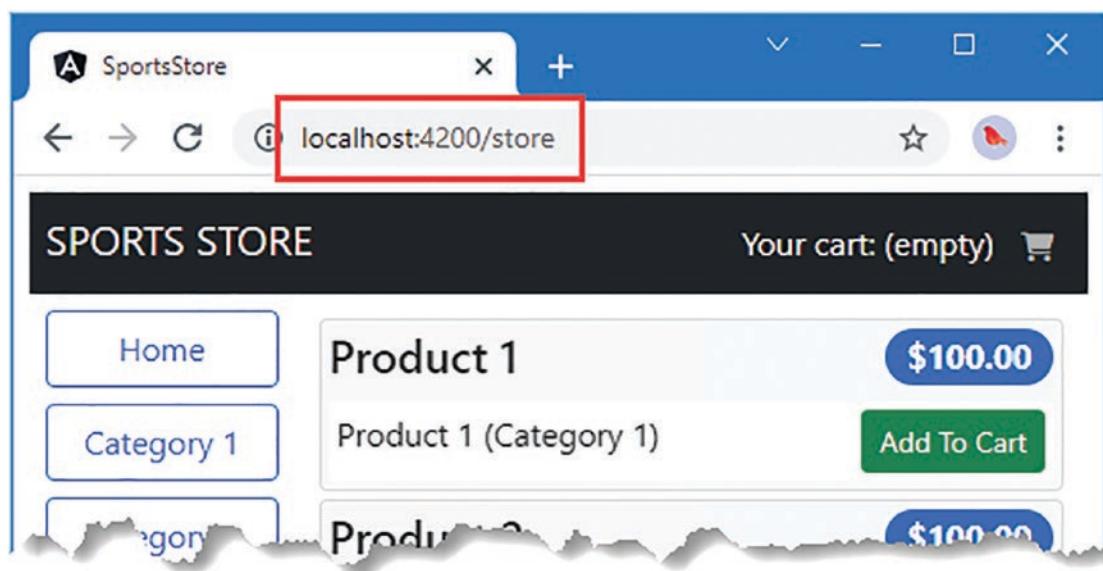
When the routing feature is used, Angular looks for the `router-outlet` element, which defines the location in which the component that corresponds to the current URL should be displayed. Listing 6-12 replaces the `store` element in the root component's template with the `router-outlet` element.

Listing 6-12. Defining the Routing Target in the app.component.ts File in the src/app Folder

```
import { Component } from "@angular/core";

@Component({
  selector: "app",
  template: "<router-outlet></router-outlet>"
})
export class AppComponent { }
```

Angular will apply the routing configuration when you save the changes and the browser reloads the HTML document. The content displayed in the browser window hasn't changed, but if you examine the browser's URL bar, you will be able to see that the routing configuration has been applied, as shown in Figure 6-2.

**Figure 6-2.** The effect of URL routing

Navigating Through the Application

With the routing configuration in place, it is time to add support for navigating between components by changing the browser's URL. The URL routing feature relies on a JavaScript API provided by the browser, which means the user can't simply type the target URL into the browser's URL bar. Instead, the navigation has to be performed by the application, either by using JavaScript code in a component or other building block or by adding attributes to HTML elements in the template.

When the user clicks one of the Add To Cart buttons, the cart detail component should be shown, which means that the application should navigate to the /cart URL. Listing 6-13 adds navigation to the component method that is invoked when the user clicks the button.

Listing 6-13. Navigating Using JavaScript in the store.component.ts File in the app/src/store Folder

```

import { Component } from "@angular/core";
import { Product } from "../model/product.model";
import { ProductRepository } from "../model/product.repository";
import { Cart } from "../model/cart.model";
import { Router } from "@angular/router";

@Component({
  selector: "store",
  templateUrl: "store.component.html"
})
export class StoreComponent {
  selectedCategory: string | undefined;
  productsPerPage = 4;
  selectedPage = 1;

  constructor(private repository: ProductRepository,
    private cart: Cart,
    private router: Router) { }

  get products(): Product[] {
    let pageIndex = (this.selectedPage - 1) * this.productsPerPage
    return this.repository.getProducts(this.selectedCategory)
      .slice(pageIndex, pageIndex + this.productsPerPage);
  }

  get categories(): string[] {
    return this.repository.getCategories();
  }

  changeCategory(newCategory?: string) {
    this.selectedCategory = newCategory;
  }

  changePage(newPage: number) {
    this.selectedPage = newPage;
  }

  changePageSize(newSize: number) {
    this.productsPerPage = Number(newSize);
    this.changePage(1);
  }

  get pageCount(): number {
    return Math.ceil(this.repository
      .getProducts(this.selectedCategory).length / this.productsPerPage)
  }
}

```

```

    addProductToCart(product: Product) {
      this.cart.addLine(product);
      this.router.navigateByUrl("/cart");
    }
}

```

The constructor has a Router parameter, which is provided by Angular through the dependency injection feature when a new instance of the component is created. In the addProductToCart method, the Router.navigateByUrl method is used to navigate to the /cart URL.

Navigation can also be done by adding the routerLink attribute to elements in the template. In Listing 6-14, the routerLink attribute has been applied to the cart button in the cart summary component's template.

Listing 6-14. Adding Navigation in the cartSummary.component.html File in the src/app/store Folder

```

<div class="float-end">
  <small class="fs-6">
    Your cart:
    <span *ngIf="cart.itemCount > 0">
      {{ cart.itemCount }} item(s)
      {{ cart.cartPrice | currency:"USD": "symbol": "2.2-2" }}
    </span>
    <span *ngIf="cart.itemCount == 0">
      (empty)
    </span>
  </small>
  <button class="btn btn-sm bg-dark text-white"
    [disabled]="cart.itemCount == 0" routerLink="/cart">
    <i class="fa fa-shopping-cart"></i>
  </button>
</div>

```

The value specified by the routerLink attribute is the URL that the application will navigate to when the button is clicked. This particular button is disabled when the cart is empty, so it will perform the navigation only when the user has added a product to the cart.

To add support for the routerLink attribute, the RouterModule module must be imported into the feature module, as shown in Listing 6-15.

Listing 6-15. Importing the Router Module in the store.module.ts File in the src/app/store Folder

```

import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule } from "@angular/forms";
import { ModelModule } from "../model/model.module";
import { StoreComponent } from "./store.component";
import { CounterDirective } from "./counter.directive";
import { CartSummaryComponent } from "./cartSummary.component";
import { CartDetailComponent } from "./cartDetail.component";
import { CheckoutComponent } from "./checkout.component";
import { RouterModule } from "@angular/router";

```

```
@NgModule({
  imports: [ModelModule, BrowserModule, FormsModule, RouterModule],
  declarations: [StoreComponent, CounterDirective, CartSummaryComponent,
    CartDetailComponent, CheckoutComponent],
  exports: [StoreComponent, CartDetailComponent, CheckoutComponent]
})
export class StoreModule { }
```

To see the effect of the navigation, save the changes of the files, and once the browser has reloaded the HTML document, click one of the Add To Cart buttons. The browser will navigate to the /cart URL, as shown in Figure 6-3.

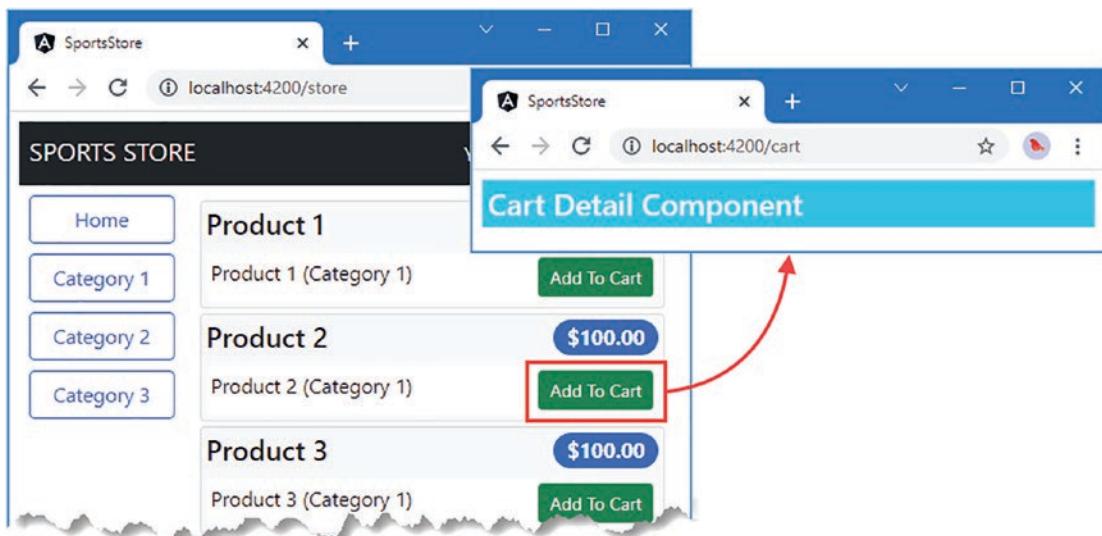


Figure 6-3. Using URL routing

Guarding the Routes

Remember that navigation can be performed only by the application. If you change the URL directly in the browser's URL bar, the browser will request the URL you enter from the web server. The Angular development server that is responding to HTTP requests will respond to any URL that doesn't correspond to a file by returning the contents of `index.html`. This is generally a useful behavior because it means you won't receive an HTTP error when the browser's reload button is clicked. But it can cause problems if the application expects the user to navigate through the application following a specific path.

As an example, if you click one of the Add To Cart buttons and then click the browser's reload button, the HTTP server will return the contents of the `index.html` file, and Angular will immediately jump to the cart detail component, skipping over the part of the application that allows the user to select products.

For some applications, being able to start using different URLs makes sense, but if that's not the case, then Angular supports *route guards*, which are used to govern the routing system.

To prevent the application from starting with the `/cart` or `/order` URL, I added a file called `storeFirst.guard.ts` in the `SportsStore/src/app` folder and defined the class shown in Listing 6-16.

Listing 6-16. The Contents of the storeFirst.guard.ts File in the src/app Folder

```

import { Injectable } from "@angular/core";
import {
  ActivatedRouteSnapshot, RouterStateSnapshot,
  Router
} from "@angular/router";
import { StoreComponent } from "./store/store.component";

@Injectable()
export class StoreFirstGuard {
  private firstNavigation = true;

  constructor(private router: Router) { }

  canActivate(route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot): boolean {
    if (this.firstNavigation) {
      this.firstNavigation = false;
      if (route.component != StoreComponent) {
        this.router.navigateByUrl("/");
        return false;
      }
    }
    return true;
  }
}

```

There are different ways to guard routes, as described in Chapter 26, and this is an example of a guard that prevents a route from being activated, which is implemented as a class that defines a `canActivate` method. The implementation of this method uses the context objects that Angular provides that describe the route that is about to be navigated to and checks to see whether the target component is a `StoreComponent`. If this is the first time that the `canActivate` method has been called and a different component is about to be used, then the `Router.navigateByUrl` method is used to navigate to the root URL.

The `@Injectable` decorator has been applied in the listing because route guards are services. Listing 6-17 registers the guard as a service using the root module's `providers` property and guards each route using the `canActivate` property.

Listing 6-17. Guarding Routes in the app.module.ts File in the src/app Folder

```

import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';
import { StoreModule } from "./store/store.module";

import { StoreComponent } from "./store/store.component";
import { CheckoutComponent } from "./store/checkout.component";
import { CartDetailComponent } from "./store/cartDetail.component";
import { RouterModule } from "@angular/router";
import { StoreFirstGuard } from "./storeFirst.guard";

```

```

@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule, StoreModule,
    RouterModule.forRoot([
      {
        path: "store", component: StoreComponent,
        canActivate: [StoreFirstGuard]
      },
      {
        path: "cart", component: CartDetailComponent,
        canActivate: [StoreFirstGuard]
      },
      {
        path: "checkout", component: CheckoutComponent,
        canActivate: [StoreFirstGuard]
      },
      { path: "**", redirectTo: "/store" }
    ]),
    providers: [StoreFirstGuard],
    bootstrap: [AppComponent]
  })
export class AppModule { }

```

If you reload the browser after clicking one of the Add To Cart buttons now, then you will see the browser is automatically directed back to safety, as shown in Figure 6-4.

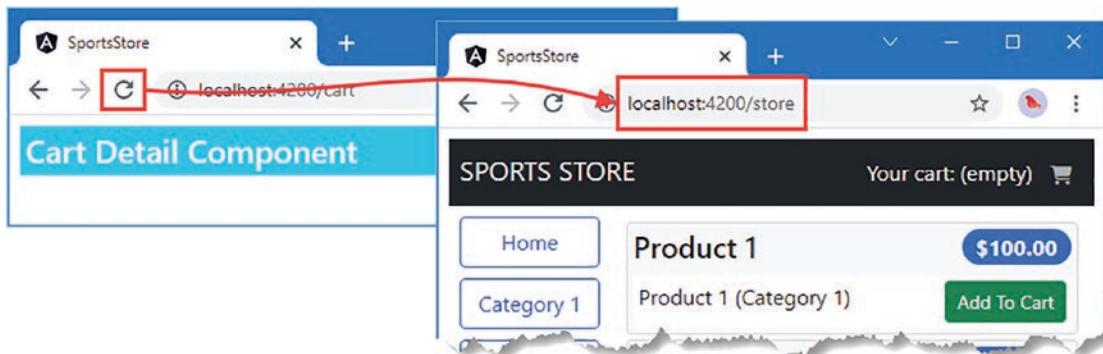


Figure 6-4. Guarding routes

Completing the Cart Detail Feature

Now that the application has navigation support, it is time to complete the view that details the contents of the user's cart. Listing 6-18 removes the inline template from the cart detail component, specifies an external template in the same directory, and adds a `Cart` parameter to the constructor, which will be accessible in the template through a property called `cart`.

Listing 6-18. Changing the Template in the cartDetail.component.ts File in the src/app/store Folder

```
import { Component } from "@angular/core";
import { Cart } from "../model/cart.model";

@Component({
  templateUrl: "cartDetail.component.html"
})
export class CartDetailComponent {

  constructor(public cart: Cart) { }
}
```

To complete the cart detail feature, I created an HTML file called `cartDetail.component.html` in the `src/app/store` folder and added the content shown in Listing 6-19.

Listing 6-19. The Contents of the cartDetail.component.html File in the src/app/store Folder

```
<div class="container-fluid">
  <div class="row">
    <div class="bg-dark text-white p-2">
      <span class="navbar-brand ml-2">SPORTS STORE</span>
    </div>
  </div>
  <div class="row">
    <div class="col mt-2">
      <h2 class="text-center">Your Cart</h2>
      <table class="table table-bordered table-striped p-2">
        <thead>
          <tr>
            <th>Quantity</th>
            <th>Product</th>
            <th class="text-end">Price</th>
            <th class="text-end">Subtotal</th>
          </tr>
        </thead>
        <tbody>
          <tr *ngIf="cart.lines.length == 0">
            <td colspan="4" class="text-center">
              Your cart is empty
            </td>
          </tr>
          <tr *ngFor="let line of cart.lines">
            <td>
              <input type="number" class="form-control-sm"
                style="width:5em" [value]="line.quantity"
                (change)="cart.updateQuantity(line.product,
                  $any($event).target.value)" />
            </td>
          </tr>
        </tbody>
      </table>
    </div>
  </div>
</div>
```

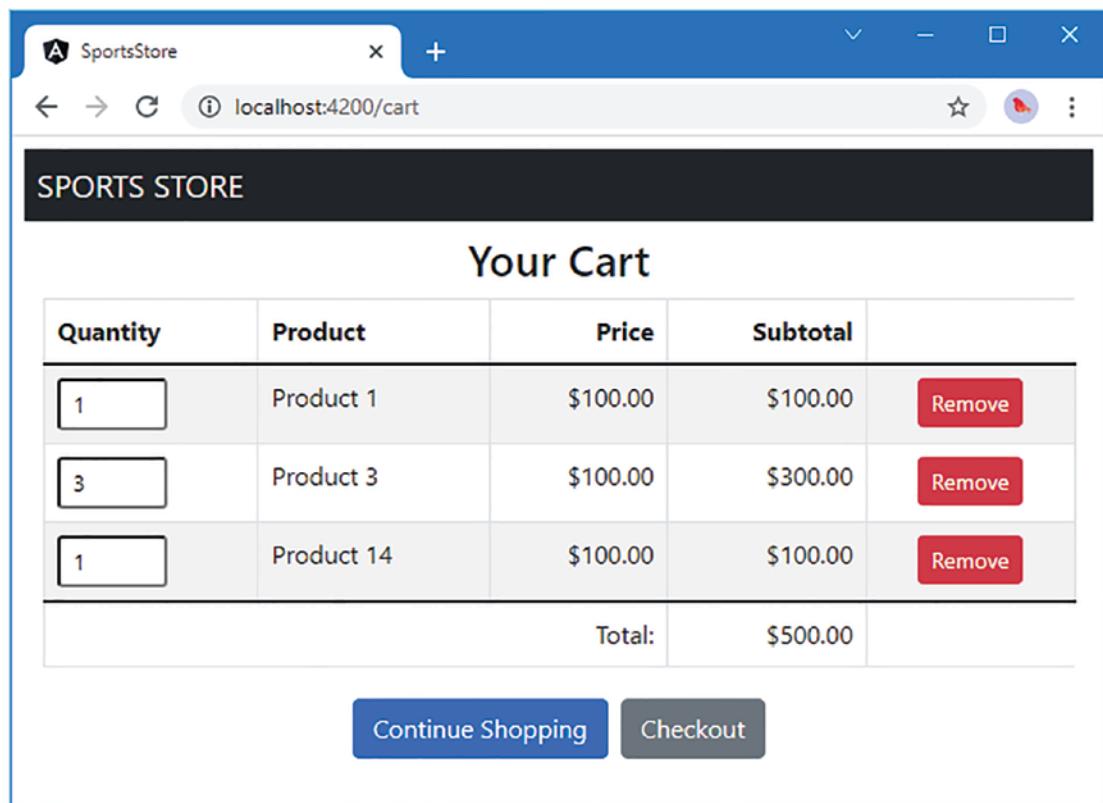
```

        <td>{{line.product.name}}</td>
        <td class="text-end">
            {{line.product.price | currency:"USD":"symbol":"2.2-2"}}
        </td>
        <td class="text-end">
            {{(line.lineTotal) | currency:"USD":"symbol":"2.2-2" }}
        </td>
        <td class="text-center">
            <button class="btn btn-sm btn-danger"
                   (click)="cart.removeLine(line.product.id ?? 0)">
                Remove
            </button>
        </td>
    </tr>
</tbody>
<tfoot>
    <tr>
        <td colspan="3" class="text-end">Total:</td>
        <td class="text-end">
            {{cart.cartPrice | currency:"USD":"symbol":"2.2-2"}}
        </td>
    </tr>
</tfoot>
</table>
</div>
</div>
<div class="row">
    <div class="col">
        <div class="text-center">
            <button class="btn btn-primary m-1" routerLink="/store">
                Continue Shopping
            </button>
            <button class="btn btn-secondary m-1" routerLink="/checkout"
                   [disabled]="cart.lines.length == 0">
                Checkout
            </button>
        </div>
    </div>
</div>
</div>

```

This template displays a table showing the user's product selections. For each product, there is an input element that can be used to change the quantity, and there is a Remove button that deletes it from the cart. There are also two navigation buttons that allow the user to return to the list of products or continue to the checkout process, as shown in Figure 6-5. The combination of the Angular data bindings and the shared Cart object means that any changes made to the cart take immediate effect, recalculating the prices; and if you click the Continue Shopping button, the changes are reflected in the cart summary component shown above the list of products.

■ Tip If you receive an error after you have saved the template, then stop and restart the `ng serve` command.



Listing 6-20. The Contents of the order.model.ts File in the src/app/model Folder

```
import { Injectable } from "@angular/core";
import { Cart } from "./cart.model";

@Injectable()
export class Order {
    public id?: number;
    public name?: string;
    public address?: string;
    public city?: string;
    public state?: string;
    public zip?: string;
    public country?: string;
    public shipped: boolean = false;

    constructor(public cart: Cart) { }

    clear() {
        this.id = undefined;
        this.name = this.address = this.city = undefined;
        this.state = this.zip = this.country = undefined;
        this.shipped = false;
        this.cart.clear();
    }
}
```

The Order class will be another service, which means there will be one instance shared throughout the application. When Angular creates the Order object, it will detect the Cart constructor parameter and provide the same Cart object that is used elsewhere in the application.

Updating the Repository and Data Source

To handle orders in the application, I need to extend the repository and the data source so they can receive Order objects. Listing 6-21 adds a method to the data source that receives an order. Since this is still the dummy data source, the method simply produces a JSON string from the order and writes it to the JavaScript console. I'll do something more useful with the objects in the next section when I create a data source that uses HTTP requests to communicate with the RESTful web service.

Listing 6-21. Handling Orders in the static.datasource.ts File in the src/app/model Folder

```
import { Injectable } from "@angular/core";
import { Product } from "./product.model";
import { Observable, from } from "rxjs";
import { Order } from "./order.model";

@Injectable()
export class StaticDataSource {
    private products: Product[] = [
        new Product(1, "Product 1", "Category 1", "Product 1 (Category 1)", 100),
        new Product(2, "Product 2", "Category 1", "Product 2 (Category 1)", 100),
```

```

        new Product(3, "Product 3", "Category 1", "Product 3 (Category 1)", 100),
        new Product(4, "Product 4", "Category 1", "Product 4 (Category 1)", 100),
        new Product(5, "Product 5", "Category 1", "Product 5 (Category 1)", 100),
        new Product(6, "Product 6", "Category 2", "Product 6 (Category 2)", 100),
        new Product(7, "Product 7", "Category 2", "Product 7 (Category 2)", 100),
        new Product(8, "Product 8", "Category 2", "Product 8 (Category 2)", 100),
        new Product(9, "Product 9", "Category 2", "Product 9 (Category 2)", 100),
        new Product(10, "Product 10", "Category 2", "Product 10 (Category 2)", 100),
        new Product(11, "Product 11", "Category 3", "Product 11 (Category 3)", 100),
        new Product(12, "Product 12", "Category 3", "Product 12 (Category 3)", 100),
        new Product(13, "Product 13", "Category 3", "Product 13 (Category 3)", 100),
        new Product(14, "Product 14", "Category 3", "Product 14 (Category 3)", 100),
        new Product(15, "Product 15", "Category 3", "Product 15 (Category 3)", 100),
    ];
}

getProducts(): Observable<Product[]> {
    return from([this.products]);
}

saveOrder(order: Order): Observable<Order> {
    console.log(JSON.stringify(order));
    return from([order]);
}
}
}

```

To manage orders, I added a file called `order.repository.ts` to the `src/app/model` folder and used it to define the class shown in Listing 6-22. There is only one method in the order repository at the moment, but I will add more functionality in Chapter 7 when I create the administration features.

Tip You don't have to use different repositories for each model type in the application, but I often do so because a single class responsible for multiple model types can become complex and difficult to maintain.

Listing 6-22. The Contents of the `order.repository.ts` File in the `src/app/model` Folder

```

import { Injectable } from "@angular/core";
import { Observable } from "rxjs";
import { Order } from "./order.model";
import { StaticDataSource } from "./static.datasource";

@Injectable()
export class OrderRepository {
    private orders: Order[] = [];

    constructor(private dataSource: StaticDataSource) {}

    getOrders(): Order[] {
        return this.orders;
    }
}

```

```

    saveOrder(order: Order): Observable<Order> {
      return this.dataSource.saveOrder(order);
    }
}

```

Updating the Feature Module

Listing 6-23 registers the Order class and the new repository as services using the providers property of the model feature module.

Listing 6-23. Registering Services in the model.module.ts File in the src/app/model Folder

```

import { NgModule } from "@angular/core";
import { ProductRepository } from "./product.repository";
import { StaticDataSource } from "./static.datasource";
import { Cart } from "./cart.model";
import { Order } from "./order.model";
import { OrderRepository } from "./order.repository";

@NgModule({
  providers: [ProductRepository, StaticDataSource, Cart, Order, OrderRepository]
})
export class ModelModule { }

```

Collecting the Order Details

The next step is to gather the details from the user required to complete the order. Angular includes built-in directives for working with HTML forms and validating their contents. Listing 6-24 prepares the checkout component, switching to an external template, receiving the Order object as a constructor parameter, and providing some additional support to help the template.

Listing 6-24. Preparing for a Form in the checkout.component.ts File in the src/app/store Folder

```

import { Component } from "@angular/core";
import { NgForm } from "@angular/forms";
import { OrderRepository } from "../model/order.repository";
import { Order } from "../model/order.model";

@Component({
  templateUrl: "checkout.component.html",
  styleUrls: ["checkout.component.css"]
})
export class CheckoutComponent {
  orderSent: boolean = false;
  submitted: boolean = false;

  constructor(public repository: OrderRepository,
    public order: Order) {}
}

```

```

    submitOrder(form: NgForm) {
        this.submitted = true;
        if (form.valid) {
            this.repository.saveOrder(this.order).subscribe(order => {
                this.order.clear();
                this.orderSent = true;
                this.submitted = false;
            });
        }
    }
}

```

The `submitOrder` method will be invoked when the user submits a form, which is represented by an `NgForm` object. If the data that the form contains is valid, then the `Order` object will be passed to the repository's `saveOrder` method, and the data in the cart and the order will be reset.

The `@Component` decorator's `styleUrls` property is used to specify one or more CSS stylesheets that should be applied to the content in the component's template. To provide validation feedback for the values that the user enters into the HTML form elements, I created a file called `checkout.component.css` in the `src/app/store` folder and defined the styles shown in Listing 6-25.

Listing 6-25. The Contents of the `checkout.component.css` File in the `src/app/store` Folder

```
input.ng-dirty.ng-invalid { border: 2px solid #ff0000 }
input.ng-dirty.ng-valid { border: 2px solid #6bc502 }
```

Angular adds elements to the `ng-dirty`, `ng-valid`, and `ng-invalid` classes to indicate their validation status. The full set of validation classes is described in Chapter 12, but the effect of the styles in Listing 6-25 is to add a green border around `input` elements that are valid and a red border around those that are invalid.

The final piece of the puzzle is the template for the component, which presents the user with the form fields required to populate the properties of an `Order` object, as shown in Listing 6-26.

Listing 6-26. The Contents of the `checkout.component.html` File in the `src/app/store` Folder

```

<div class="container-fluid">
    <div class="row">
        <div class="bg-dark text-white p-2">
            <span class="navbar-brand ml-2">SPORTS STORE</span>
        </div>
    </div>
</div>

<div *ngIf="orderSent" class="m-2 text-center">
    <h2>Thanks!</h2>
    <p>Thanks for placing your order.</p>
    <p>We'll ship your goods as soon as possible.</p>
    <button class="btn btn-primary" routerLink="/store">Return to Store</button>
</div>
<form *ngIf="!orderSent" #form="ngForm" novalidate
      (ngSubmit)="submitOrder(form)" class="m-2">
    <div class="form-group">
        <label>Name</label>
        <input class="form-control" #name="ngModel" name="name">
    </div>
</form>

```

```
[(ngModel)]="order.name" required />
<span *ngIf="submitted && name.invalid" class="text-danger">
  Please enter your name
</span>
</div>
<div class="form-group">
  <label>Address</label>
  <input class="form-control" #address="ngModel" name="address"
    [(ngModel)]="order.address" required />
  <span *ngIf="submitted && address.invalid" class="text-danger">
    Please enter your address
  </span>
</div>
<div class="form-group">
  <label>City</label>
  <input class="form-control" #city="ngModel" name="city"
    [(ngModel)]="order.city" required />
  <span *ngIf="submitted && city.invalid" class="text-danger">
    Please enter your city
  </span>
</div>
<div class="form-group">
  <label>State</label>
  <input class="form-control" #state="ngModel" name="state"
    [(ngModel)]="order.state" required />
  <span *ngIf="submitted && state.invalid" class="text-danger">
    Please enter your state
  </span>
</div>
<div class="form-group">
  <label>Zip/Postal Code</label>
  <input class="form-control" #zip="ngModel" name="zip"
    [(ngModel)]="order.zip" required />
  <span *ngIf="submitted && zip.invalid" class="text-danger">
    Please enter your zip/postal code
  </span>
</div>
<div class="form-group">
  <label>Country</label>
  <input class="form-control" #country="ngModel" name="country"
    [(ngModel)]="order.country" required />
  <span *ngIf="submitted && country.invalid" class="text-danger">
    Please enter your country
  </span>
</div>
<div class="text-center">
  <button class="btn btn-secondary m-1" routerLink="/cart">Back</button>
  <button class="btn btn-primary m-1" type="submit">Complete Order</button>
</div>
</form>
```

The form and input elements in this template use Angular features to ensure that the user provides values for each field, and they provide visual feedback if the user clicks the Complete Order button without completing the form. Part of this feedback comes from applying the styles that were defined in Listing 6-25, and part comes from span elements that remain hidden until the user tries to submit an invalid form.

Tip Requiring values is only one of the ways that Angular can validate form fields, and as I explained in Chapter 12, you can easily add your own custom validation as well.

To see the process, start with the list of products and click one of the Add To Cart buttons to add a product to the cart. Click the Checkout button, and you will see the HTML form shown in Figure 6-6. Click the Complete Order button without entering text into any of the input elements, and you will see the validation feedback messages. Fill out the form and click the Complete Order button; you will see the confirmation message shown in the figure.

Tip Restart the `ng serve` command if you see an error after saving the template.

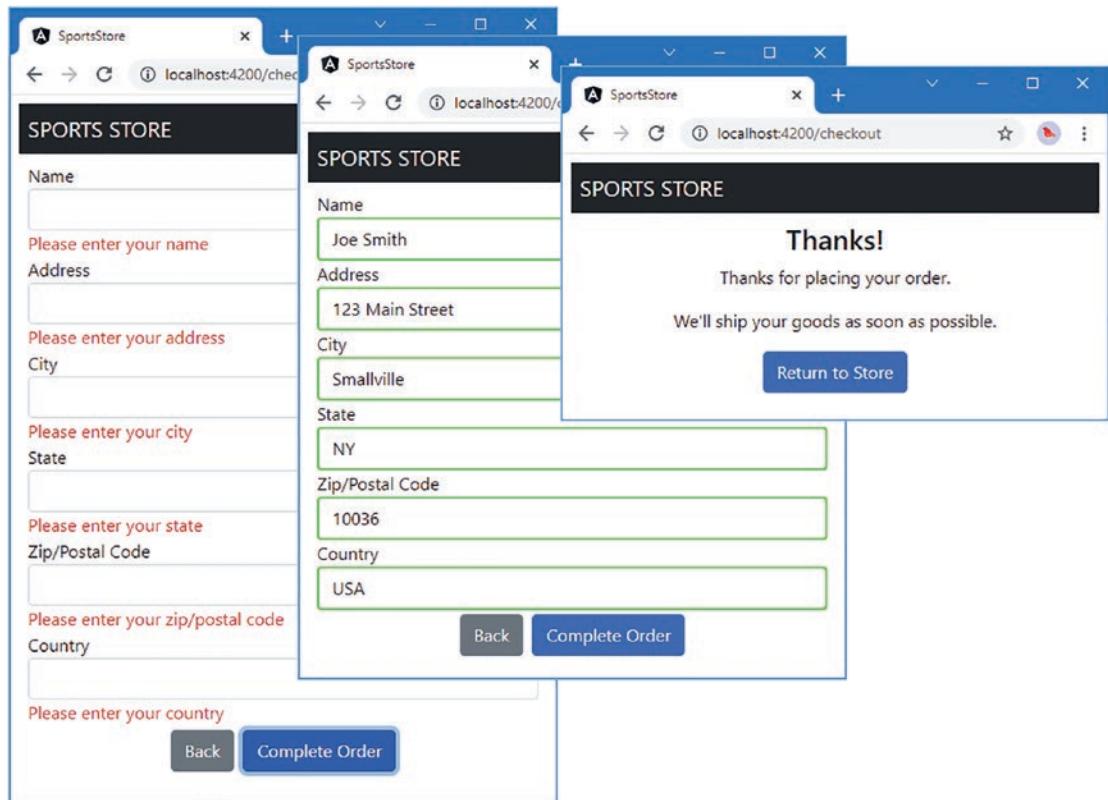


Figure 6-6. Completing an order

If you look at the browser's JavaScript console, you will see a JSON representation of the order like this:

```
{"cart":  
  {"lines": [  
    {"product": {"id": 1, "name": "Product 1", "category": "Category 1",  
      "description": "Product 1 (Category 1)", "price": 100, "quantity": 1},  
      "itemCount": 1, "cartPrice": 100},  
    "shipped": false,  
    "name": "Joe Smith", "address": "123 Main Street",  
    "city": "Smallville", "state": "NY", "zip": "10036", "country": "USA"  
  ]}  
}
```

Using the RESTful Web Service

Now that the basic SportsStore functionality is in place, it is time to replace the dummy data source with one that gets its data from the RESTful web service that was created during the project setup in Chapter 5.

To create the data source, I added a file called `rest.datasource.ts` in the `src/app/model` folder and added the code shown in Listing 6-27.

Listing 6-27. The Contents of the `rest.datasource.ts` File in the `src/app/model` Folder

```
import { Injectable } from "@angular/core";
import { HttpClient } from "@angular/common/http";
import { Observable } from "rxjs";
import { Product } from "./product.model";
import { Order } from "./order.model";

const PROTOCOL = "http";
const PORT = 3500;

@Injectable()
export class RestDataSource {
  baseUrl: string;

  constructor(private http: HttpClient) {
    this.baseUrl = `${PROTOCOL}://${location.hostname}:${PORT}/`;
  }

  getProducts(): Observable<Product[]> {
    return this.http.get<Product[]>(this.baseUrl + "products");
  }

  saveOrder(order: Order): Observable<Order> {
    return this.http.post<Order>(this.baseUrl + "orders", order);
  }
}
```

Angular provides a built-in service called `HttpClient` that is used to make HTTP requests. The `RestDataSource` constructor receives the `HttpClient` service and uses the global `location` object provided by the browser to determine the URL that the requests will be sent to, which is port 3500 on the same host that the application has been loaded from.

The methods defined by the `RestDataSource` class correspond to the ones defined by the static data source but are implemented using the `HttpClient` service, described in Chapter 23.

Tip When obtaining data via HTTP, it is possible that network congestion or server load will delay the request and leave the user looking at an application that has no data. In Chapter 26, I explain how to configure the routing system to prevent this problem.

Applying the Data Source

To complete this chapter, I am going to apply the RESTful data source by reconfiguring the application so that the switch from the dummy data to the REST data is done with changes to a single file. Listing 6-28 changes the behavior of the data source service in the model feature module.

Listing 6-28. Changing the Service Configuration in the `model.module.ts` File in the `src/app/model` Folder

```
import { NgModule } from "@angular/core";
import { ProductRepository } from "./product.repository";
import { StaticDataSource } from "./static.datasource";
import { Cart } from "./cart.model";
import { Order } from "./order.model";
import { OrderRepository } from "./order.repository";
import { RestDataSource } from "./rest.datasource";
import { HttpClientModule } from "@angular/common/http";

@NgModule({
  imports: [HttpClientModule],
  providers: [ProductRepository, StaticDataSource, Cart, Order, OrderRepository,
    { provide: StaticDataSource, useClass: RestDataSource }]
})
export class ModelModule { }
```

The `imports` property is used to declare a dependency on the `HttpClientModule` feature module, which provides the `HttpClient` service used in Listing 6-27. The change to the `providers` property tells Angular that when it needs to create an instance of a class with a `StaticDataSource` constructor parameter, it should use a `RestDataSource` instead. Since both objects define the same methods, the dynamic JavaScript type system means that the substitution is seamless. When all the changes have been saved and the browser reloads the application, you will see the dummy data has been replaced with the data obtained via HTTP, as shown in Figure 6-7.

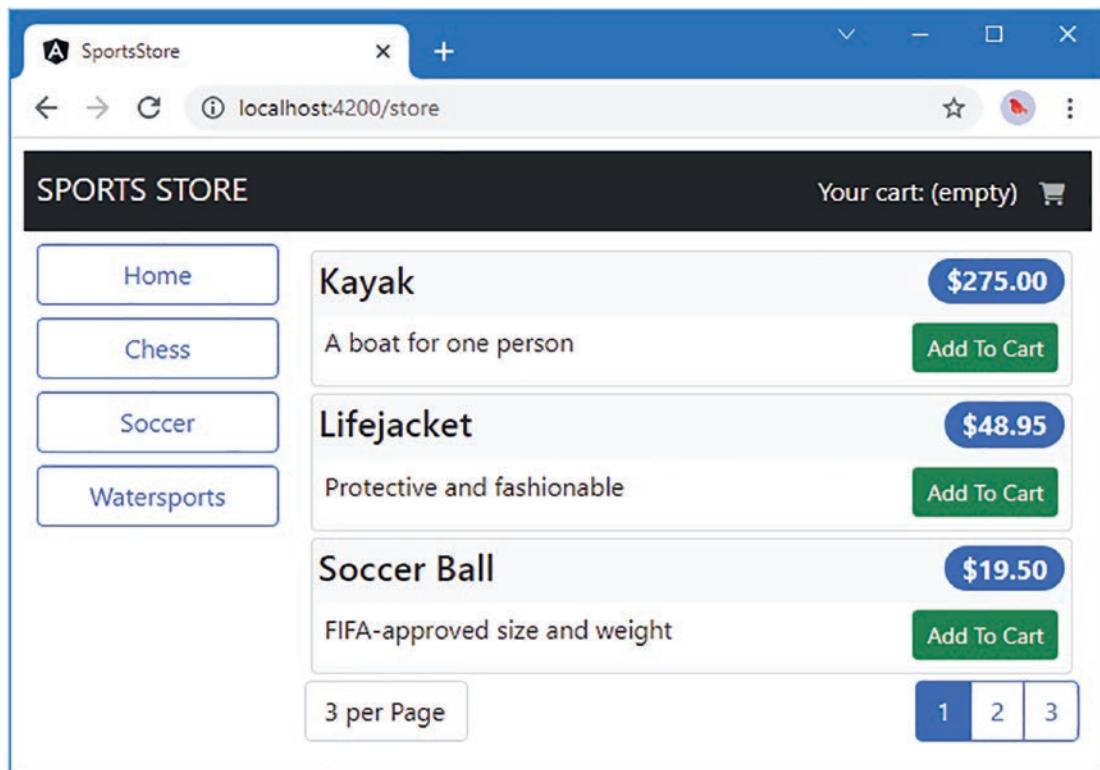


Figure 6-7. Using the RESTful web service

If you go through the process of selecting products and checking out, you can see that the data source has written the order to the web service by navigating to this URL:

`http://localhost:3500/db`

This will display the full contents of the database, including the collection of orders. You won't be able to request the `/orders` URL because it requires authentication, which I set up in the next chapter.

■ **Tip** Remember that the data provided by the RESTful web service is reset when you stop the server and start it again using the `npm run json` command.

Summary

In this chapter, I continued adding features to the SportsStore application, adding support for a shopping cart into which the user can place products and a checkout process that completes the shopping process. To complete the chapter, I replaced the dummy data source with one that sends HTTP requests to the RESTful web service. In the next chapter, I create administration features that allow the SportsStore data to be managed.

CHAPTER 7



SportsStore: Administration

In this chapter, I continue building the SportsStore application by adding administration features. Relatively few users will need to access the administration features, so it would be wasteful to force all users to download the administration code and content when it is unlikely to be used. Instead, I am going to put the administration features in a separate module that will be loaded only when it is required.

Preparing the Example Application

No preparation is required for this chapter, which continues using the SportsStore project from Chapter 6. To start the RESTful web service, open a command prompt and run the following command in the `SportsStore` folder:

```
npm run json
```

Open a second command prompt and run the following command in the `SportsStore` folder to start the development tools and HTTP server:

```
ng serve --open
```

Tip You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/Apress/pro-angular-5ed>. See Chapter 1 for how to get help if you have problems running the examples.

Creating the Module

The process for creating the feature module follows the same pattern you have seen in earlier chapters. The key difference is that it is important that no other part of the application has dependencies on the module or the classes it contains, which would undermine the dynamic loading of the module and cause the JavaScript module to load the administration code, even if it is not used.

The starting point for the administration features will be authentication, which will ensure that only authorized users can administer the application. I created a file called `auth.component.ts` in the `src/app/admin` folder and used it to define the component shown in Listing 7-1.

Listing 7-1. The Content of the `auth.component.ts` File in the `src/app/admin` Folder

```
import { Component } from "@angular/core";
import { NgForm } from "@angular/forms";
import { Router } from "@angular/router";

@Component({
  templateUrl: "auth.component.html"
})
export class AuthComponent {
  username?: string;
  password?: string;
  errorMessage?: string;

  constructor(private router: Router) {}

  authenticate(form: NgForm) {
    if (form.valid) {
      // perform authentication
      this.router.navigateByUrl("/admin/main");
    } else {
      this.errorMessage = "Form Data Invalid";
    }
  }
}
```

The component defines properties for the `username` and `password` that will be used to authenticate the user, an `errorMessage` property that will be used to display messages when there are problems, and an `authenticate` method that will perform the authentication process (but that does nothing at the moment).

To provide the component with a template, I created a file called `auth.component.html` in the `src/app/admin` folder and added the content shown in Listing 7-2.

Listing 7-2. The Content of the `auth.component.html` File in the `src/app/admin` Folder

```
<div class="bg-info p-2 text-center text-white">
  <h3>SportsStore Admin</h3>
</div>
<div class="bg-danger mt-2 p-2 text-center text-white" *ngIf="errorMessage != null">
  {{errorMessage}}
</div>
<div class="p-2">
  <form novalidate #form="ngForm" (ngSubmit)="authenticate(form)">
    <div class="form-group">
      <label>Name</label>
      <input class="form-control" name="username"
        [(ngModel)]="username" required />
    </div>
  </form>
</div>
```

```

<div class="form-group">
    <label>Password</label>
    <input class="form-control" type="password" name="password"
        [(ngModel)]="password" required />
</div>
<div class="text-center p-2">
    <button class="btn btn-secondary m-1" routerLink="/">Go back</button>
    <button class="btn btn-primary m-1" type="submit">Log In</button>
</div>
</form>
</div>

```

The template contains an HTML form that uses two-way data binding expressions for the component's properties. There is a button that will submit the form, a button that navigates back to the root URL, and a div element that is visible only when there is an error message to display.

To create a placeholder for the administration features, I added a file called `admin.component.ts` in the `src/app/admin` folder and defined the component shown in Listing 7-3.

Listing 7-3. The Contents of the `admin.component.ts` File in the `src/app/admin` Folder

```

import { Component } from "@angular/core";

@Component({
    templateUrl: "admin.component.html"
})
export class AdminComponent {}

```

The component doesn't contain any functionality at the moment. To provide a template for the component, I added a file called `admin.component.html` to the `src/app/admin` folder and the placeholder content shown in Listing 7-4.

Listing 7-4. The Contents of the `admin.component.html` File in the `src/app/admin` Folder

```

<div class="bg-info p-2 text-white">
    <h3>Placeholder for Admin Features</h3>
</div>

```

To define the feature module, I added a file called `admin.module.ts` in the `src/app/admin` folder and added the code shown in Listing 7-5.

Listing 7-5. The Contents of the `admin.module.ts` File in the `src/app/admin` Folder

```

import { NgModule } from "@angular/core";
import { CommonModule } from "@angular/common";
import { FormsModule } from "@angular/forms";
import { RouterModule } from "@angular/router";
import { AuthComponent } from "./auth.component";
import { AdminComponent } from "./admin.component";

```

```

let routing = RouterModule.forChild([
  { path: "auth", component: AuthComponent },
  { path: "main", component: AdminComponent },
  { path: "**", redirectTo: "auth" }
]);

@NgModule({
  imports: [CommonModule, FormsModule, routing],
  declarations: [AuthComponent, AdminComponent]
})
export class AdminModule { }

```

The `RouterModule.forChild` method is used to define the routing configuration for the feature module, which is then included in the module's `imports` property.

A dynamically loaded module must be self-contained and include all the information that Angular requires, including the routing URLs that are supported and the components they display. If any other part of the application depends on the module, then it will be included in the JavaScript bundle with the rest of the application code, which means that all users will have to download code and resources for features they won't use.

However, a dynamically loaded module is allowed to declare dependencies on the main part of the application. This module relies on the functionality in the data model module, which has been added to the module's `imports` so that components can access the model classes and the repositories.

Configuring the URL Routing System

Dynamically loaded modules are managed through the routing configuration, which triggers the loading process when the application navigates to a specific URL. Listing 7-6 extends the routing configuration of the application so that the `/admin` URL will load the administration feature module.

Listing 7-6. Configuring a Dynamically Loaded Module in the `app.module.ts` File in the `src/app` Folder

```

import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';
import { StoreModule } from './store/store.module';
import { StoreComponent } from './store/store.component';
import { CheckoutComponent } from './store/checkout.component';
import { CartDetailComponent } from './store/cartDetail.component';
import { RouterModule } from '@angular/router';
import { StoreFirstGuard } from './storeFirst.guard';

@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule, StoreModule,
    RouterModule.forRoot([
      {
        path: "store", component: StoreComponent,
        canActivate: [StoreFirstGuard]
      },
    ]),
  ],
  providers: []
})
export class AppModule { }

```

```

    {
      path: "cart", component: CartDetailComponent,
      canActivate: [StoreFirstGuard]
    },
    {
      path: "checkout", component: CheckoutComponent,
      canActivate: [StoreFirstGuard]
    },
    {
      path: "admin",
      loadChildren: () => import("./admin/admin.module")
        .then(m => m.AdminModule),
      canActivate: [StoreFirstGuard]
    },
    { path: "**", redirectTo: "/store" }
  ]),
  providers: [StoreFirstGuard],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

The new route tells Angular that when the application navigates to the /admin URL, it should load a feature module defined by a class called AdminModule from the admin/admin.module.ts file, whose path is specified relative to the app.module.ts file. When Angular processes the admin module, it will incorporate the routing information it contains into the overall set of routes and complete the navigation.

Navigating to the Administration URL

The final preparatory step is to provide the user with the ability to navigate to the /admin URL so that the administration feature module will be loaded and its component displayed to the user. Listing 7-7 adds a button to the store component's template that will perform the navigation.

Listing 7-7. Adding a Navigation Button in the store.component.html File in the src/app/store Folder

```

...
<div class="d-grid gap-2">
  <button class="btn btn-outline-primary" (click)="changeCategory()">
    Home
  </button>
  <button *ngFor="let cat of categories"
    class="btn btn-outline-primary"
    [class.active]="cat == selectedCategory"
    (click)="changeCategory(cat)">
    {{cat}}
  </button>
  <button class="btn btn-danger mt-5" routerLink="/admin">
    Admin
  </button>
</div>
...

```

To reflect the changes, stop the development tools and restart them by running the following command in the SportsStore folder:

```
ng serve
```

Use the browser to navigate to `http://localhost:4200` and use the browser's F12 developer tools to see the network requests made by the browser as the application is loaded. The files for the administration module will not be loaded until you click the Admin button, at which point Angular will request the files and display the login page shown in Figure 7-1.

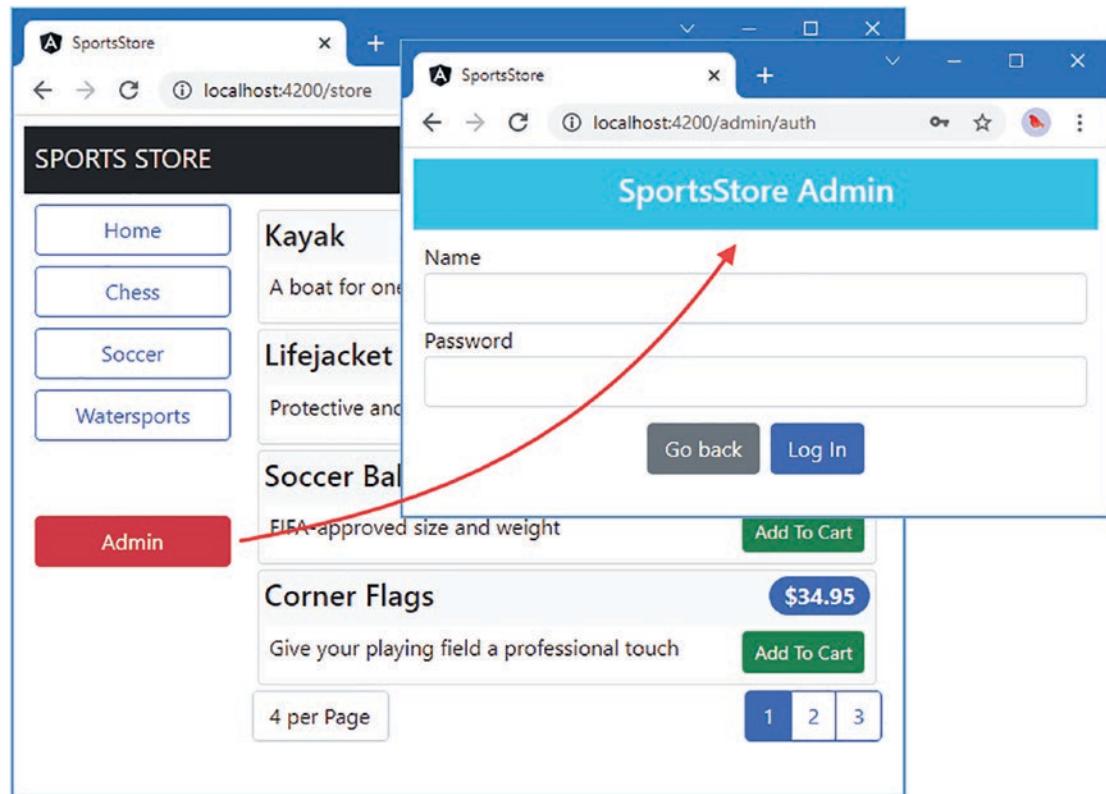


Figure 7-1. Using a dynamically loaded module

Enter any name and password into the form fields and click the Log In button to see the placeholder content, as shown in Figure 7-2. If you leave either of the form fields empty, a warning message will be displayed.

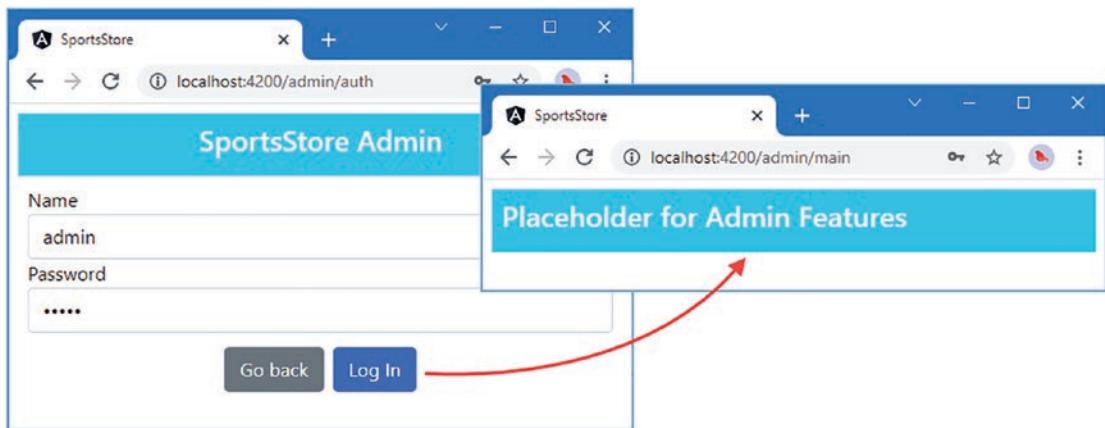


Figure 7-2. The placeholder administration features

Implementing Authentication

The RESTful web service has been configured so that it requires authentication for the requests that the administration feature will require. In the sections that follow, I add support for authenticating the user by sending an HTTP request to the RESTful web service.

Understanding the Authentication System

When the RESTful web service authenticates a user, it will return a JSON Web Token (JWT) that the application must include in subsequent HTTP requests to show that authentication has been successfully performed. You can read the JWT specification at <https://tools.ietf.org/html/rfc7519>, but for the SportsStore application, it is enough to know that the Angular application can authenticate the user by sending a POST request to the /login URL, including a JSON-formatted object in the request body that contains name and password properties. There is only one set of valid credentials in the authentication code I added to the application in Chapter 5, which is shown in Table 7-1.

Table 7-1. The Authentication Credentials Supported by the RESTful Web Service

Username	Password
admin	secret

As I noted in Chapter 5, you should not hard-code credentials in real projects, but this is the username and password that you will need for the SportsStore application.

If the correct credentials are sent to the `/login` URL, then the response from the RESTful web service will contain a JSON object like this:

```
{
  "success": true,
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJkYXRhIjoiYWRtaW4iLCJleHBpcmVz
  SW4i0iIxaCIsImlhdCI6MTQ3ODk1NjI1Mno.1JaDDrSu-bHBtdWrz0312p_DG5tKypGv6cA
  NgOyzlg8"
}
```

The `success` property describes the outcome of the authentication operation, and the `token` property contains the JWT, which should be included in subsequent requests using the `Authorization` HTTP header in this format:

```
Authorization: Bearer<eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJkYXRhIjoiYWRtaW4iLC
  JleHBpcmVzSW4i0iIxaCIsImlhdCI6MTQ3ODk1NjI1Mno.1JaDDrSu-
  bHBtdWrz0312p_DG5tKypGv6cANgOyzlg8>
```

I configured the JWT tokens returned by the server so they expire after one hour. If the wrong credentials are sent to the server, then the JSON object returned in the response will just contain a `success` property set to `false`, like this:

```
{
  "success": false
}
```

Extending the Data Source

The RESTful data source will do most of the work because it is responsible for sending the authentication request to the `/login` URL and including the JWT in subsequent requests. Listing 7-8 adds authentication to the `RestDataSource` class and defines a variable that will store the JWT once it has been obtained.

Listing 7-8. Adding Authentication in the `rest.datasource.ts` File in the `src/app/model` Folder

```
import { Injectable } from "@angular/core";
import { HttpClient } from "@angular/common/http";
import { map, Observable } from "rxjs";
import { Product } from "./product.model";
import { Order } from "./order.model";

const PROTOCOL = "http";
const PORT = 3500;

@Injectable()
export class RestDataSource {
  baseUrl: string;
  auth_token?: string;
```

```

constructor(private http: HttpClient) {
    this.baseUrl = `${PROTOCOL}://${location.hostname}:${PORT}/`;
}

getProducts(): Observable<Product[]> {
    return this.http.get<Product[]>(this.baseUrl + "products");
}

saveOrder(order: Order): Observable<Order> {
    return this.http.post<Order>(this.baseUrl + "orders", order);
}

authenticate(user: string, pass: string): Observable<boolean> {
    return this.http.post<any>(this.baseUrl + "login", {
        name: user, password: pass
    }).pipe(map(response => {
        this.auth_token = response.success ? response.token : null;
        return response.success;
    }));
}
}
}

```

The pipe method and map function are provided by the RxJS package, and they allow the response event from the server, which is presented through an `Observable<any>` to be transformed into an event in the `Observable<bool>` that is the result of the authenticate method.

Creating the Authentication Service

Rather than expose the data source directly to the rest of the application, I am going to create a service that can be used to perform authentication and determine whether the application has been authenticated. I added a file called `auth.service.ts` in the `src/app/model` folder and added the code shown in Listing 7-9.

Listing 7-9. The Contents of the `auth.service.ts` File in the `src/app/model` Folder

```

import { Injectable } from "@angular/core";
import { Observable } from "rxjs";
import { RestDataSource } from "./rest.datasource";

@Injectable()
export class AuthService {

    constructor(private datasource: RestDataSource) {}

    authenticate(username: string, password: string): Observable<boolean> {
        return this.datasource.authenticate(username, password);
    }

    get authenticated(): boolean {
        return this.datasource.auth_token != null;
    }
}

```

```

    clear() {
      this.datasource.auth_token = undefined;
    }
}

```

The authenticate method receives the user's credentials and passes them on to the data source authenticate method, returning an Observable that will yield true if the authentication process has succeeded and false otherwise. The authenticated property is a getter-only property that returns true if the data source has obtained an authentication token. The clear method removes the token from the data source.

Listing 7-10 registers the new service with the model feature module. It also adds a providers entry for the RestDataSource class, which has been used only as a substitute for the StaticDataSource class in earlier chapters. Since the AuthService class has a RestDataSource constructor parameter, it needs its own entry in the module.

Listing 7-10. Configuring the Services in the model.module.ts File in the src/app/model Folder

```

import { NgModule } from "@angular/core";
import { ProductRepository } from "./product.repository";
import { StaticDataSource } from "./static.datasource";
import { Cart } from "./cart.model";
import { Order } from "./order.model";
import { OrderRepository } from "./order.repository";
import { RestDataSource } from "./rest.datasource";
import { HttpClientModule } from "@angular/common/http";
import { AuthService } from "./auth.service";

@NgModule({
  imports: [HttpClientModule],
  providers: [ProductRepository, StaticDataSource, Cart, Order, OrderRepository,
    { provide: StaticDataSource, useClass: RestDataSource },
    RestDataSource, AuthService]
})
export class ModelModule { }

```

Enabling Authentication

The next step is to wire up the component that obtains the credentials from the user so that it will perform authentication through the new service, as shown in Listing 7-11.

Listing 7-11. Enabling Authentication in the auth.component.ts File in the src/app/admin Folder

```

import { Component } from "@angular/core";
import { NgForm } from "@angular/forms";
import { Router } from "@angular/router";
import { AuthService } from "../model/auth.service";

@Component({
  templateUrl: "auth.component.html"
})

```

```

export class AuthComponent {
    username?: string;
    password?: string;
    errorMessage?: string;

    constructor(private router: Router,
        private auth: AuthService) { }

    authenticate(form: NgForm) {
        if (form.valid) {
            this.auth.authenticate(this.username ?? "", this.password ?? "")
                .subscribe(response => {
                    if (response) {
                        this.router.navigateByUrl("/admin/main");
                    }
                    this.errorMessage = "Authentication Failed";
                })
        } else {
            this.errorMessage = "Form Data Invalid";
        }
    }
}

```

To prevent the application from navigating directly to the administration features, which will lead to HTTP requests being sent without a token, I added a file called `auth.guard.ts` in the `src/app/admin` folder and defined the route guard shown in Listing 7-12.

Listing 7-12. The Contents of the `auth.guard.ts` File in the `src/app/admin` Folder

```

import { Injectable } from "@angular/core";
import { ActivatedRouteSnapshot, RouterStateSnapshot,
    Router } from "@angular/router";
import { AuthService } from "../model/auth.service";

@Injectable()
export class AuthGuard {

    constructor(private router: Router,
        private auth: AuthService) { }

    canActivate(route: ActivatedRouteSnapshot,
        state: RouterStateSnapshot): boolean {

        if (!this.auth.authenticated) {
            this.router.navigateByUrl("/admin/auth");
            return false;
        }
        return true;
    }
}

```

Listing 7-13 applies the route guard to one of the routes defined by the administration feature module.

Listing 7-13. Guarding a Route in the admin.module.ts File in the src/app/admin Folder

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';
import { RouterModule } from '@angular/router';
import { AuthComponent } from './auth.component';
import { AdminComponent } from './admin.component';
import { AuthGuard } from './auth.guard';

let routing = RouterModule.forChild([
  { path: "auth", component: AuthComponent },
  { path: "main", component: AdminComponent },
  { path: "main", component: AdminComponent, canActivate: [AuthGuard] },
  { path: "**", redirectTo: "auth" }
]);

@NgModule({
  imports: [CommonModule, FormsModule, routing],
  declarations: [AuthComponent, AdminComponent],
  providers: [AuthGuard]
})
export class AdminModule { }
```

To test the authentication system, click the Admin button, enter some credentials, and click the Log In button. If the credentials are the ones from Table 7-1, then you will see the placeholder for the administration features. If you enter other credentials, you will see an error message. Figure 7-3 illustrates both outcomes.

Tip The token isn't stored persistently, so if you can, reload the application in the browser to start again and try a different set of credentials.

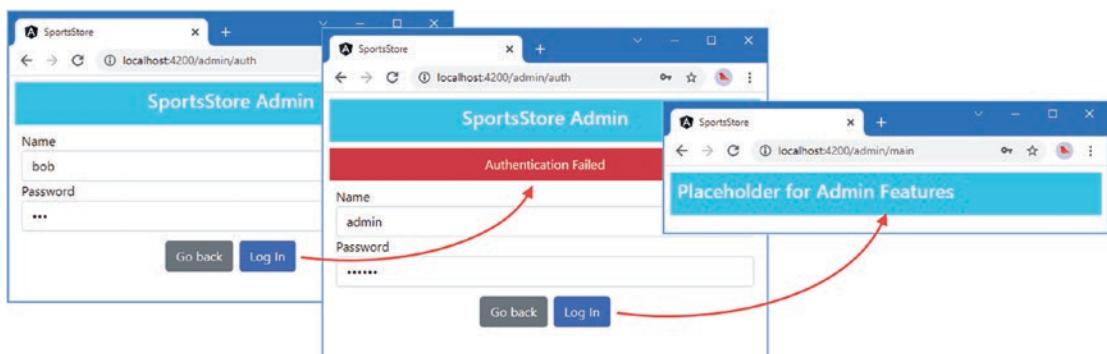


Figure 7-3. Testing the authentication feature

Extending the Data Source and Repositories

With the authentication system in place, the next step is to extend the data source so that it can send authenticated requests and to expose those features through the order and product repository classes. Listing 7-14 adds methods to the data source that include the authentication token.

Listing 7-14. Adding New Operations in the rest.datasource.ts File in the src/app/model Folder

```
import { Injectable } from "@angular/core";
import { HttpClient } from "@angular/common/http";
import { map, Observable } from "rxjs";
import { Product } from "./product.model";
import { Order } from "./order.model";
import { HttpHeaders } from '@angular/common/http';

const PROTOCOL = "http";
const PORT = 3500;

@Injectable()
export class RestDataSource {
    baseUrl: string;
    auth_token?: string;

    constructor(private http: HttpClient) {
        this.baseUrl = `${PROTOCOL}://${location.hostname}:${PORT}/`;
    }

    getProducts(): Observable<Product[]> {
        return this.http.get<Product[]>(this.baseUrl + "products");
    }

    saveOrder(order: Order): Observable<Order> {
        return this.http.post<Order>(this.baseUrl + "orders", order);
    }

    authenticate(user: string, pass: string): Observable<boolean> {
        return this.http.post<any>(this.baseUrl + "login", {
            name: user, password: pass
        }).pipe(map(response => {
            this.auth_token = response.success ? response.token : null;
            return response.success;
        }));
    }

    saveProduct(product: Product): Observable<Product> {
        return this.http.post<Product>(this.baseUrl + "products",
            product, this.getOptions());
    }
}
```

```

updateProduct(product: Product): Observable<Product> {
    return this.http.put<Product>(`${this.baseUrl}products/${product.id}`,
        product, this.getOptions());
}

deleteProduct(id: number): Observable<Product> {
    return this.http.delete<Product>(`${this.baseUrl}products/${id}`,
        this.getOptions());
}

getOrders(): Observable<Order[]> {
    return this.http.get<Order[]>(this.baseUrl + "orders", this.getOptions());
}

deleteOrder(id: number): Observable<Order> {
    return this.http.delete<Order>(`${this.baseUrl}orders/${id}`,
        this.getOptions());
}

updateOrder(order: Order): Observable<Order> {
    return this.http.put<Order>(`${this.baseUrl}orders/${order.id}`,
        order, this.getOptions());
}

private getOptions() {
    return {
        headers: new HttpHeaders({
            "Authorization": `Bearer ${this.auth_token}`
        })
    }
}
}

```

Listing 7-15 adds new methods to the product repository class that allow products to be created, updated, or deleted. The `saveProduct` method is responsible for creating and updating products, which is an approach that works well when using a single object managed by a component, which you will see demonstrated later in this chapter. The listing also changes the type of the constructor argument to `RestDataSource`.

Listing 7-15. Adding New Operations in the `product.repository.ts` File in the `src/app/model` Folder

```

import { Injectable } from "@angular/core";
import { Product } from "./product.model";
//import { StaticDataSource } from "./static.datasource";
import { RestDataSource } from "./rest.datasource";

@Injectable()
export class ProductRepository {
    private products: Product[] = [];
    private categories: string[] = [];
}

```

```

constructor(private dataSource: RestDataSource) {
    dataSource.getProducts().subscribe(data => {
        this.products = data;
        this.categories = data.map(p => p.category ?? "(None)")
            .filter((c, index, array) => array.indexOf(c) == index).sort();
    });
}

getProducts(category?: string): Product[] {
    return this.products
        .filter(p => category == undefined || category == p.category);
}

getProduct(id: number): Product | undefined {
    return this.products.find(p => p.id == id);
}

getCategories(): string[] {
    return this.categories;
}

saveProduct(product: Product) {
    if (product.id == null || product.id == 0) {
        this.dataSource.saveProduct(product)
            .subscribe(p => this.products.push(p));
    } else {
        this.dataSource.updateProduct(product)
            .subscribe(p => {
                this.products.splice(this.products.
                    findIndex(p => p.id == product.id), 1, product);
            });
    }
}

deleteProduct(id: number) {
    this.dataSource.deleteProduct(id).subscribe(p => {
        this.products.splice(this.products.
            findIndex(p => p.id == id), 1);
    })
}
}

```

Listing 7-16 makes the corresponding changes to the order repository, adding methods that allow orders to be modified and deleted.

Listing 7-16. Adding New Operations in the order.repository.ts File in the src/app/model Folder

```

import { Injectable } from "@angular/core";
import { Observable } from "rxjs";
import { Order } from "./order.model";
//import { StaticDataSource } from "./static.datasource";
import { RestDataSource } from "./rest.datasource";

```

```

@Injectable()
export class OrderRepository {
    private orders: Order[] = [];
    private loaded: boolean = false;

    constructor(private dataSource: RestDataSource) { }

    loadOrders() {
        this.loaded = true;
        this.dataSource.getOrders()
            .subscribe(orders => this.orders = orders);
    }

    getOrders(): Order[] {
        if (!this.loaded) {
            this.loadOrders();
        }
        return this.orders;
    }

    saveOrder(order: Order): Observable<Order> {
        this.loaded = false;
        return this.dataSource.saveOrder(order);
    }

    updateOrder(order: Order) {
        this.dataSource.updateOrder(order).subscribe(order => {
            this.orders.splice(this.orders.
                findIndex(o => o.id == order.id), 1, order);
        });
    }

    deleteOrder(id: number) {
        this.dataSource.deleteOrder(id).subscribe(order => {
            this.orders.splice(this.orders.findIndex(o => id == o.id), 1);
        });
    }
}

```

The order repository defines a `loadOrders` method that gets the orders from the repository and that ensures the request isn't sent to the RESTful web service until authentication has been performed.

Installing the Component Library

All the features presented to the user so far have been written using the Angular API and styled using the features provided by the Bootstrap CSS package. An alternative approach is to use a component library that contains commonly required features, such as tables and layouts, which lets you focus on the features that are unique to your project. The advantage of using a component library is that you can get a project up and running quickly, but the drawbacks are that you must fit your data and code into the model expected by the component library and that it can be difficult to perform customizations.

In this chapter, I am going to use the Angular Material component library. There are other good packages available for Angular, but Angular Material is the most popular package and has features that suit most projects. To add Angular Material to the project, stop the `ng serve` command and run the command shown in Listing 7-17 in the SportsStore folder.

Listing 7-17. Installing the Component Library Package

```
ng add @angular/material@13.0.2
```

The installation process asks several questions. The first question is just a request to confirm that you want to install the package:

```
Using package manager: npm
Package information loaded.
The package @angular/material@13.0.2 will be installed and executed.
Would you like to proceed? (Y/n)
```

The rest of the questions are specific to Angular Material and allow the theme to be selected and configure typography and animation options:

```
? Choose a prebuilt theme name, or "custom" for a custom theme: (Use arrow keys)
> Indigo/Pink      [ Preview: https://material.angular.io?theme=indigo-pink ]
Deep Purple/Amber  [ Preview: https://material.angular.io?theme=deeppurple-amber ]
Pink/Blue Grey    [ Preview: https://material.angular.io?theme=pink-bluegrey ]
Purple/Green       [ Preview: https://material.angular.io?theme=purple-green ]
Custom
? Set up global Angular Material typography styles? (y/N)
? Set up browser animations for Angular Material? (Y/n)
```

For the SportsStore project, select the default options.

Each feature provided by Angular Material is defined in its own module, and the simplest way to deal with this is to define a separate module that is used just to select the Angular Material features that are required by a project. Add a file named `material.module.ts` to the `src/app/admin` folder with the content shown in Listing 7-18.

Listing 7-18. The Contents of the `material.module.ts` File in the `src/app/admin` Folder

```
import { NgModule } from "@angular/core";
const features: any[] = [];
@NgModule({
  imports: [features],
  exports: [features]
})
export class MaterialFeatures {}
```

No Angular Material features are selected at present, but I'll add to this file as I work through the administration features. In Listing 7-19, I have incorporated the module into the application.

Listing 7-19. Using Material Features in the admin.module.ts File in the src/app/admin Folder

```

import { NgModule } from "@angular/core";
import { CommonModule } from "@angular/common";
import { FormsModule } from "@angular/forms";
import { RouterModule } from "@angular/router";
import { AuthComponent } from "./auth.component";
import { AdminComponent } from "./admin.component";
import { AuthGuard } from "./auth.guard";
import { MaterialFeatures } from "./material.module";

let routing = RouterModule.forChild([
  { path: "auth", component: AuthComponent },
  { path: "main", component: AdminComponent },
  { path: "main", component: AdminComponent, canActivate: [AuthGuard] },
  { path: "**", redirectTo: "auth" }
]);
@NgModule({
  imports: [CommonModule, FormsModule, routing, MaterialFeatures],
  declarations: [AuthComponent, AdminComponent],
  providers: [AuthGuard]
})
export class AdminModule { }

```

Save the changes and run the `ng serve` command in the SportsStore folder to start the Angular development tools again.

Creating the Administration Feature Structure

Now that the authentication system is in place and the repositories provide the full range of operations, I can create the structure that will display the administration features, which I create by building on the existing URL routing configuration. Table 7-2 lists the URLs that I am going to support and the functionality that each will present to the user.

Table 7-2. The URLs for Administration Features

Name	Description
/admin/main/products	Navigating to this URL will display all the products in a table, along with buttons that allow an existing product to be edited or deleted and a new product to be created.
/admin/main/products/create	Navigating to this URL will present the user with an empty editor for creating a new product.
/admin/main/products/edit/1	Navigating to this URL will present the user with a populated editor for editing an existing product.
/admin/main/orders	Navigating to this URL will present the user with all the orders in a table, along with buttons to mark an order shipped and to cancel an order by deleting it.

Creating the Placeholder Components

I find the easiest way to add features to an Angular project is to define components that have placeholder content and build the structure of the application around them. Once the structure is in place, then I return to the components and implement the features in detail. For the administration features, I started by adding a file called `productTable.component.ts` to the `src/app/admin` folder and defined the component shown in Listing 7-20. This component will be responsible for showing a list of products, along with buttons required to edit and delete them or to create a new product.

Listing 7-20. The Contents of the `productTable.component.ts` File in the `src/app/admin` Folder

```
import { Component } from "@angular/core";

@Component({
  template: `
    <h3 style="padding-top: 10px">
      Product Table Placeholder
    </h3>
  `
})
export class ProductTableComponent {}
```

I added a file called `productEditor.component.ts` in the `src/app/admin` folder and used it to define the component shown in Listing 7-21, which will be used to allow the user to enter the details required to create or edit a component.

Listing 7-21. The Contents of the `productEditor.component.ts` File in the `src/app/admin` Folder

```
import { Component } from "@angular/core";

@Component({
  template: `<h3 style="padding-top: 10px">
    Product Editor Placeholder
  </h3>`
})
export class ProductEditorComponent {}
```

To create the component that will be responsible for managing customer orders, I added a file called `orderTable.component.ts` to the `src/app/admin` folder and added the code shown in Listing 7-22.

Listing 7-22. The Contents of the `orderTable.component.ts` File in the `src/app/admin` Folder

```
import { Component } from "@angular/core";

@Component({
  template: `<h3 style="padding-top: 10px">
    Order Table Placeholder
  </h3>`
})
export class OrderTableComponent {}
```

Preparing the Common Content and the Feature Module

The components created in the previous section will be responsible for specific features. To bring those features together and allow the user to navigate between them, I need to modify the template of the placeholder component that I have been using to demonstrate the result of a successful authentication attempt. I replaced the placeholder content with the elements shown in Listing 7-23.

Listing 7-23. Replacing the Content in the admin.component.html File in the src/app/admin Folder

```

<mat-toolbar color="primary">
  <button mat-icon-button *ngIf=" sidenav.mode === 'over'" 
    (click)="sidenav.toggle()">
    <mat-icon *ngIf="!sidenav.opened">menu</mat-icon>
    <mat-icon *ngIf="sidenav.opened">close</mat-icon>
  </button>
  <span></span>
  SportsStore Administration
  <span></span>
</mat-toolbar>

<mat-sidenav-container>
  <mat-sidenav #sidenav="matSidenav" class="mat-elevation-z8">

    <button mat-button class="menu-button"
      routerLink="/admin/main/products"
      routerLinkActive="mat-accent"
      (click)="sidenav.close()">
      <mat-icon>shopping_cart</mat-icon>
      <span>Products</span>
    </button>

    <button mat-button class="menu-button"
      routerLink="/admin/main/orders"
      routerLinkActive="mat-accent"
      (click)="sidenav.close()">
      <mat-icon>local_shipping</mat-icon>
      <span>Orders</span>
    </button>

    <mat-divider></mat-divider>

    <button mat-button class="menu-button logout" (click)="logout()">
      <mat-icon>logout</mat-icon>
      <span>Logout</span>
    </button>

  </mat-sidenav>
  <mat-sidenav-content>
    <div class="content">
      <router-outlet></router-outlet>
    </div>
  </mat-sidenav-content>
</mat-sidenav-container>
```

When you first start working with a component library, it can take a while to make sense of how the components are applied. This template relies on the Angular Material toolbar, applied using the `mat-toolbar` element, and the `sidenav` component, which is applied through the `mat-sidenav-container`, `mat-sidenav`, and `mat-sidenav-content` elements. A `sidenav` is a collapsible panel that contains navigation content that will allow the user to select different administration features.

This template also contains a `router-outlet` element that will be used to display the components from the previous section. The `sidenav` panel contains buttons to which the `mat-button` directive has been applied, which formats the buttons to match the rest of the Angular Material theme. These buttons are configured with `routerLink` attributes that target the `router-outlet` element and are styled with the `routerLinkAttribute` attribute to indicate which feature has been selected.

Listing 7-24 adds dependencies on the Angular Material features used in this template.

Listing 7-24. Adding Features in the material.module.ts File in the src/app/admin Folder

```
import { NgModule } from "@angular/core";
import { MatToolbarModule } from "@angular/material/toolbar";
import { MatSidenavModule } from "@angular/material/sidenav";
import { MatIconModule } from "@angular/material/icon";
import { MatDividerModule } from '@angular/material/divider';
import { MatButtonModule } from "@angular/material/button";

const features: any[] = [MatToolbarModule, MatSidenavModule, MatIconModule,
    MatDividerModule, MatButtonModule];

@NgModule({
    imports: [features],
    exports: [features]
})
export class MaterialFeatures {}
```

One drawback of the Angular Material package is that it requires CSS styles to be applied to fine-tune the component layout. Listing 7-25 defines the styles required to lay out the components used in Listing 7-23.

Listing 7-25. Defining Styles in the styles.css File in the src Folder

```
html, body { height: 100%}
body { margin: 0; font-family: Roboto, "Helvetica Neue", sans-serif; }

mat-toolbar span { flex: 1 1 auto; }

.menu-button { width: 100%; font-size: 1rem; }
.menu-button .mat-icon { margin-right: 10px; }
.menu-button span { flex: 1 1 auto; }

mat-sidenav { margin: 16px; width: 175px; border-right: none;
    border-radius: 4px; padding: 4px;
}
mat-sidenav .mat-divider { margin-top: 20px; margin-bottom: 5px; }
mat-sidenav-container { height: calc(100vh - 60px); }
mat-sidenav .mat-button-wrapper {
```

```

display: flex; width: 100%; justify-content: baseline; align-content: center;
}
mat-sidenav .mat-button-wrapper mat-icon { margin-top: 5px; }
mat-sidenav .mat-button-wrapper span { text-align: start; }

```

These styles can be awkward to determine, and I find the most useful approach is to use the browser's F12 developer tools to work out how to select the elements I am interested in and determine how they are styled.

The sidenav panel defined in Listing 7-23 contains a Logout button that has an event binding that targets a method called logout. Listing 7-26 adds this method to the component, which uses the authentication service to remove the bearer token and navigates the application to the default URL.

Listing 7-26. Implementing the Logout Method in the admin.component.ts File in the src/app/admin Folder

```

import { Component } from "@angular/core";
import { Router } from "@angular/router";
import { AuthService } from "../model/auth.service";

@Component({
  templateUrl: "admin.component.html"
})
export class AdminComponent {

  constructor(private auth: AuthService,
    private router: Router) { }

  logout() {
    this.auth.clear();
    this.router.navigateByUrl("/");
  }
}

```

Listing 7-27 enables the placeholder components that will be used for each administration feature and extends the URL routing configuration to implement the URLs from Table 7-2.

Listing 7-27. Configuring the Feature Module in the admin.module.ts File in the src/app/admin Folder

```

import { NgModule } from "@angular/core";
import { CommonModule } from "@angular/common";
import { FormsModule } from "@angular/forms";
import { RouterModule } from "@angular/router";
import { AuthComponent } from "./auth.component";
import { AdminComponent } from "./admin.component";
import { AuthGuard } from "./auth.guard";
import { MaterialFeatures } from "./material.module";
import { ProductTableComponent } from "./productTable.component";
import { ProductEditorComponent } from "./productEditor.component";
import { OrderTableComponent } from "./orderTable.component";

let routing = RouterModule.forChild([
  { path: "auth", component: AuthComponent },
  // { path: "main", component: AdminComponent },

```

```
// { path: "main", component: AdminComponent, canActivate: [AuthGuard] },
{
  path: "main", component: AdminComponent, canActivate: [AuthGuard],
  children: [
    { path: "products/:mode/:id", component: ProductEditorComponent },
    { path: "products/:mode", component: ProductEditorComponent },
    { path: "products", component: ProductTableComponent },
    { path: "orders", component: OrderTableComponent },
    { path: "**", redirectTo: "products" }
  ]
},
{ path: "**", redirectTo: "auth" }
]);

@NgModule({
  imports: [CommonModule, FormsModule, routing, MaterialFeatures],
  declarations: [AuthComponent, AdminComponent, ProductTableComponent,
    ProductEditorComponent, OrderTableComponent],
  providers: [AuthGuard]
})
export class AdminModule { }
```

Individual routes can be extended using the `children` property, which is used to define routes that will target a nested router-outlet element, which I describe in Chapter 24. As you will see, components can get details of the active route from Angular so they can adapt their behavior. Routes can include route parameters, such as `:mode` or `:id`, that match any URL segment and that can be used to provide information to components that can be used to change their behavior.

When all the changes have been saved, click the Admin button and authenticate as `admin` with the password `secret`. You will see the new layout, as shown in Figure 7-4. Click the button on the left side of the toolbar to display the navigation panel, and click the Orders button to change the selected component, or the Logout button to exit the administration area.

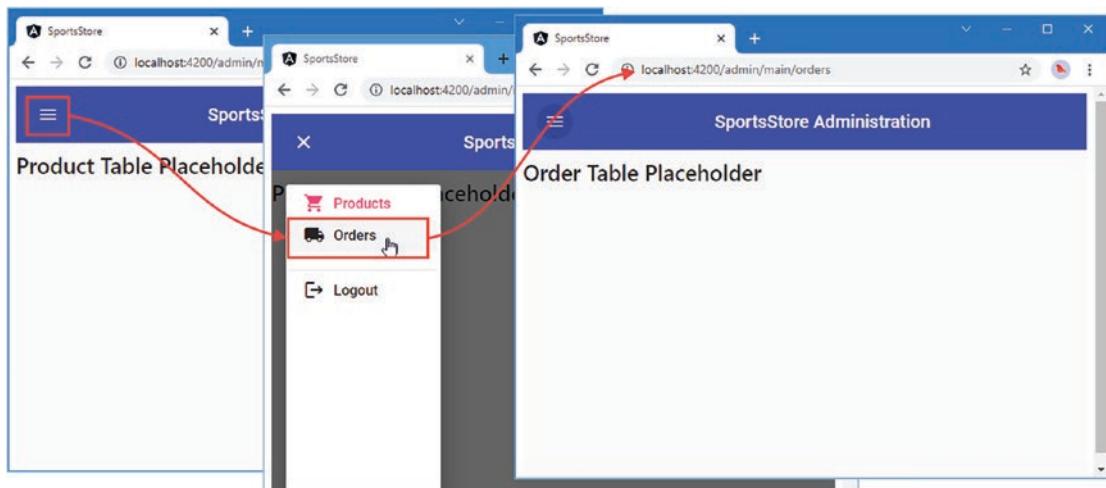


Figure 7-4. The administration layout structure

Implementing the Product Table Feature

The initial administration feature presented to the user will be a table of products, with the ability to create a new product and delete or edit an existing one. Listing 7-28 adds the Angular Material table component to the application.

Listing 7-28. Adding Features in the material.module.ts File in the src/app/admin Folder

```
import { NgModule } from '@angular/core';
import { MatToolbarModule } from '@angular/material/toolbar';
import { MatSidenavModule } from '@angular/material/sidenav';
import { MatIconModule } from '@angular/material/icon';
import { MatDividerModule } from '@angular/material/divider';
import { MatButtonModule } from '@angular/material/button';
import { MatTableModule } from '@angular/material/table';

const features: any[] = [MatToolbarModule, MatSidenavModule, MatIconModule,
  MatDividerModule, MatButtonModule, MatTableModule];

@NgModule({
  imports: [features],
  exports: [features]
})
export class MaterialFeatures {}
```

To provide the template that defines the table, I added a file called `productTable.component.html` in the `src/app/admin` folder and added the markup shown in Listing 7-29.

Listing 7-29. The Contents of the productTable.component.html File in the src/app/admin Folder

```
<table mat-table [dataSource]="dataSource">

<mat-text-column name="id"></mat-text-column>
<mat-text-column name="name"></mat-text-column>
<mat-text-column name="category"></mat-text-column>

<ng-container matColumnDef="price">
  <th mat-header-cell *matHeaderCellDef>Price</th>
  <td mat-cell *matCellDef="let item"> {{item.price | currency:"USD"}} </td>
</ng-container>

<ng-container matColumnDef="buttons">
  <th mat-header-cell *matHeaderCellDef></th>
  <td mat-cell *matCellDef="let p">
    <button mat-flat-button color="accent"
      (click)="deleteProduct(p.id)">
      Delete
    </button>
    <button mat-flat-button color="warn"
      [routerLink]="/admin/main/products/edit", p.id]>
      Edit
    </button>
  </td>
</ng-container>
```

```

<tr mat-header-row *matHeaderRowDef="colsAndRows"></tr>
<tr mat-row *matRowDef="let row; columns: colsAndRows"></tr>
</table>

<button mat-flat-button color="primary" routerLink="/admin/main/products/create">
    Create New Product
</button>
```

The table relies on the features provided by the Angular Material table component, which has an unusual approach to defining the table contents, but one that provides a good foundation for extra features, as I demonstrate shortly. The table defines columns that display the details of products, and each row contains a Delete button that invokes a component method named `delete` method, and an Edit button that navigates to a URL that targets the editor component. The editor component is also the target of the Create New Product button, although a different URL is used.

Once again, custom CSS styles are required to fine-tune the layout of the table, as shown in Listing 7-30.

Listing 7-30. Defining Styles in the styles.css File in the src Folder

```

html, body { height: 100% }
body { margin: 0; font-family: Roboto, "Helvetica Neue", sans-serif; }

mat-toolbar span { flex: 1 1 auto; }

.menu-button { width: 100%; font-size: 1rem; }
.menu-button .mat-icon { margin-right: 10px; }
.menu-button span { flex: 1 1 auto; }

mat-sidenav { margin: 16px; width: 175px; border-right: none;
    border-radius: 4px; padding: 4px;
}
mat-sidenav .mat-divider { margin-top: 20px; margin-bottom: 5px; }
mat-sidenav-container { height: calc(100vh - 60px); }
mat-sidenav .mat-button-wrapper {
    display: flex; width: 100%; justify-content: baseline; align-content: center;
}
mat-sidenav .mat-button-wrapper mat-icon { margin-top: 5px; }
mat-sidenav .mat-button-wrapper span { text-align: start; }

table[mat-table] { width: 100%; table-layout: auto; }
table[mat-table] button { margin-left: 5px; }
table[mat-table] th.mat-header-cell { font-size: large; font-weight: bold; }
table[mat-table] .mat-column-name { width: 25%; }
table[mat-table] .mat-column-buttons { width: 30%; }
table[mat-table] + button[mat-flat-button] { margin-top: 10px; }
```

As I explained when I installed the Angular Material package, using a component library means that you have to adapt your application or data to the expectations of the package. Taking full advantage of the Angular Material table requires the use of a data source class, which can require work when the data displayed in the table is obtained via an HTTP request. In Part 3, I demonstrate how to avoid this issue using observables, including with the Angular Material table in Chapter 28. For this chapter, I am going

to demonstrate a different approach, which is to use the features that Angular provides for detecting and processing updates. Listing 7-31 removes the placeholder content from the product table component and adds the logic required to implement this feature.

Listing 7-31. Adding Features in the productTable.component.ts File in the src/app/admin Folder

```
import { Component, IterableDiffer, IterableDiffers } from "@angular/core";
import { MatTableDataSource } from "@angular/material/table";
import { Product } from "../model/product.model";
import { ProductRepository } from "../model/product.repository";

@Component({
  templateUrl: "productTable.component.html"
})
export class ProductTableComponent {
  colsAndRows: string[] = ['id', 'name', 'category', 'price', 'buttons'];
  dataSource = new MatTableDataSource<Product>(this.repository.getProducts());
  differ: IterableDiffer<Product>;

  constructor(private repository: ProductRepository, differs: IterableDiffers) {
    this.differ = differs.find(this.repository.getProducts()).create();
  }

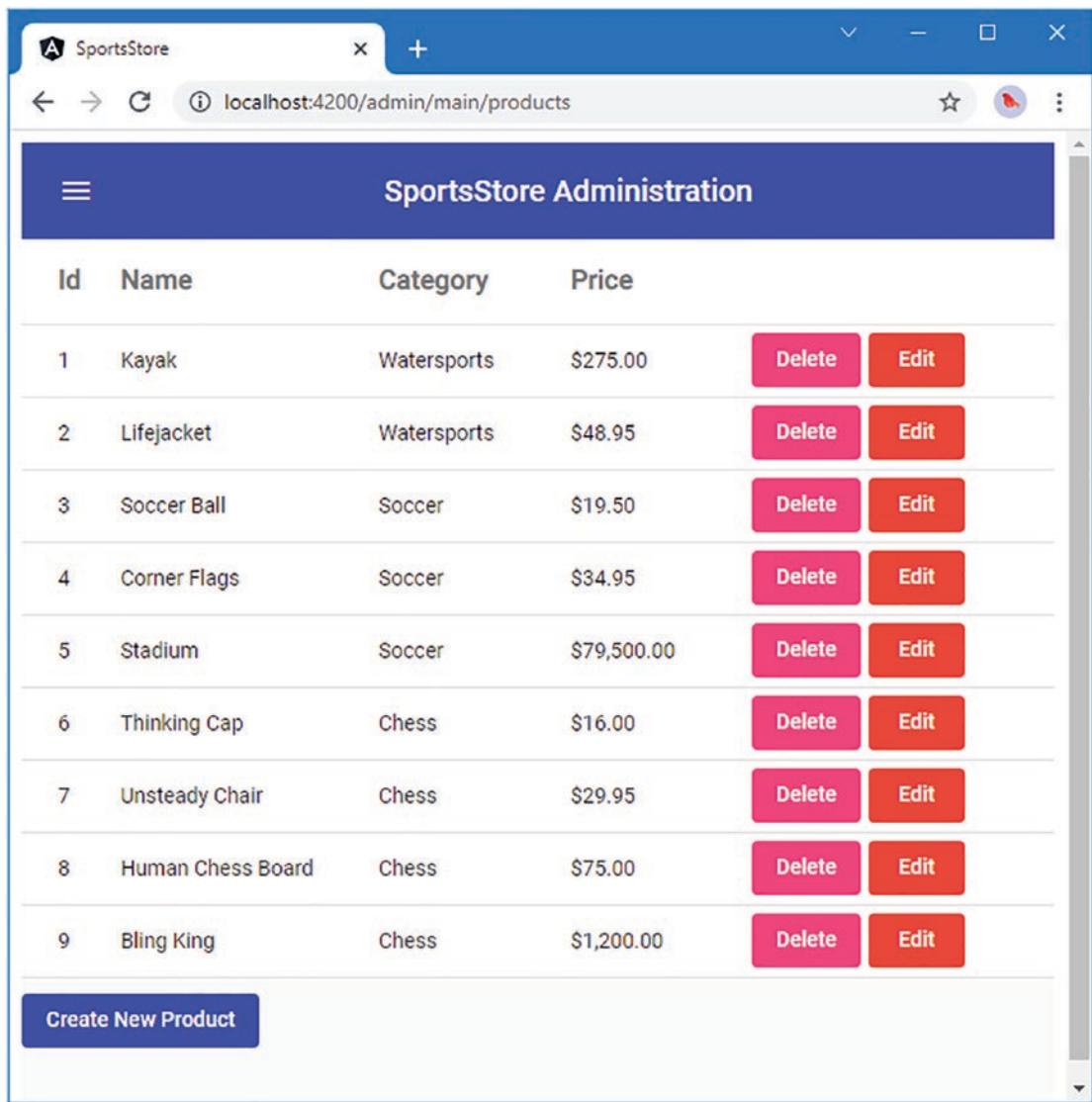
  ngDoCheck() {
    let changes = this.differ?.diff(this.repository.getProducts());
    if (changes != null) {
      this.dataSource.data = this.repository.getProducts();
    }
  }

  deleteProduct(id: number) {
    this.repository.deleteProduct(id);
  }
}
```

The colsAndRows property is used to specify the columns that are displayed in the table. I have selected all of the columns that were defined, but this feature can be used to programmatically alter the structure of the table.

The MatTableDataSource<Product> class connects the data in the application with the table. The data source object is created with the data in the repository, but this isn't helpful if the component is displayed before the application receives the data from the server. Angular has an efficient change-detection system, which it uses to ensure that updates are processed with the minimum of work, and the ngDoCheck method allows me to hook into that system and check to see if the data in the repository has been changed, using features that are described in context in Chapter 13. If there is a change in the data, then I refresh the data source, which has the effect of updating the table.

Save the changes and log into the administration features, and you will see the table shown in Figure 7-5.



The screenshot shows a web browser window titled "SportsStore" with the URL "localhost:4200/admin/main/products". The page has a dark blue header with the text "SportsStore Administration". Below the header is a table with the following data:

Id	Name	Category	Price	Delete	Edit
1	Kayak	Watersports	\$275.00	Delete	Edit
2	Lifejacket	Watersports	\$48.95	Delete	Edit
3	Soccer Ball	Soccer	\$19.50	Delete	Edit
4	Corner Flags	Soccer	\$34.95	Delete	Edit
5	Stadium	Soccer	\$79,500.00	Delete	Edit
6	Thinking Cap	Chess	\$16.00	Delete	Edit
7	Unsteady Chair	Chess	\$29.95	Delete	Edit
8	Human Chess Board	Chess	\$75.00	Delete	Edit
9	Bling King	Chess	\$1,200.00	Delete	Edit

Create New Product

Figure 7-5. Displaying the product table

Using the Table Component Features

Getting a library component working can require effort, but once the initial work is done, it becomes relatively simple to take advantage of the additional features that are provided. In the case of the Angular Material table, these features include filtering, sorting, and paginate data. I demonstrate the filtering support when implementing the orders feature, but in this section, I am going to use the pagination feature. The first step is to add the pagination feature to the application, as shown in Listing 7-32.

Listing 7-32. Adding a Feature in the material.module.ts File in the src/app/admin Folder

```

import { NgModule } from "@angular/core";
import { MatToolbarModule } from "@angular/material/toolbar";
import { MatSidenavModule } from "@angular/material/sidenav";
import { MatIconModule } from '@angular/material/icon';
import { MatDividerModule } from '@angular/material/divider';
import { MatButtonModule } from "@angular/material/button";
import { MatTableModule } from "@angular/material/table";
import { MatPaginatorModule } from "@angular/material/paginator";

const features: any[] = [MatToolbarModule, MatSidenavModule, MatIconModule,
  MatDividerModule, MatButtonModule, MatTableModule, MatPaginatorModule];

@NgModule({
  imports: [features],
  exports: [features]
})
export class MaterialFeatures {}

```

The next step is to add a paginator to the component that displays the table, as shown in Listing 7-33.

Listing 7-33. Adding Pagination in the productTable.component.html File in the src/app/admin Folder

```

<table mat-table [dataSource]="dataSource">

  <mat-text-column name="id"></mat-text-column>
  <mat-text-column name="name"></mat-text-column>
  <mat-text-column name="category"></mat-text-column>

  <ng-container matColumnDef="price">
    <th mat-header-cell *matHeaderCellDef>Price</th>
    <td mat-cell *matCellDef="let item"> {{item.price | currency:"USD"}} </td>
  </ng-container>

  <ng-container matColumnDef="buttons">
    <th mat-header-cell *matHeaderCellDef></th>
    <td mat-cell *matCellDef="let p">
      <button mat-flat-button color="accent"
        (click)="deleteProduct(p.id)">
        Delete
      </button>
      <button mat-flat-button color="warn"
        [routerLink]="/admin/main/products/edit", p.id]>
        Edit
      </button>
    </td>
  </ng-container>

  <tr mat-header-row *matHeaderRowDef="colsAndRows"></tr>
  <tr mat-row *matRowDef="let row; columns: colsAndRows"></tr>
</table>

```

```
<div class="bottom-box">
  <button mat-flat-button color="primary" routerLink="/admin/main/products/create">
    Create New Product
  </button>
  <mat-paginator [pageSize]="5" [pageSizeOptions]=[3, 5, 10]">
  </mat-paginator>
</div>
```

The paginator must be associated with the data source that is used by the table, which is done in the component, as shown in Listing 7-34.

Listing 7-34. Connecting the Paginator in the productTable.component.ts File in the src/app/admin Folder

```
import { Component, IterableDiffer, IterableDifferences, ViewChild } from "@angular/core";
import { MatTableDataSource } from "@angular/material/table";
import { Product } from "../model/product.model";
import { ProductRepository } from "../model/product.repository";
import { MatPaginator } from "@angular/material/paginator";

@Component({
  templateUrl: "productTable.component.html"
})
export class ProductTableComponent {
  colsAndRows: string[] = ['id', 'name', 'category', 'price', 'buttons'];

  dataSource = new MatTableDataSource<Product>(this.repository.getProducts());
  differ: IterableDiffer<Product>;

  @ViewChild(MatPaginator)
  paginator? : MatPaginator

  constructor(private repository: ProductRepository, differs: IterableDifferences) {
    this.differ = differs.find(this.repository.getProducts()).create();
  }

  ngDoCheck() {
    let changes = this.differ?.diff(this.repository.getProducts());
    if (changes != null) {
      this.dataSource.data = this.repository.getProducts();
    }
  }

  ngAfterViewInit() {
    if (this.paginator) {
      this.dataSource.paginator = this.paginator;
    }
  }

  deleteProduct(id: number) {
    this.repository.deleteProduct(id);
  }
}
```

The `ViewChild` decorator is used to query the component's template content, as described in Chapter 15, and is used here to find the paginator component. The `ngAfterViewInit` method is called after Angular has finished processing the template, as described in Chapter 13, by which time the paginator component will have been created and can be associated with the data source.

And, of course, some additional CSS styles are required to manage the layout, as shown in Listing 7-35.

Listing 7-35. Defining Styles in the styles.css File in the src Folder

```
html, body { height: 100%}
body { margin: 0; font-family: Roboto, "Helvetica Neue", sans-serif; }

mat-toolbar span { flex: 1 1 auto; }

.menu-button { width: 100%; font-size: 1rem; }
.menu-button .mat-icon { margin-right: 10px; }
.menu-button span { flex: 1 1 auto; }

mat-sidenav { margin: 16px; width: 175px; border-right: none;
              border-radius: 4px; padding: 4px;
}
mat-sidenav .mat-divider { margin-top: 20px; margin-bottom: 5px; }
mat-sidenav-container { height: calc(100vh - 60px); }
mat-sidenav .mat-button-wrapper {
    display: flex; width: 100%; justify-content: baseline; align-content: center;
}
mat-sidenav .mat-button-wrapper mat-icon { margin-top: 5px; }
mat-sidenav .mat-button-wrapper span { text-align: start; }

table[mat-table] { width: 100%; table-layout: auto; }
table[mat-table] button { margin-left: 5px; }
table[mat-table] th.mat-header-cell { font-size: large; font-weight: bold; }
table[mat-table] .mat-column-name { width: 25%; }
table[mat-table] .mat-column-buttons { width: 30%; }
/* table[mat-table] + button[mat-flat-button] { margin-top: 10px; } */

.bottom-box { background-color: white; padding-bottom: 20px; }
.bottom-box > button[mat-flat-button] { margin-top: 10px; }
.bottom-box mat-paginator { float: right; font-size: 14px; }
```

Save the changes, and you will see that the initial work adapting the application to work in the model expected by Angular Material has paid off, and I am able to use the built-in support for pagination, as shown in Figure 7-6.

Id	Name	Category	Price		
1	Kayak	Watersports	\$275.00	<button>Delete</button>	<button>Edit</button>
2	Lifejacket	Watersports	\$48.95	<button>Delete</button>	<button>Edit</button>
3	Soccer Ball	Soccer	\$19.50	<button>Delete</button>	<button>Edit</button>
4	Corner Flags	Soccer	\$34.95	<button>Delete</button>	<button>Edit</button>
5	Stadium	Soccer	\$79,500.00	<button>Delete</button>	<button>Edit</button>

Create New Product Items per page: 5 1 - 5 of 9

Figure 7-6. Using the Angular Material table paginator

Implementing the Product Editor

Components can receive information about the current routing URL and adapt their behavior accordingly. The editor component needs to use this feature to differentiate between requests to create a new component and edit an existing one.

Listing 7-36 adds the functionality to the editor component required to create or edit products.

Listing 7-36. Adding Functionality in the productEditor.component.ts File in the src/app/admin Folder

```
import { Component } from "@angular/core";
import { Router, ActivatedRoute } from "@angular/router";
import { Product } from "../model/product.model";
import { ProductRepository } from "../model/product.repository";

@Component({
  templateUrl: "productEditor.component.html"
})
export class ProductEditorComponent {
  editing: boolean = false;
  product: Product = new Product();
```

```

constructor(private repository: ProductRepository,
            private router: Router,
            activeRoute: ActivatedRoute) {

    this.editing = activeRoute.snapshot.params["mode"] == "edit";
    if (this.editing) {
        Object.assign(this.product,
            repository.getProduct(activeRoute.snapshot.params["id"]));
    }
}

save() {
    this.repository.saveProduct(this.product);
    this.router.navigateByUrl("/admin/main/products");
}
}

```

Angular will provide an `ActivatedRoute` object as a constructor argument when it creates a new instance of the component class, and this object can be used to inspect the activated route. In this case, the component works out whether it should be editing or creating a product and, if editing, retrieves the current details from the repository. There is also a `save` method, which uses the repository to save changes that the user has made.

An HTML form will be used to allow the user to edit products. The Angular Material package provides support for form fields; Listing 7-37 adds those features to the SportsStore application.

Listing 7-37. Adding a Feature in the material.module.ts File in the src/app/admin Folder

```

import { NgModule } from "@angular/core";
import { MatToolbarModule } from "@angular/material/toolbar";
import { MatSidenavModule } from "@angular/material/sidenav";
import { MatIconModule } from '@angular/material/icon';
import { MatDividerModule } from '@angular/material/divider';
import { MatButtonModule } from "@angular/material/button";
import { MatTableModule } from "@angular/material/table";
import { MatPaginatorModule } from "@angular/material/paginator";
import { MatFormFieldModule } from '@angular/material/form-field';
import { MatInputModule } from '@angular/material/input';

const features: any[] = [MatToolbarModule, MatSidenavModule, MatIconModule,
MatDividerModule, MatButtonModule, MatTableModule, MatPaginatorModule,
MatFormFieldModule, MatInputModule];

@NgModule({
    imports: [features],
    exports: [features]
})
export class MaterialFeatures {}

```

To define the template with the form, add a file called `productEditor.component.html` in the `src/app/admin` folder and add the markup shown in Listing 7-38.

Listing 7-38. The Contents of the productEditor.component.html File in the src/app/admin Folder

```
<h3 class="heading">{{editing ? "Edit" : "Create"}} Product</h3>

<form (ngSubmit)="save()">

  <mat-form-field *ngIf="editing">
    <mat-label>ID</mat-label>
    <input matInput name="id" [(ngModel)]="product.id" disabled />
  </mat-form-field>

  <mat-form-field>
    <mat-label>Name</mat-label>
    <input matInput name="name" [(ngModel)]="product.name" />
  </mat-form-field>

  <mat-form-field>
    <mat-label>Category</mat-label>
    <input matInput name="category" [(ngModel)]="product.category" />
  </mat-form-field>

  <mat-form-field>
    <mat-label>Description</mat-label>
    <input name="description" matInput [(ngModel)]="product.description"/>
  </mat-form-field>

  <mat-form-field>
    <mat-label>Price</mat-label>
    <input matInput name="price" [(ngModel)]="product.price" />
  </mat-form-field>

  <button type="submit" mat-flat-button color="primary">
    {{editing ? "Save" : "Create"}}
  </button>
  <button type="reset" mat-stroked-button routerLink="/admin/main/products">
    Cancel
  </button>
</form>
```

The template contains a form with fields for the properties defined by the `Product` model class. The field for the `id` property is shown only when editing an existing product and is disabled because the value cannot be changed. Listing 7-39 defines the styles that are required to lay out the form.

Listing 7-39. Defining Styles in the styles.css File in the src Folder

```
html, body { height: 100% }
body { margin: 0; font-family: Roboto, "Helvetica Neue", sans-serif; }

mat-toolbar span { flex: 1 1 auto; }

.menu-button { width: 100%; font-size: 1rem; }
.menu-button .mat-icon { margin-right: 10px; }
.menu-button span { flex: 1 1 auto; }
```

```
mat-sidenav { margin: 16px; width: 175px; border-right: none;
    border-radius: 4px; padding: 4px;
}
mat-sidenav .mat-divider { margin-top: 20px; margin-bottom: 5px; }
mat-sidenav-container { height: calc(100vh - 60px); }
mat-sidenav .mat-button-wrapper {
    display: flex; width: 100%; justify-content: baseline; align-content: center;
}
mat-sidenav .mat-button-wrapper mat-icon { margin-top: 5px; }
mat-sidenav .mat-button-wrapper span { text-align: start; }

table[mat-table] { width: 100%; table-layout: auto; }
table[mat-table] button { margin-left: 5px; }
table[mat-table] th.mat-header-cell { font-size: large; font-weight: bold; }
table[mat-table] .mat-column-name { width: 25%; }
table[mat-table] .mat-column-buttons { width: 30%; }
/* table[mat-table] + button[mat-flat-button] { margin-top: 10px; } */

.bottom-box { background-color: white; padding-bottom: 20px; }
.bottom-box > button[mat-flat-button] { margin-top: 10px; }
.bottom-box mat-paginator { float: right; font-size: 14px; }

mat-form-field { width: 100%; }
mat-form-field:first-child { margin-top: 20px; }
form button[mat-flat-button] { margin-top: 10px; margin-right: 10px; }
h3.heading { margin-top: 20px; }
```

To see how the component works, authenticate to access the Admin features and click the Create New Product button that appears under the table of products. Fill out the form, click the Create button, and the new product will be sent to the RESTful web service where it will be assigned an ID property and displayed in the product table, as shown in Figure 7-7.

■ **Tip** Restart the `ng serve` command if you see an error after saving these changes.

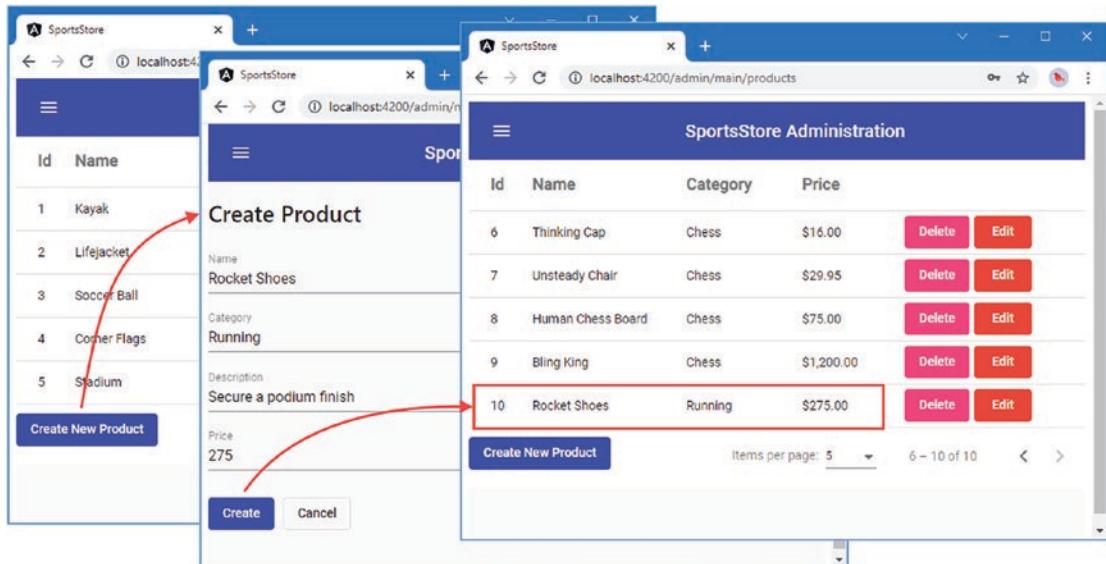


Figure 7-7. Creating a new product

The editing process works in a similar way. Click one of the Edit buttons to see the current details, edit them using the form fields, and click the Save button to save the changes, as shown in Figure 7-8.

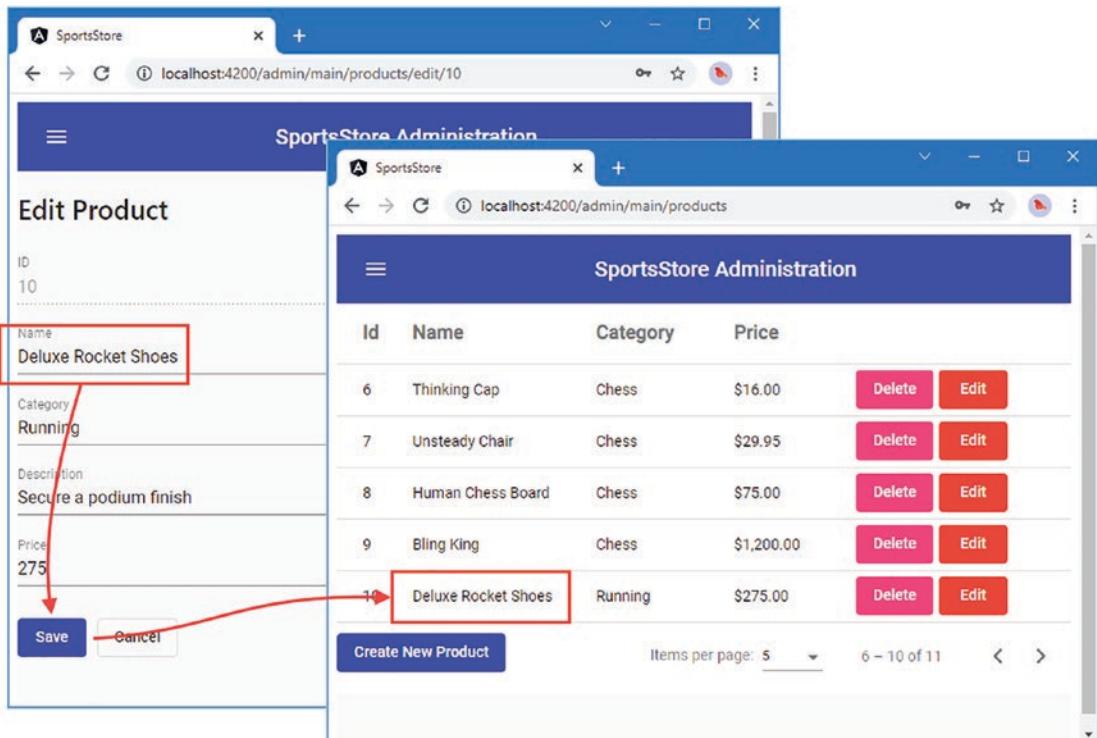


Figure 7-8. Editing an existing product

Implementing the Order Table Feature

The order management feature is nice and simple. It requires a table that lists the set of orders, along with buttons that will set the shipped property or delete an order entirely. The table will be displayed with a checkbox that will include shipped orders in the table. Listing 7-40 adds the Angular Material checkbox feature to the SportsStore project.

Listing 7-40. Adding a Feature in the material.module.ts File in the src/app/admin Folder

```
import { NgModule } from "@angular/core";
import { MatToolbarModule } from "@angular/material/toolbar";
import { MatSidenavModule } from "@angular/material/sidenav";
import { MatIconModule } from '@angular/material/icon';
import { MatDividerModule } from '@angular/material/divider';
import { MatButtonModule } from "@angular/material/button";
import { MatTableModule } from "@angular/material/table";
import { MatPaginatorModule } from "@angular/material/paginator";
import { MatFormFieldModule } from '@angular/material/form-field';
import { MatInputModule } from '@angular/material/input';
import { MatCheckboxModule } from '@angular/material/checkbox';

const features: any[] = [MatToolbarModule, MatSidenavModule, MatIconModule,
  MatDividerModule, MatButtonModule, MatTableModule, MatPaginatorModule,
  MatFormFieldModule, MatInputModule, MatCheckboxModule];

@NgModule({
  imports: [features],
  exports: [features]
})
export class MaterialFeatures {}
```

To create the template, I added a file called `orderTable.component.html` to the `src/app/admin` folder with the content shown in Listing 7-41.

Listing 7-41. The Contents of the `orderTable.component.html` File in the `src/app/admin` Folder

```
<mat-checkbox [(ngModel)]="includeShipped">Display Shipped Orders</mat-checkbox>

<table class="orders" mat-table [dataSource]="dataSource">

  <mat-text-column name="name"></mat-text-column>
  <mat-text-column name="zip"></mat-text-column>

  <ng-container matColumnDef="cart_p">
    <th mat-header-cell *matHeaderCellDef></th>
    <td mat-cell *matCellDef="let order">
      <table mat-table [dataSource]="order.cart.lines">
        <ng-container matColumnDef="p">
          <th mat-header-cell *matHeaderCellDef>Product</th>
          <td mat-cell *matCellDef="let line">{{ line.product.name }}</td>
        </ng-container>
      </table>
    </td>
  </ng-container>
```

```

        <tr mat-header-row *matHeaderRowDef="['p']"></tr>
        <tr mat-row *matRowDef="let row; columns: ['p']"></tr>
    </table>
</td>
</ng-container>

<ng-container matColumnDef="cart_q">
    <th mat-header-cell *matHeaderCellDef></th>
    <td mat-cell *matCellDef="let order">
        <table mat-table [dataSource]="order.cart.lines">
            <ng-container matColumnDef="q">
                <th mat-header-cell *matHeaderCellDef>Quantity</th>
                <td mat-cell *matCellDef="let line">{{ line.quantity }}</td>
            </ng-container>
            <tr mat-header-row *matHeaderRowDef="['q']"></tr>
            <tr mat-row *matRowDef="let row; columns: ['q']"></tr>
        </table>
    </td>
</ng-container>

<ng-container matColumnDef="buttons">
    <th mat-header-cell *matHeaderCellDef>Actions</th>
    <td mat-cell *matCellDef="let o">
        <button mat-flat-button color="primary" (click)="toggleShipped(o)">
            {{ o.shipped ? "Unship" : "Ship" }}
        </button>
        <button mat-flat-button color="warn" (click)="delete(o.id)">
            Delete
        </button>
    </td>
</ng-container>

<tr mat-header-row *matHeaderRowDef="colsAndRows"></tr>
<tr mat-row *matRowDef="let row; columns: colsAndRows"></tr>

<tr class="mat-row" *matNoDataRow>
    <td class="mat-cell no-data" colspan="4">No orders to display</td>
</tr>
</table>

```

This template contains tables within a table, which allows me to produce a variable number of rows for each order, reflecting the customer's product selections. There is also a checkbox that sets a property named `includeShipped`, which is defined in Listing 7-42, along with the rest of the features required to support the template.

Listing 7-42. Adding Features in the `orderTable.component.ts` File in the `src/app/admin` Folder

```

import { Component, IterableDiffer, IterableDiffers } from "@angular/core";
import { MatTableDataSource } from "@angular/material/table";
import { Order } from "../model/order.model";
import { OrderRepository } from "../model/order.repository";

```

```

@Component({
  templateUrl: "orderTable.component.html"
})
export class OrderTableComponent {
  colsAndRows: string[] = ['name', 'zip','cart_p','cart_q', 'buttons'];

  dataSource = new MatTableDataSource<Order>(this.repository.getOrders());
  differ: IterableDiffer<Order>;

  constructor(private repository: OrderRepository, differs: IterableDiffers) {
    this.differ = differs.find(this.repository.getOrders()).create();
    this.dataSource.filter = "true";
    this.dataSource.filterPredicate = (order, include) => {
      return !order.shipped || include.toString() == "true"
    };
  }

  get includeShipped(): boolean {
    return this.dataSource.filter == "true";
  }

  set includeShipped(include: boolean) {
    this.dataSource.filter = include.toString()
  }

  toggleShipped(order: Order) {
    order.shipped = !order.shipped;
    this.repository.updateOrder(order);
  }

  delete(id: number) {
    this.repository.deleteOrder(id);
  }

  ngDoCheck() {
    let changes = this.differ?.diff(this.repository.getOrders());
    if (changes != null) {
      this.dataSource.data = this.repository.getOrders();
    }
  }
}

```

The way that data is filtered in this example is a good example of adapting to the way the component library works. The support for filtering data provided by the Angular Material table is intended to search for strings in the table and to update the filtered data only when a new search string is specified. I have replaced the function used to filter rows and ensure that filtering is applied by binding changes from the checkbox so that the search string is altered each time.

The final step is to define yet more CSS to control the layout of the table and its contents, as shown in Listing 7-43.

Listing 7-43. Defining Styles in the styles.css File in the src Folder

```

html, body { height: 100%}
body { margin: 0; font-family: Roboto, "Helvetica Neue", sans-serif; }

mat-toolbar span { flex: 1 1 auto; }

.menu-button { width: 100%; font-size: 1rem; }
.menu-button .mat-icon { margin-right: 10px; }
.menu-button span { flex: 1 1 auto; }

mat-sidenav { margin: 16px; width: 175px; border-right: none;
    border-radius: 4px; padding: 4px;
}
mat-sidenav .mat-divider { margin-top: 20px; margin-bottom: 5px; }
mat-sidenav-container { height: calc(100vh - 60px); }
mat-sidenav .mat-button-wrapper {
    display: flex; width: 100%; justify-content: baseline; align-content: center;
}
mat-sidenav .mat-button-wrapper mat-icon { margin-top: 5px; }
mat-sidenav .mat-button-wrapper span { text-align: start; }

table[mat-table] { width: 100%; table-layout: auto; }
table[mat-table] button { margin-left: 5px; }
table[mat-table] th.mat-header-cell { font-size: large; font-weight: bold; }
table[mat-table] .mat-column-name { width: 25%; }
table[mat-table] .mat-column-buttons { width: 30%; }
/* table[mat-table] + button[mat-flat-button] { margin-top: 10px;} */

.bottom-box { background-color: white; padding-bottom: 20px; }
.bottom-box > button[mat-flat-button] { margin-top: 10px; }
.bottom-box mat-paginator { float: right; font-size: 14px; }

mat-form-field { width: 100%; }
mat-form-field:first-child { margin-top: 20px; }
form button[mat-flat-button] { margin-top: 10px; margin-right: 10px; }
h3.heading { margin-top: 20px; }

mat-checkbox { margin: 10px; font-size: large;}
td.no-data { font-size: large;}
table.orders tbody, table.orders thead { vertical-align: top }
table.orders td { padding-top: 10px;}
table.orders table th:first-of-type, table.orders table td:first-of-type {
    margin: 0; padding: 0;
}

```

Remember that the data presented by the RESTful web service is reset each time the process is started, which means you will have to use the shopping cart and check out to create orders. Once that's done, you can inspect and manage them using the Orders section of the administration tool, as shown in Figure 7-9.

The screenshot shows a web browser window titled "SportsStore" with the URL "localhost:4200/admin/main/orders". The page has a dark blue header with the text "SportsStore Administration". Below the header, there is a checkbox labeled "Display Shipped Orders" which is checked. The main content area displays two orders in a table format.

Name	Zip	Product	Quantity	Actions
Bob Smith	10036	Kayak	1	<button>Ship</button> <button>Delete</button>
		Stadium	1	
Alice Jones	SW1A 1AA	Bling King	2	<button>Ship</button> <button>Delete</button>
		Lifejacket	1	
		Thinking Cap	1	

Figure 7-9. Managing orders

Summary

In this chapter, I created a dynamically loaded Angular feature module that contains the administration tools required to manage the catalog of products and process orders. In the next chapter, I finish the SportsStore application and prepare it for deployment into production.

CHAPTER 8



SportsStore: Progressive Features and Deployment

In this chapter, I prepare the SportsStore application for deployment by adding progressive features that will allow it to work while offline and show you how to prepare and deploy the application into a Docker container, which can be used on most hosting platforms.

Preparing the Example Application

No preparation is required for this chapter, which continues using the SportsStore project from Chapter 7.

Tip You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/Apress/pro-angular-5ed>. See Chapter 1 for how to get help if you have problems running the examples.

Adding Progressive Features

A *progressive web application* (PWA) behaves more like a native application, which means it can continue working when there is no network connectivity, its code and content are cached so it can start immediately, and it can use features such as notifications. Progressive web application features are not specific to Angular, but in the sections that follow, I add progressive features to the SportsStore application to show you how it is done.

Tip The process for developing and testing a PWA can be laborious because it can be done only when the application is built for production, which means that the automatic build tools cannot be used.

Installing the PWA Package

The Angular team provides an NPM package that can be used to bring PWA features to Angular projects. Run the command shown in Listing 8-1 in the SportsStore folder to download and install the PWA package.

Tip Notice that this command is `ng add`, rather than the `npm install` command that I use elsewhere for adding packages. The `ng add` command is used specifically to install packages, such as `@angular/pwa`, that have been designed to enhance or reconfigure an Angular project.

Listing 8-1. Installing a Package

```
ng add @angular/pwa
```

Caching the Data URLs

The `@angular/pwa` package configures the application so that HTML, JavaScript, and CSS files are cached, which will allow the application to be started even when there is no network available. I also want the product catalog to be cached so that the application has data to present to the user. In Listing 8-2, I added a new section to the `ngsw-config.json` file, which is used to configure the PWA features for an Angular application and is added to the project by the `@angular/pwa` package.

Listing 8-2. Caching the Data URLs in the `ngsw-config.json` File in the SportsStore Folder

```
{
  "$schema": "./node_modules/@angular/service-worker/config/schema.json",
  "index": "/index.html",
  "assetGroups": [
    {
      "name": "app",
      "installMode": "prefetch",
      "resources": {
        "files": [
          "/favicon.ico",
          "/index.html",
          "/manifest.webmanifest",
          "/*.css",
          "/*.js"
        ]
      }
    },
    {
      "name": "assets",
      "installMode": "lazy",
      "updateMode": "prefetch",
      "resources": {
        "files": [
          "/assets/**",
          "/*.(svg|cur|jpg|jpeg|png|apng|webp|avif|gif|otf|ttf|woff|woff2)"
        ]
      }
    }
  ],
}
```

```

"dataGroups": [
  {
    "name": "api-product",
    "urls": ["/api/products"],
    "cacheConfig" : {
      "maxSize": 100,
      "maxAge": "5d"
    }
  }],
  "navigationUrls": ["/**"]
}

```

The PWA's code and content required to run the application are cached and updated when new versions are available, ensuring that updates are applied consistently when they are available, using the configuration in the assetGroups section of the configuration file.

The application's data is cached using the dataGroups section of the configuration file, which allows data to be managed using its own cache settings. In this listing, I configured the cache so that it will contain data from 100 requests, and that data will be valid for five days. The final configuration section is navigationUrls, which specifies the range of URLs that will be directed to the `index.html` file. In this example, I used a wildcard to match all URLs.

Note I am just touching the surface of the cache features that you can use in a PWA. There are lots of choices available, including the ability to try to connect to the network and then fall back to cached data if there is no connection. See <https://angular.io/guide/service-worker-intro> for details.

Responding to Connectivity Changes

The SportsStore application isn't an ideal candidate for progressive features because connectivity is required to place an order. To avoid user confusion when the application is running without connectivity, I am going to disable the checkout process. The APIs that are used to add progressive features provide information about the state of connectivity and send events when the application goes offline and online. To provide the application with details of its connectivity, I added a file called `connection.service.ts` to the `src/app/model` folder and used it to define the service shown in Listing 8-3.

Listing 8-3. The Contents of the `connection.service.ts` File in the `src/app/model` Folder

```

import { Injectable } from "@angular/core";
import { Observable, Subject } from "rxjs";

@Injectable()
export class ConnectionService {
  private connEvents: Subject<boolean>;

  constructor() {
    this.connEvents = new Subject<boolean>();
    window.addEventListener("online",
      (e) => this.handleConnectionChange(e));
  }
}

```

```

        window.addEventListener("offline",
            (e) => this.handleConnectionChange(e));
    }

    private handleConnectionChange(event: any) {
        this.connEvents.next(this.connected);
    }

    get connected(): boolean {
        return window.navigator.onLine;
    }

    get Changes(): Observable<boolean> {
        return this.connEvents;
    }
}

```

This service presets the connection status to the rest of the application, obtaining the status through the browser's `navigator.onLine` property and responding to the `online` and `offline` events, which are triggered when the connection state changes and which are accessed through the `addEventListener` method provided by the browser. In Listing 8-4, I added the new service to the module for the data model.

Listing 8-4. Adding a Service in the `model.module.ts` File in the `src/app/model` Folder

```

import { NgModule } from "@angular/core";
import { ProductRepository } from "./product.repository";
import { StaticDataSource } from "./static.datasource";
import { Cart } from "./cart.model";
import { Order } from "./order.model";
import { OrderRepository } from "./order.repository";
import { RestDataSource } from "./rest.datasource";
import { HttpClientModule } from "@angular/common/http";
import { AuthService } from "./auth.service";
import { ConnectionService } from "./connection.service";

@NgModule({
  imports: [HttpClientModule],
  providers: [ProductRepository, StaticDataSource, Cart, Order, OrderRepository,
    { provide: StaticDataSource, useClass: RestDataSource },
    RestDataSource, AuthService, ConnectionService]
})
export class ModelModule { }

```

To prevent the user from checking out when there is no connection, I updated the cart detail component so that it receives the connection service in its constructor, as shown in Listing 8-5.

Listing 8-5. Receiving a Service in the `cartDetail.component.ts` File in the `src/app/store` Folder

```

import { Component } from "@angular/core";
import { Cart } from "../model/cart.model";
import { ConnectionService } from "../model/connection.service";

```

```

@Component({
  templateUrl: "cartDetail.component.html"
})
export class CartDetailComponent {
  public connected: boolean = true;

  constructor(public cart: Cart, private connection: ConnectionService) {
    this.connected = this.connection.connected;
    connection.Changes.subscribe((state) => this.connected = state);
  }
}

```

The component defines a connected property that is set from the service and then updated when changes are received. To complete this feature, I changed the checkout button so that it is disabled when there is no connectivity, as shown in Listing 8-6.

Listing 8-6. Reflecting Connectivity in the cartDetail.component.html File in the src/app/store Folder

```

...
<div class="row">
  <div class="col">
    <div class="text-center">
      <button class="btn btn-primary m-1" routerLink="/store">
        Continue Shopping
      </button>
      <button class="btn btn-secondary m-1" routerLink="/checkout"
        [disabled]="cart.lines.length == 0 || !connected">
        {{ connected ? 'Checkout' : 'Offline' }}
      </button>
    </div>
  </div>
</div>
...

```

Preparing the Application for Deployment

In the sections that follow, I prepare the SportsStore application so that it can be deployed.

Creating the Data File

When I created the RESTful web service, I provided the json-server package with a JavaScript file, which is executed each time the server starts and ensures that the same data is always used. That isn't helpful in production, so I added a file called `serverdata.json` to the SportsStore folder with the contents shown in Listing 8-7. When the json-server package is configured to use a JSON file, any changes that are made by the application will be persisted.

Listing 8-7. The Contents of the `serverdata.json` File in the SportsStore Folder

```
{
  "products": [
    { "id": 1, "name": "Kayak", "category": "Watersports",
      "description": "A boat for one person", "price": 275 },
    ...
  ]
}
```

```

    { "id": 2, "name": "Lifejacket", "category": "Watersports",
      "description": "Protective and fashionable", "price": 48.95 },
    { "id": 3, "name": "Soccer Ball", "category": "Soccer",
      "description": "FIFA-approved size and weight", "price": 19.50 },
    { "id": 4, "name": "Corner Flags", "category": "Soccer",
      "description": "Give your playing field a professional touch",
      "price": 34.95 },
    { "id": 5, "name": "Stadium", "category": "Soccer",
      "description": "Flat-packed 35,000-seat stadium", "price": 79500 },
    { "id": 6, "name": "Thinking Cap", "category": "Chess",
      "description": "Improve brain efficiency by 75%", "price": 16 },
    { "id": 7, "name": "Unsteady Chair", "category": "Chess",
      "description": "Secretly give your opponent a disadvantage",
      "price": 29.95 },
    { "id": 8, "name": "Human Chess Board", "category": "Chess",
      "description": "A fun game for the family", "price": 75 },
    { "id": 9, "name": "Bling Bling King", "category": "Chess",
      "description": "Gold-plated, diamond-studded King", "price": 1200 }
  ],
  "orders": []
}

```

Creating the Server

When the application is deployed, I am going to use a single HTTP port to handle the requests for the application and its data, rather than the two ports that I have been using in development. Using separate ports is simpler in development because it means that I can use the Angular development HTTP server without having to integrate the RESTful web service. Angular doesn't provide an HTTP server for deployment, and since I have to provide one, I am going to configure it so that it will handle both types of request and include support for HTTP and HTTPS connections, as explained in the sidebar.

USING SECURE CONNECTIONS FOR PROGRESSIVE WEB APPLICATIONS

When you add progressive features to an application, you must deploy it so that it can be accessed over secure HTTP connections. If you do not, the progressive features will not work because the underlying technology—called *service workers*—won't be allowed by the browser over regular HTTP connections.

You can test progressive features using localhost, as I demonstrate shortly, but an SSL/TLS certificate is required when you deploy the application. If you do not have a certificate, then a good place to start is <https://letsencrypt.org>, where you can get one for free, although you should note that you also need to own the domain or hostname that you intend to deploy to generate a certificate. For this book, I deployed the SportsStore application with its progressive features to sportsstore.adam-freeman.com, which is a domain that I use for development testing and receiving emails. This is not a domain that provides public HTTP services, and you won't be able to access the SportsStore application through this domain.

Run the commands shown in Listing 8-8 in the SportsStore folder to install the packages that are required to create the HTTP/HTTPS server.

Listing 8-8. Installing Additional Packages

```
npm install --save-dev express@4.17.3
npm install --save-dev connect-history-api-fallback@1.6.0
npm install --save-dev https@1.0.0
```

I added a file called `server.js` to the SportsStore with the content shown in Listing 8-9, which uses the newly added packages to create an HTTP and HTTPS server that includes the `json-server` functionality that will provide the RESTful web service. (The `json-server` package is specifically designed to be integrated into other applications.)

Listing 8-9. The Contents of the `server.js` File in the SportsStore Folder

```
const express = require("express");
const https = require("https");
const fs = require("fs");
const history = require("connect-history-api-fallback");
const jsonServer = require("json-server");
const bodyParser = require('body-parser');
const auth = require("./authMiddleware");
const router = jsonServer.router("serverdata.json");

const enableHttps = false;

const ssloptions = {}

if (enableHttps) {
    ssloptions.cert =  fs.readFileSync("./ssl/sportsstore.crt");
    ssloptions.key = fs.readFileSync("./ssl/sportsstore.pem");
}

const app = express();

app.use(bodyParser.json());
app.use(auth);
app.use("/api", router);
app.use(history());
app.use("/", express.static("./dist/SportsStore"));

app.listen(80,
    () => console.log("HTTP Server running on port 80"));

if (enableHttps) {
    https.createServer(ssloptions, app).listen(443,
        () => console.log("HTTPS Server running on port 443"));
} else {
    console.log("HTTPS disabled")
}
```

The server can read the details of the SSL/TLS certificate from files in the `ssl` folder, which is where you should place the files for your certificate. If you do not have a certificate, then you can disable HTTPS by setting the `enableHttps` value to `false`. You will still be able to test the application using the local server, but you won't be able to use the progressive features in deployment.

Changing the Web Service URL in the Repository Class

Now that the RESTful data and the application's JavaScript and HTML content will be delivered by the same server, I need to change the URL that the application uses to get its data, as shown in Listing 8-10.

Listing 8-10. Changing the URL in the `rest.datasource.ts` File in the `src/app/model` Folder

```
import { Injectable } from "@angular/core";
import { HttpClient } from "@angular/common/http";
import { Observable } from "rxjs";
import { Product } from "./product.model";
import { Cart } from "./cart.model";
import { Order } from "./order.model";
import { map } from "rxjs/operators";
import { HttpHeaders } from '@angular/common/http';

const PROTOCOL = "http";
const PORT = 3500;

@Injectable()
export class RestDataSource {
  baseUrl: string;
  auth_token: string;

  constructor(private http: HttpClient) {
    //this.baseUrl = `${PROTOCOL}://${location.hostname}:${PORT}/`;
    this.baseUrl = "/api/"
  }

  // ...methods omitted for brevity...
}
```

FIXING THE BROWSER VERSION ISSUE

At the time of writing, there is a bug that prevents the build process from parsing the output from one of the tools it depends on. Add these entries shown to the `.browserslistrc` file in the `SportsStore` folder:

```
# This file is used by the build system to adjust CSS and JS output to
support the specified browsers below.
# For additional information regarding the format and rule options,
please see:
# https://github.com/browserslist/browserslist#queries
```

```
# For the full list of supported browsers by the Angular framework,  
please see:  
# https://angular.io/guide/browser-support  
  
# You can see what browsers were selected by your queries by running:  
#   npx browserslist  
  
last 1 Chrome version  
last 1 Firefox version  
last 2 Edge major versions  
last 2 Safari major versions  
last 2 iOS major versions  
Firefox ESR  
  
not ios_saf 15.2-15.3  
not safari 15.2-15.3
```

This issue may have been resolved by the time you read this book, but the changes allow the build process to complete.

Building and Testing the Application

To build the application for production, run the command shown in Listing 8-11 in the SportsStore folder.

Listing 8-11. Building the Application for Production

```
ng build
```

This command builds an optimized version of the application without the additions that support the development tools. The output from the build process is placed in the `dist/SportsStore` folder. In addition to the JavaScript files, there is an `index.html` file that has been copied from the `SportsStore/src` folder and modified to use the newly built files.

Note Angular provides support for server-side rendering, where the application is run in the server, rather than the browser. This is a technique that can improve the perception of the application's startup time and can improve indexing by search engines. This is a feature that should be used with caution because it has serious limitations and can undermine the user experience. For these reasons, I have not covered server-side rendering in this book. You can learn more about this feature at <https://angular.io/guide/universal>.

The build process can take a few minutes to complete. Once the build is ready, run the command shown in Listing 8-12 in the SportsStore folder to start the HTTP server. If you have not configured the server to use a valid SSL/TLS certificate, you should change the value of the `enableHttps` constant in the `server.js` file and then run the command in Listing 8-12.

Listing 8-12. Starting the Production HTTP Server

```
node server.js
```

Once the server has started, open a new browser window and navigate to `http://localhost`, and you will see the familiar content shown in Figure 8-1.

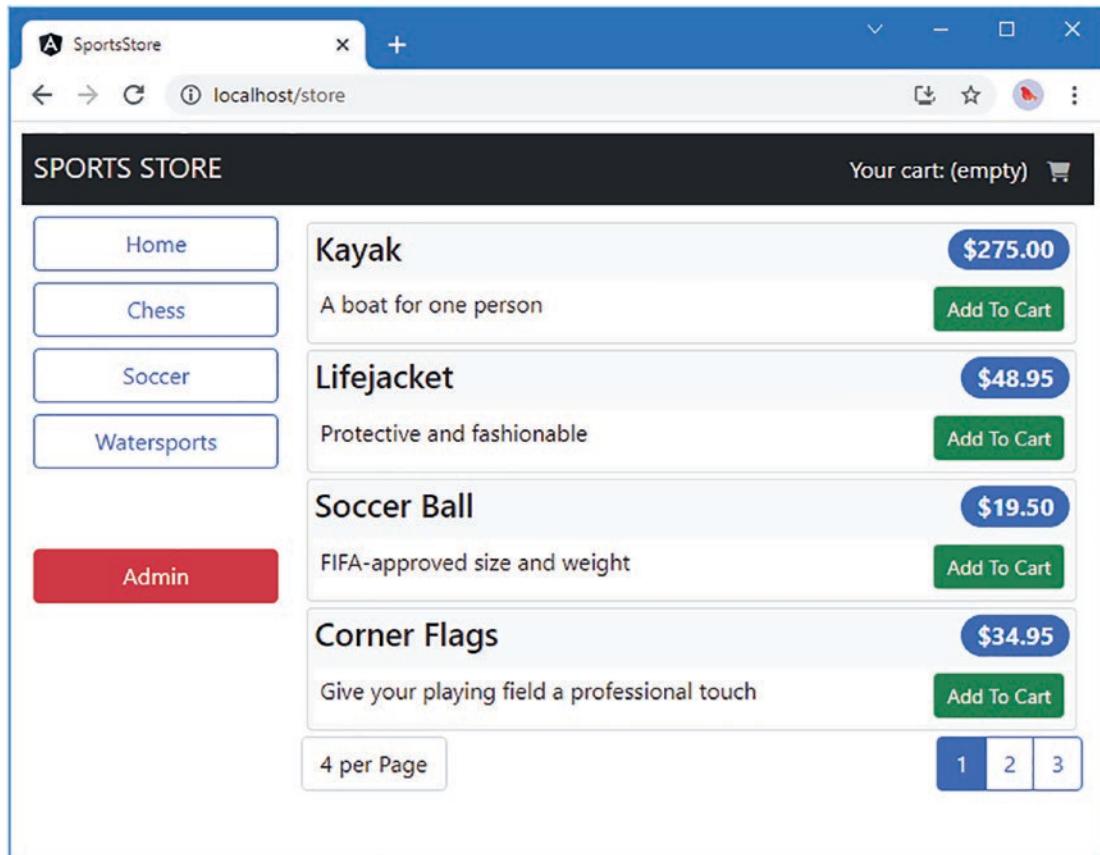


Figure 8-1. Testing the application

Testing the Progressive Features

Open the F12 development tools, navigate to the Network tab, click the arrow to the right of Online, and select Offline, as shown in Figure 8-2. This simulates a device without connectivity, but since SportsStore is a progressive web application, it has been cached by the browser, along with its data.

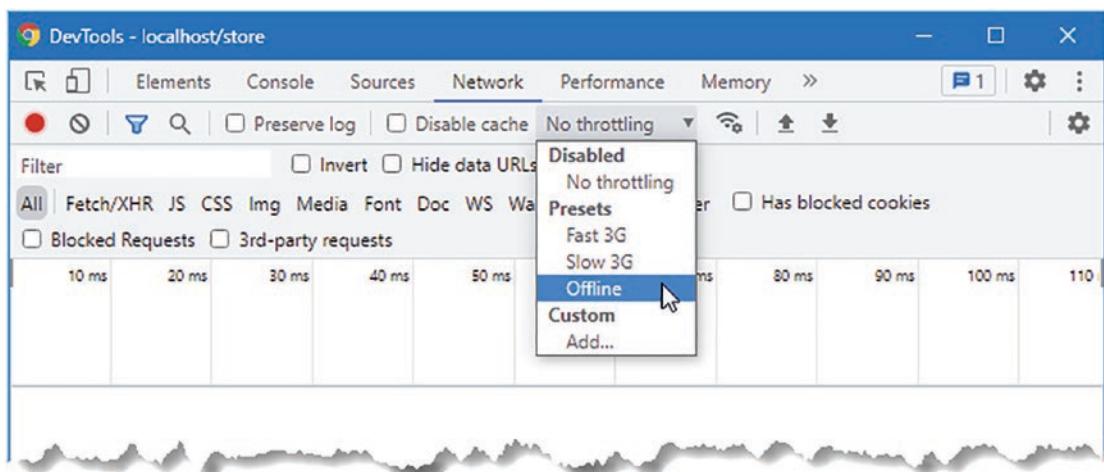


Figure 8-2. Going offline

Once the application is offline, the application will be loaded from the browser's cache. If you click an Add To Cart button, you will see that the Checkout button is disabled, as shown in Figure 8-3. Uncheck the Offline checkbox, and the button's text will change so that the user can place an order.

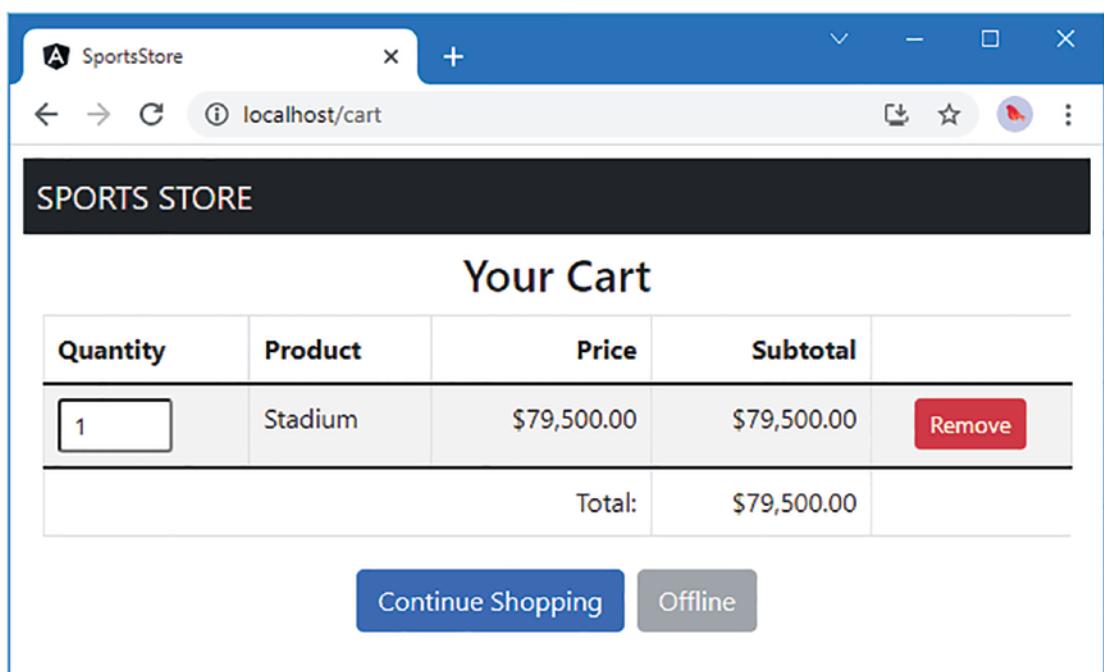


Figure 8-3. Reflecting the connection status in the application

Containerizing the SportsStore Application

To complete this chapter, I am going to create a container for the SportsStore application so that it can be deployed into production. At the time of writing, Docker is the most popular way to create containers, which is a pared-down version of Linux with just enough functionality to run the application. Most cloud platforms or hosting engines have support for Docker, and its tools run on the most popular operating systems.

Installing Docker

The first step is to download and install the Docker tools on your development machine, which are available from www.docker.com/products. There are versions for macOS, Windows, and Linux, and there are some specialized versions to work with the Amazon and Microsoft cloud platforms. The free Community edition of Docker Desktop is sufficient for this chapter.

Preparing the Application

The first step is to create a configuration file for NPM that will be used to download the additional packages required by the application for use in the container. I created a file called `deploy-package.json` in the SportsStore folder with the content shown in Listing 8-13.

Listing 8-13. The Contents of the `deploy-package.json` File in the SportsStore Folder

```
{
  "dependencies": {
    "@fortawesome/fontawesome-free": "6.0.0",
    "bootstrap": "5.1.3"
  },
  "devDependencies": {
    "json-server": "0.17.0",
    "jsonwebtoken": "8.5.1",
    "express": "4.17.3",
    "https": "1.0.0",
    "connect-history-api-fallback": "1.6.0"
  },
  "scripts": {
    "start": "node server.js"
  }
}
```

The `dependencies` section omits Angular and all of the other runtime packages that were added to the `package.json` file when the project was created because the build process incorporates all of the JavaScript code required by the application into the files in the `dist/SportsStore` folder. The `devDependencies` section includes the tools required by the production HTTP/HTTPS server.

The `scripts` section of the `deploy-package.json` file is set up so that the `npm start` command will start the production server, which will provide access to the application and its data.

Creating the Docker Container

To define the container, I added a file called `Dockerfile` (with no extension) to the `SportsStore` folder and added the content shown in Listing 8-14.

Listing 8-14. The Contents of the Dockerfile File in the SportsStore Folder

```
FROM node:16.3.0

RUN mkdir -p /usr/src/sportsstore

COPY dist/SportsStore /usr/src/sportsstore/dist/SportsStore
COPY ssl /usr/src/sportsstore/ssl

COPY authMiddleware.js /usr/src/sportsstore/
COPY serverdata.json /usr/src/sportsstore/
COPY server.js /usr/src/sportsstore/server.js
COPY deploy-package.json /usr/src/sportsstore/package.json

WORKDIR /usr/src/sportsstore

RUN npm install

EXPOSE 80

CMD ["node", "server.js"]
```

The contents of the `Dockerfile` use a base image that has been configured with Node.js and copies the files required to run the application, including the bundle file containing the application and the `package.json` file that will be used to install the packages required to run the application in deployment.

To speed up the containerization process, I created a file called `.dockerignore` in the `SportsStore` folder with the content shown in Listing 8-15. This tells Docker to ignore the `node_modules` folder, which is not required in the container and takes a long time to process.

Listing 8-15. The Contents of the `.dockerignore` File in the SportsStore Folder

```
node_modules
```

Run the command shown in Listing 8-16 in the `SportsStore` folder to create an image that will contain the `SportsStore` application, along with all of the tools and packages it requires.

Tip The `SportsStore` project must contain an `ssl` directory, even if you have not installed a certificate. This is because there is no way to check to see whether a file exists when using the `COPY` command in the `Dockerfile`.

Listing 8-16. Building the Docker Image

```
docker build . -t sportsstore -f Dockerfile
```

An image is a template for containers. As Docker processes the instructions in the Dockerfile, the NPM packages will be downloaded and installed, and the configuration and code files will be copied into the image.

Running the Application

Once the image has been created, create and start a new container using the command shown in Listing 8-17.

Tip Make sure you stop the test server you started in Listing 8-12 before starting the Docker container since both use the same ports to listen for requests.

Listing 8-17. Starting the Docker Container

```
docker run -p 80:80 -p 443:443 sportsstore
```

You can test the application by opening `http://localhost` in the browser, which will display the response provided by the web server running in the container, as shown in Figure 8-4.

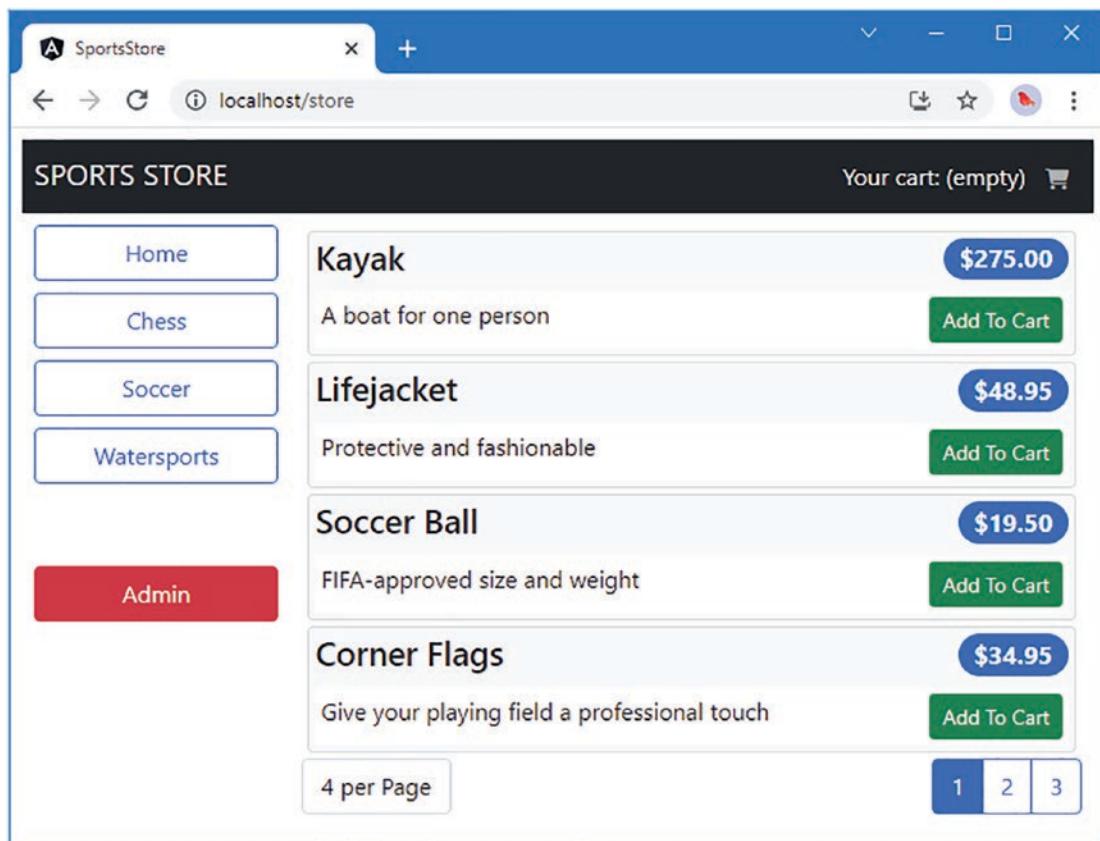


Figure 8-4. Running the containerized SportsStore application

To stop the container, run the command shown in Listing 8-18.

Listing 8-18. Listing the Containers

```
docker ps
```

You will see a list of running containers, like this (I have omitted some fields for brevity):

CONTAINER ID	IMAGE	COMMAND	CREATED
ecc84f7245d6	sportssstore	"docker-entrypoint.s..."	33 seconds ago

Using the value in the Container ID column, run the command shown in Listing 8-19.

Listing 8-19. Stopping the Container

```
docker stop ecc84f7245d6
```

The application is ready to deploy to any platform that supports Docker, although the progressive features will work only if you have configured an SSL/TLS certificate for the domain to which the application is deployed.

Summary

This chapter completes the SportsStore application, showing how an Angular application can be prepared for deployment and how easy it is to put an Angular application into a container such as Docker. That's the end of this part of the book. In Part 2, I begin the process of digging into the details and show you how the features I used to create the SportsStore application work in depth.

CHAPTER 9



Understanding Angular Projects and Tools

In this chapter, I explain the structure of an Angular project and the tools that are used for development. By the end of the chapter, you will understand how the parts of a project fit together and have a foundation on which to apply the more advanced features that are described in the chapters that follow.

Tip You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/Apress/pro-angular-5ed>. See Chapter 1 for how to get help if you have problems running the examples.

Creating a New Angular Project

The `angular-cli` package you installed in Chapter 1 contains all the functionality required to create a new Angular project that contains some placeholder content to jump-start development, and it contains a set of tightly integrated tools that are used to build, test, and prepare Angular applications for deployment.

To create a new Angular project, open a command prompt, navigate to a convenient location, and run the command shown in Listing 9-1.

Listing 9-1. Creating a Project

```
ng new example --routing false --style css --skip-git --skip-tests
```

The `ng new` command creates new projects, and the argument is the project name, which is `example` in this case. The `ng new` command has a set of arguments that shape the project that is created; Table 9-1 describes the most useful.

Table 9-1. Useful `ng new` Options

Argument	Description
<code>--directory</code>	This option is used to specify the name of the directory for the project. It defaults to the project name.
<code>--force</code>	When true, this option overwrites any existing files.
<code>--minimal</code>	This option creates a project without adding support for testing frameworks.
<code>--package-manager</code>	This option is used to specify the package manager that will be used to download and install the packages required by Angular. If omitted, NPM will be used. Other options are <code>yarn</code> , <code>pnpm</code> , and <code>cnpm</code> . The default package manager is suitable for most projects.
<code>--prefix</code>	This option applies a prefix to all of the component selectors, as described in the “Understanding How an Angular Application Works” section.
<code>--routing</code>	This option is used to create a routing module in the project. I explain how the routing feature works in detail in Chapters 24–26.
<code>--skip-git</code>	Using this option prevents a Git repository from being created in the project. You must install the Git tools if you create a project without this option.
<code>--skip-install</code>	This option prevents the initial operation that downloads and installs the packages required by Angular applications and the project’s development tools.
<code>--skip-tests</code>	This option prevents the addition of the initial configuration for testing tools.
<code>--style</code>	This option specifies how stylesheets are handled. I use the <code>css</code> option throughout this book, but popular CSS preprocessors such as SCSS, SASS, and LESS also are supported. Chapter 28 contains an advanced example that uses SCSS.

The project initialization process performed by the `ng new` command can take some time to complete because there are a large number of packages required by the project, both to run the Angular application and for the development and testing tools that I describe in this chapter.

Understanding the Project Structure

Use your preferred code editor to open the example folder, and you will see the files and folder structure shown in Figure 9-1. The figure shows the way that Visual Studio Code presents the project; other editors may present the project contents differently.

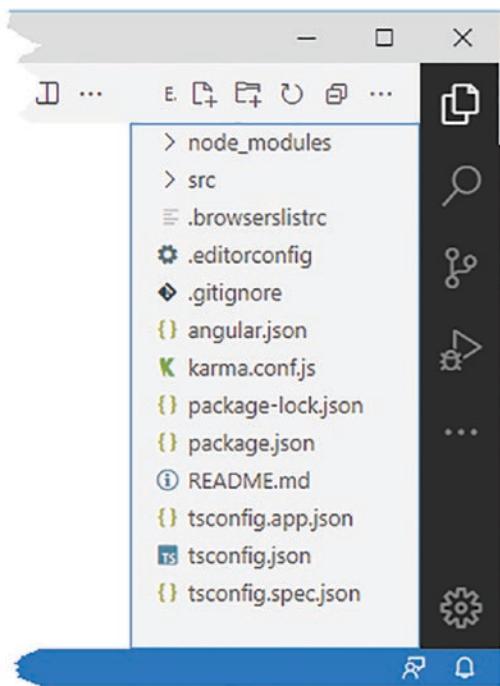


Figure 9-1. The contents of a new Angular project

Table 9-2 describes the files and folders that are added to a new project by the `ng new` command and that provide the starting point for most Angular development.

Table 9-2. The Files and Folders in a New Angular Project

Name	Description
node_modules	This folder contains the NPM packages that are required for the application and for the Angular development tools, as described in the “Understanding the Packages Folder” section.
src	This folder contains the application’s source code, resources, and configuration files, as described in the “Understanding the Source Code Folder” section.
.browserslistrc	This file is used to specify the browsers that the application will support, which can alter the way that code is compiled and prepared for distribution. The default settings are suitable for most projects, but you can find details of how to change the target browsers at https://github.com/browserslist/browserslist .
.editorconfig	This file contains settings that configure text editors. Not all editors respond to this file, but it may override the preferences you have defined. You can learn more about the editor settings that can be set in this file at http://editorconfig.org .
.gitignore	This file contains a list of files and folders that are excluded from version control when using Git.
angular.json	This file contains the configuration for the Angular development tools.
karma.conf.js	This file configures the Karma test runner. See Chapter 29 for details of unit testing in Angular projects.
package.json	This file contains details of the NPM packages required by the application and the development tools and defines the commands that run the development tools, as described in the “Understanding the Packages Folder” section.
package-lock.json	This file contains version information for all the packages that are installed in the node_modules folder, as described in the “Understanding the Packages Folder” section.
README.md	This is a readme file that contains the list of commands for the development tools, which are described in the “Using the Development Tools” section.
tsconfig.json	This file contains the configuration settings for the TypeScript compiler. You don’t need to change the compiler configuration in most Angular projects.
tsconfig.app.json	This file contains additional configuration options for the TypeScript compiler related to the locations of source files, type definition files, and where compiled output will be written.
tsconfig.spec.json	This file contains additional configuration options for the TypeScript compiler related to the locations of source files for unit tests.

You won’t need all these files in every project, and you can remove the ones you don’t require. I tend to remove the README.md, .editorconfig, and .gitignore files, for example, because I am already familiar with the tool commands, I prefer not to override my editor settings, and I don’t use Git for version control.

Understanding the Source Code Folder

The `src` folder contains the application's files, including the source code and static assets, such as images. This folder is the focus of most development activities, and Figure 9-2 shows the contents of the `src` folder created using the `ng new` command.

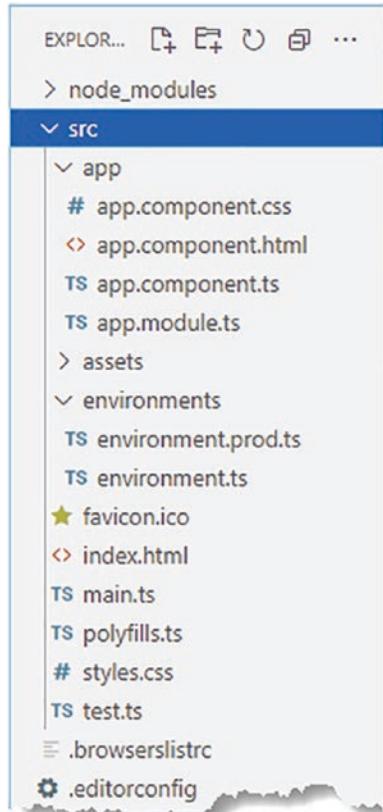


Figure 9-2. The contents of the `src` folder

The `app` folder is where you will add the custom code and content for your application, and its structure becomes more complex as you add features. The other files support the development process, as described in Table 9-3.

Table 9-3. The Files and Folders in the src Folder

Name	Description
app	This folder contains an application's source code and content. The contents of this folder are the topic of the "Understanding How an Angular Application Works" section and other chapters in this part of the book.
assets	This folder is used for the static resources required by the application, such as images.
environments	This folder contains configuration files that define settings for different environments. By default, the only configuration setting is the production flag, which is set to true when the application is built for deployment, as explained in the "Understanding the Application Bootstrap" section.
favicon.ico	This file contains an icon that browsers will display in the tab for the application. The default image is the Angular logo.
index.html	This is the HTML file that is sent to the browser during development, as explained in the "Understanding the HTML Document" section.
main.ts	This file contains the TypeScript statements that start the application when they are executed, as described in the "Understanding the Application Bootstrap" section.
polyfills.ts	This file is used to include polyfills in the project to provide support for features that are not available natively in some browsers.
styles.css	This file is used to define CSS styles that are applied throughout the application.
tests.ts	This is the configuration file for the Karma test package, which I describe in Chapter 29.

Understanding the Packages Folder

The world of JavaScript application development depends on a rich ecosystem of packages, some of which contain the Angular framework that will be sent to the browser through small packages that are used behind the scenes during development. A lot of packages are required for an Angular project; the example project created at the start of this chapter, for example, requires more than 850 packages.

Many of these packages are just a few lines of code, but there is a complex hierarchy of dependencies between them that is too large to manage manually, so a package manager is used. The package manager is given an initial list of packages required for the project. Each of these packages is then inspected for its dependencies, and the process continues until the complete set of packages has been created. All the required packages are downloaded and installed in the `node_modules` folder.

The initial set of packages is defined in the `package.json` file using the `dependencies` and `devDependencies` properties. The `dependencies` property is used to list the packages that the application will require to run. Here are the `dependencies` packages from the `package.json` file in the example application, although you may see different version numbers in your project:

```
...
"dependencies": {
  "@angular/animations": "~13.0.0",
  "@angular/common": "~13.0.0",
  "@angular/compiler": "~13.0.0",
  "@angular/core": "~13.0.0",
  "@angular/forms": "~13.0.0",
```

```

"@angular/platform-browser": "~13.0.0",
"@angular/platform-browser-dynamic": "~13.0.0",
"@angular/router": "~13.0.0",
"rxjs": "~7.4.0",
"tslib": "^2.3.0",
"zone.js": "~0.11.4"
},
...

```

Most of the packages provide Angular functionality, with a handful of supporting packages that are used behind the scenes. For each package, the package.json file includes details of the version numbers that are acceptable, using the format described in Table 9-4.

Table 9-4. The Package Version Numbering System

Format	Description
13.0.0	Expressing a version number directly will accept only the package with the exact matching version number, e.g., 13.0.0.
*	Using an asterisk accepts any version of the package to be installed.
>13.0.0	Prefixing a version number with > or >= accepts any version of the package that is greater than
=13.0.0	or greater than or equal to a given version.
<13.0.0	Prefixing a version number with < or <= accepts any version of the package that is less than or
=13.0.0	less than or equal to a given version.
~13.0.0	Prefixing a version number with a tilde (the ~ character) accepts versions to be installed even if the patch level number (the last of the three version numbers) doesn't match. For example, specifying ~13.0.0 means you will accept version 13.0.1 or 13.0.2 (which would contain patches to version 13.0.0) but not version 13.1.0 (which would be a new minor release).
^13.0.0	Prefixing a version number with a caret (the ^ character) will accept versions even if the minor release number (the second of the three version numbers) or the patch number doesn't match. For example, specifying ^13.0.0 means you will accept versions 13.1.0, and 13.2.0, for example, but not version 14.0.0.

The version numbers specified in the dependencies section of the package.json file will accept minor updates and patches. Version flexibility is more important when it comes to the devDependencies section of the file, which contains a list of the packages that are required for development but which will not be part of the finished application. There are 19 packages listed in the devDependencies section of the package.json file in the example application, each of which has a range of acceptable versions.

```

...
"devDependencies": {
  "@angular-devkit/build-angular": "~13.0.3",
  "@angular/cli": "~13.0.3",
  "@angular/compiler-cli": "~13.0.0",
  "@types/jasmine": "~3.10.0",
  "@types/node": "^12.11.1",
  "jasmine-core": "~3.10.0",
  "karma": "~6.3.0",
  "karma-chrome-launcher": "~3.1.0",
}

```

```

    "karma-coverage": "~2.0.3",
    "karma-jasmine": "~4.0.0",
    "karma-jasmine-html-reporter": "~1.7.0",
    "typescript": "~4.4.3"
}
...

```

Once again, you may see different details, but the key point is that the management of dependencies between packages is too complex to do manually and is delegated to a package manager. The most widely used package manager is NPM, which is installed alongside Node.js and was part of the preparations for this book in Chapter 2.

The packages required for basic development are automatically downloaded and installed into the `node_modules` folder when you create a project, but Table 9-5 lists some commands that you may find useful during development. All these commands should be run inside the project folder, which is the one that contains the `package.json` file.

UNDERSTANDING GLOBAL AND LOCAL PACKAGES

NPM can install packages so they are specific to a single project (known as a *local install*) or so they can be accessed from anywhere (known as a *global install*). Few packages require global installs, but one exception is the `@angular/cli` package installed in Chapter 2 as part of the preparations for this book. The `@angular-cli` package requires a global install because it is used to create new projects. The individual packages required for the project are installed locally, into the `node_modules` folder.

Table 9-5. Useful NPM Commands

Command	Description
<code>npm install</code>	This command performs a local install of the packages specified in the <code>package.json</code> file.
<code>npm install package@version</code>	This command performs a local install of a specific version of a package and updates the <code>package.json</code> file to add the package to the <code>dependencies</code> section.
<code>npm install package@version --save-dev</code>	This command performs a local install of a specific version of a package and updates the <code>package.json</code> file to add the package to the <code>devDependencies</code> section.
<code>npm install --global package@version</code>	This command performs a global install of a specific version of a package.
<code>npm list</code>	This command lists all of the local packages and their dependencies.
<code>npm run <script name></code>	This command executes one of the scripts defined in the <code>package.json</code> file, as described next.
<code>npx package@version</code>	This command downloads and executes a package.

The last two commands described in Table 9-5 are oddities, but package managers have traditionally included support for running commands that are defined in the `scripts` section of the `package.json` file. In an Angular project, this feature is used to provide access to the tools that are used during development and that prepare the application for deployment. Here is the `scripts` section of the `package.json` file in the example project:

```

...
"scripts": {
  "ng": "ng",
  "start": "ng serve",
  "build": "ng build",
  "watch": "ng build --watch --configuration development",
  "test": "ng test"
},
...

```

Table 9-6 summarizes these commands, and I demonstrate their use in later sections of this chapter or in later chapters in this part of the book.

Table 9-6. The Commands in the scripts Section of the package.json File

Name	Description
ng	This command runs the ng command, which provides access to the Angular development tools.
start	This command starts the development tools and is equivalent to the ng serve command.
build	This command performs the production build process.
test	This command starts the unit testing tools, which are described in Chapter 29, and is equivalent to the ng test command.

These commands are run by using `npm run` followed by the name of the command that you require, and this must be done in the folder that contains the `package.json` file. So, if you want to run the `test` command in the example project, navigate to the `example` folder and type `npm run test`. You can get the same result by using the command `ng test`.

The `npx` command is useful for downloading and executing a package in a single command, which I use in the “Running the Production Build” section later in the chapter. Not all packages are set up for use with `npx`, which is a recent feature.

Adding Packages with Schematics to an Angular Project

As noted in Table 9-5, the `npm install` command can be used to add a JavaScript package to the project. Packages installed with this command are added to the `node_modules` folder and then typically require some manual integration to make them part of the Angular application. You can see an example of this in the “Understanding the Styles Bundle” section, where I install the popular Bootstrap CSS framework and configure Angular to include its CSS stylesheet in the content sent to the browser.

Some JavaScript packages take advantage of the *schematics API* provided by the `@angular/cli` package to automate the integration process. Typically, this is because the package provides Angular-specific functionality, such as the Angular Material package, but some package authors provide schematics because Angular is so widely used. The `npm install` command doesn’t understand the schematics API, so the `ng add` command is used to download these packages and perform the integration. Run the command shown in Listing 9-2 in the example folder to install the Angular Material package.

Listing 9-2. Installing the Angular Material Package

```
ng add @angular/material@13.0.2
```

The schematics API allows package authors to ask the user questions and use the responses during the integration process. As you go through the setup for Angular Material, you will be asked four questions, and you can press the Enter key to select the default answer for each one.

The first question is just a request to confirm that you want to install the package:

```
Using package manager: npm
Package information loaded.
The package @angular/material@13.0.2 will be installed and executed.
Would you like to proceed? (Y/n)
```

The `ng add` command uses the package manager selected when the Angular project was created to download the package. The example project was created to use the `npm` package manager, but, as noted earlier, other package managers can be selected, and these will be used automatically by the `ng add` command.

The remaining questions are specific to Angular Material and allow the theme to be selected and typography and animation options to be configured:

```
? Choose a prebuilt theme name, or "custom" for a custom theme: (Use arrow keys)
> Indigo/Pink      [ Preview: https://material.angular.io?theme=indigo-pink ]
Deep Purple/Amber  [ Preview: https://material.angular.io?theme=deeppurple-amber ]
Pink/Blue Grey    [ Preview: https://material.angular.io?theme=pink-bluegrey ]
Purple/Green       [ Preview: https://material.angular.io?theme=purple-green ]
Custom
? Set up global Angular Material typography styles? (y/N)
? Set up browser animations for Angular Material? (Y/n)
```

Each package that uses the schematics API will ask questions, and once you have made your choices, the package will be integrated into the Angular project, and the list of files that are changed is shown:

```
UPDATE src/app/app.module.ts (423 bytes)
UPDATE angular.json (3770 bytes)
UPDATE src/index.html (552 bytes)
UPDATE src/styles.css (181 bytes)
```

I describe all of these files in later sections, and you will see the additions that the Angular Material package has made to each of them.

Note You don't need to understand the schematics API to add packages to an Angular project. But if you are interested in publishing a package for use by Angular developers, then you can learn about the features available at <https://angular.io/guide/schematics-authoring>.

Using the Development Tools

Projects created using the `ng new` command include a complete set of development tools that monitor the application's files and build the project when a change is detected. Run the command shown in Listing 9-3 in the example folder to start the development tools.

Listing 9-3. Starting the Development Tools

```
ng serve
```

The command starts the build process, which produces messages like these at the command prompt:

```
...
Generating browser application bundles (phase: building)...
...
```

At the end of the process, you will see a summary of the bundles that have been created, like this:

```
...
Browser application bundle generation complete.

Initial Chunk Files | Names           |      Size
vendor.js           | vendor          | 1.89 MB
polyfills.js        | polyfills       | 339.12 kB
styles.css, styles.js | styles          | 289.27 kB
main.js             | main            | 51.79 kB
runtime.js          | runtime         | 6.85 kB

| Initial Total | 2.57 MB
```

```
Build at: 2021-12-05T07:51:57.541Z - Hash: 5762f0ed7a6c45f4 - Time: 10531ms
```

```
** Angular Live Development Server is listening on localhost:4200, open your browser on
http://localhost:4200/ **
```

```
Compiled successfully.
...
```

Understanding the Development HTTP Server

To simplify the development process, the project incorporates an HTTP server that is tightly integrated with the build process. After the initial build process, the HTTP server is started, and a message is displayed that tells you which port is being used to listen for requests, like this:

```
...
** Angular Live Development Server is listening on localhost:4200, open your browser on
http://localhost:4200/ **
...
```

The default is port 4200, but you may see a different message if you are already using port 4200. Open a new browser window and request `http://localhost:4200`; you will see the placeholder content added to the project by the `ng new` command, as shown in Figure 9-3.

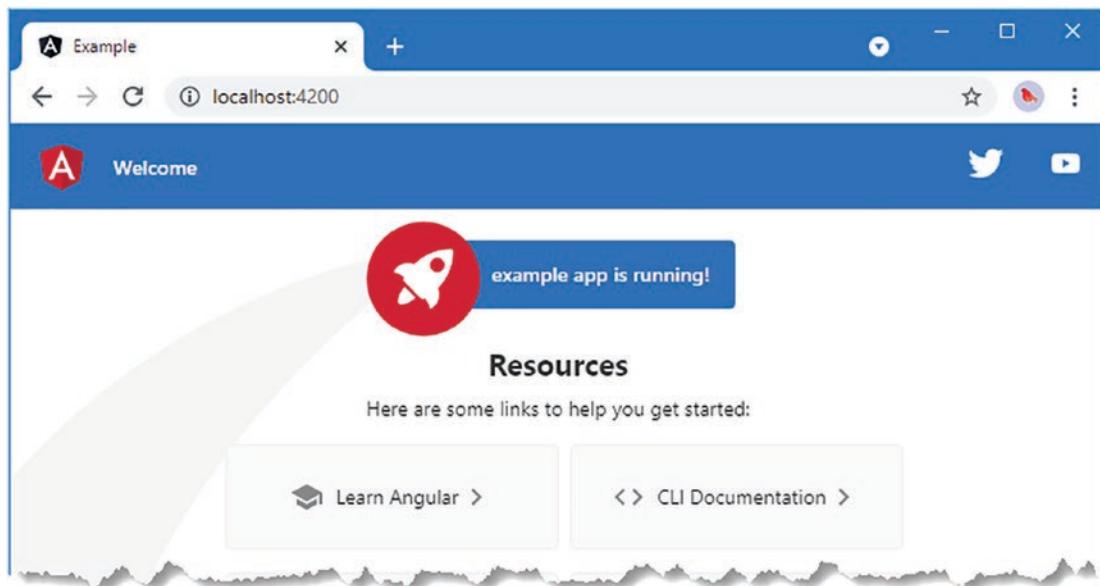


Figure 9-3. Using the HTTP development server

Understanding the Build Process

When you run `ng serve`, the project is built so that it can be used by the browser. This is a process that requires three important tools: the TypeScript compiler, the Angular compiler, and a package named `webpack`.

Angular applications are created using TypeScript files and HTML templates containing expressions, neither of which can be understood by browsers. The TypeScript compiler is responsible for compiling the TypeScript files into JavaScript, and the Angular compiler is responsible for transforming templates into JavaScript statements that use the browser APIs to create the HTML elements in the template file and evaluate the expressions they contain.

The build process is managed through `webpack`, which is a module bundler, meaning that it takes the compiled output and consolidates it into a module that can be sent to the browser. This process is known as *bundling*, which is a bland description for an important function, and it is one of the key tools that you will rely on while developing an Angular application, albeit one that you won't deal with directly since it is managed for you by the Angular development tools.

When you run the `ng serve` command, you will see a series of messages as `webpack` processes the application. `Webpack` starts with the code in the `main.ts` file, which is the entry point for the application and follows the `import` statements it contains to discover its dependencies, repeating this process for each file on which there is a dependency. `Webpack` works its way through the `import` statements, compiling each TypeScript and template file on which a dependency is declared to produce JavaScript code for the entire application.

Note This section describes the development build process. See the “Understanding the Production Build Process” section for details of the process used to prepare an application for deployment.

The output from the `main.ts` compilation process is combined into a single file, known as a *bundle*. During the bundling process, webpack generates multiple bundles, each of which contains resources required by the application. At the end of the process, you will see a summary of the bundles that have been created, like this:

Initial Chunk Files	Names	Size
<code>vendor.js</code>	<code>vendor</code>	1.89 MB
<code>polyfills.js</code>	<code>polyfills</code>	339.12 kB
<code>styles.css, styles.js</code>	<code>styles</code>	289.27 kB
<code>main.js</code>	<code>main</code>	51.79 kB
<code>runtime.js</code>	<code>runtime</code>	6.85 kB
	<code>Initial Total</code>	2.57 MB

...

The initial build process can take a while to complete because five bundles are produced, as described in Table 9-7.

Table 9-7. The Bundles Produced by the Angular Build Process

Name	Description
<code>main.js</code>	This file contains the compiled output produced from the <code>src/app</code> folder.
<code>polyfills.js</code>	This file contains JavaScript polyfills required for features used by the application that are not supported by the target browsers.
<code>runtime.js</code>	This file contains the code that loads the other modules.
<code>styles.js</code>	This file contains JavaScript code that adds the application’s global CSS stylesheets.
<code>vendor.js</code>	This file contains the third-party packages the application depends on, including the Angular packages.

Understanding the Application Bundle

The full build process is performed only when the `ng serve` command is first run. Thereafter, bundles are rebuilt if the files they are composed of change. You can see this by replacing the contents of the `app.component.html` file with the elements shown in Listing 9-4.

Listing 9-4. Replacing the Contents of the `app.component.html` File in the `src/app` Folder

```
<div>
    Hello, World
</div>
```

When you save the changes, only the affected bundles will be rebuilt, and you will see messages at the command prompt like this:

```
Initial Chunk Files | Names      |      Size
runtime.js          | runtime    | 6.85 kB
main.js            | main       | 5.96 kB
3 unchanged chunks
Build at: 2021-12-05T08:34:49.753Z - Hash: 387bfffb37bb32d06 - Time: 243ms
Compiled successfully.
```

Selectively compiling files and preparing bundles ensures that the effect of changes during development can be seen quickly. Figure 9-4 shows the effect of the change in Listing 9-4.

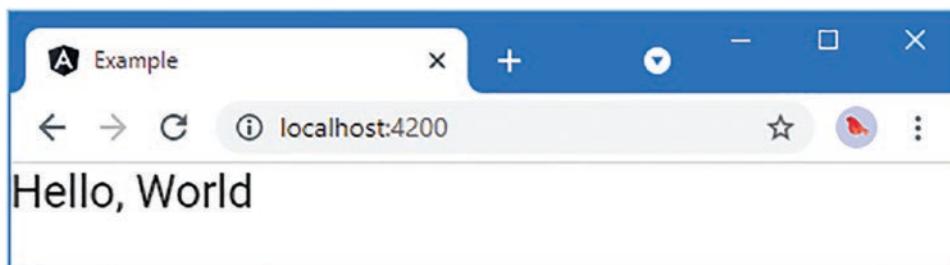


Figure 9-4. Changing a file used in the main.js bundle

UNDERSTANDING HOT RELOADING

During development, the Angular development tools add support for a feature called *hot reloading*. This is the feature that meant you saw the effect of the change in Listing 9-4 automatically. The JavaScript code added to the bundle opens a connection back to the Angular development HTTP server. When a change triggers a build, the server sends a signal over the HTTP connection, which causes the browser to reload the application automatically.

Understanding the Polyfills Bundle

The Angular build process targets the most recent versions of browsers by default, which can be a problem if you need to provide support for older browsers (something that commonly arises in corporate applications where old browsers are often). The `polyfills.js` bundle is used to provide implementations of JavaScript features to older versions that do not have native support. The content of the `polyfills.js` file is determined by the `polyfills.ts` file, which can be found in the `src` folder. Only one polyfill is enabled by default, which enables the `Zone.js` library, which is used by Angular to perform change detection in browsers. You can add your own polyfills to the bundle by adding `import` statements to the `polyfills.ts` file.

Understanding the Styles Bundle

The `styles.js` bundle is used to add CSS stylesheets to the application. The bundle file contains JavaScript code that uses the browser API to define styles, along with the contents of the CSS stylesheets the application requires. (It may seem counterintuitive to use JavaScript to distribute a CSS file, but it works well and has the advantage of making the application self-contained so that it can be deployed as a series of JavaScript files that do not rely on additional assets to be set up on the deployment web servers.)

CSS stylesheets are added to the application using the `styles` section of the `angular.json` file. Run the command shown in Listing 9-5 in the `example` folder to see the current set of stylesheets included in the styles bundle.

Listing 9-5. Displaying the Configured Stylesheets

```
ng config "projects.example.architect.build.options.styles"
```

The `ng config` command is used to get and change configuration settings in the `angular.json` file. The argument to the `ng config` command in Listing 9-5 selects the `projects.example.architect.build.options.styles` setting, which defines the stylesheets that are included in the styles bundle and produces the following results:

```
[  
  "./node_modules/@angular/material/prebuilt-themes/indigo-pink.css",  
  "src/styles.css"  
]
```

The `indigo-pink.css` file was added to the list when the Angular Material package was installed. The `styles.css` file was added to the list as part of the initial configuration when the project was created.

The structure of the `angular.json` file and the effect of the settings it contains are described at <https://angular.io/guide/workspace-config>. Using this description, I was able to determine the configuration option for the CSS stylesheets.

Many projects require no direct changes to the `angular.json` file and can rely on the default settings. One exception is when manual integration is required for packages that don't use the schematics API. Run the command shown in Listing 9-6 in the `example` folder to install the popular Bootstrap CSS framework.

Listing 9-6. Adding a Package to the Project

```
npm install bootstrap@5.1.3
```

The Bootstrap package isn't specific to Angular development and doesn't use the schematics API, which means that a manual change must be made to the `angular.config` file to include the Bootstrap CSS stylesheet in the styles bundle, which is done by running the command shown in Listing 9-7 in the `example` folder. Take care to enter the command exactly as shown and do not introduce additional spaces or quotes.

Listing 9-7. Changing the Application Configuration

```
ng config projects.example.architect.build.options.styles \
'["./node_modules/@angular/material/prebuilt-themes/indigo-pink.css", \
"src/styles.css", \
"node_modules/bootstrap/dist/css/bootstrap.min.css"]'
```

If you are using Windows, then use a PowerShell prompt to run the command shown in Listing 9-8 in the example folder.

Listing 9-8. Changing the Application Configuration Using PowerShell

```
ng config projects.example.architect.build.options.styles ` 
'["./node_modules/@angular/material/prebuilt-themes/indigo-pink.css",
"src/styles.css",
"node_modules/bootstrap/dist/css/bootstrap.min.css"]'
```

Check that the stylesheet has been added to the configuration by running the command in Listing 9-5 again, which should produce the following result:

```
[ 
  "./node_modules/@angular/material/prebuilt-themes/indigo-pink.css",
  "src/styles.css",
  "node_modules/bootstrap/dist/css/bootstrap.min.css"
]
```

EDITING THE CONFIGURATION FILE DIRECTLY

The commands to edit the configuration can be difficult to enter correctly, and it is easy to mistype the character escape sequences required to ensure that the command prompt passes the setting to the `ng config` command in the format it expects.

An alternative approach is to edit the `angular.json` file directly and add the stylesheet to the `styles` section, like this:

```
...
"architect": {
  "build": {
    "builder": "@angular-devkit/build-angular:browser",
    "options": {
      "outputPath": "dist/example",
      "index": "src/index.html",
      "main": "src/main.ts",
      "polyfills": "src/polyfills.ts",
      "tsConfig": "tsconfig.app.json",
      "assets": [
```

```

    "src/favicon.ico",
    "src/assets"
],
"styles": [
    "./node_modules/@angular/material/prebuilt-themes/indigo-pink.css",
    "src/styles.css",
    "node_modules/bootstrap/dist/css/bootstrap.min.css"
],
"scripts": []
},
...

```

There are two `styles` sections in the `angular.json` file, and you must make sure to add the filename to the one closest to the top of the file. Save the changes to the file and run the command shown in Listing 9-5 to check that you have edited the correct `styles` section. If you don't see the new stylesheet in the output, then you have edited the wrong part of the file.

Add the classes shown in Listing 9-9 to the `div` element in the `app.component.html` file. These classes apply styles defined by the Bootstrap CSS framework.

Listing 9-9. Adding Classes in the `app.component.html` File in the `src/app` Folder

```
<div class="bg-primary text-white text-center">
    Hello, World
</div>
```

The development tools do not detect changes to the `angular.json` file, so stop them by typing `Control+C` and run the command shown in Listing 9-10 in the example folder to start them again.

Listing 9-10. Starting the Angular Development Tools

```
ng serve
```

A new `styles.js` bundle will be created during the initial startup. Reload the browser window if the browser doesn't reconnect to the development HTTP server, and you will see the effect of the new styles, as shown in Figure 9-5. (These styles were applied by the classes I added to the `div` element in Listing 9-9.)

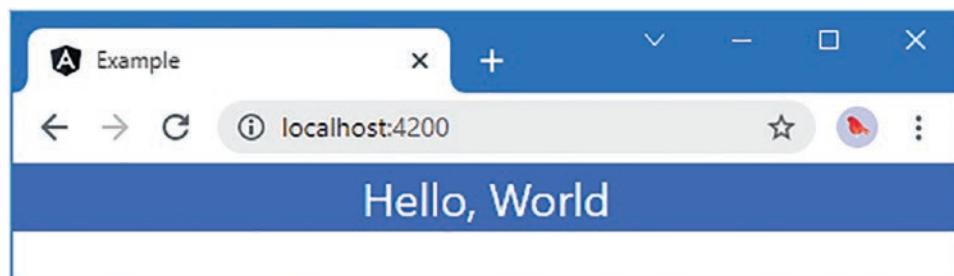


Figure 9-5. Adding a stylesheet

The original bundle contained just the `styles.css` file in the `src` folder, which is empty by default (but which was modified by the Angular Material installation), and the stylesheet from the Angular Material package. Now that the bundle contains the Bootstrap stylesheet, the bundle is larger, as shown by the build message:

```
...
styles.css, styles.js | styles          | 450.86 kB
...
```

This may seem like a large file just for some styles, but it is this size only during development, as I explain in the “Understanding the Production Build Process” section.

Using the Linter

A linter is a tool that inspects source code to ensure that it conforms to a set of coding conventions and rules. Run the command shown in Listing 9-11 in the `example` folder, which installs the popular ESLint linter package and uses the schematics API to configure the project.

Listing 9-11. Adding the Linter Package

```
ng add @angular-eslint/schematics@13.0.1
```

As part of the integration process, the linter package creates a configuration file named `.eslintrc.json` in the `example` folder. Two changes are required to configure the linter, as shown in Listing 9-12.

Listing 9-12. Configuring the Linter in the `.eslintrc.json` File in the `example` Folder

```
{
  "root": true,
  "ignorePatterns": [
    "projects/**/*",
    "src/test.ts"
  ],
  "overrides": [
    {
      "files": [
        "*.ts"
      ],
      "parserOptions": {
        "project": [
          "tsconfig.json"
        ],
        "createDefaultProgram": true
      },
      "extends": [
        "plugin:@angular-eslint/ng-cli-compat",
        "plugin:@angular-eslint/recommended",
      ]
    }
  ]
}
```

```

    "plugin:@angular-eslint/template/process-inline-templates"
],
"rules": {
  "@angular-eslint/directive-selector": [
    "error",
    {
      "type": "attribute",
      "prefix": "app",
      "style": "camelCase"
    }
  ],
  "@angular-eslint/component-selector": [
    "error",
    {
      "type": "element",
      "prefix": "app",
      "style": "kebab-case"
    }
  ]
},
{
  "files": [
    "*.html"
  ],
  "extends": [
    "plugin:@angular-eslint/template/recommended"
  ],
  "rules": {}
}
]
}

```

The first change excludes the `test.ts` file from linting. This file is created by the `ng new` command to support unit tests, and its contents will produce linting warnings. The second change expands the set of rules applied by the linter.

Additional JavaScript packages are required to support the expanded set of linting rules. Install these packages by running the command shown in Listing 9-13 in the example folder.

Listing 9-13. Installing Additional Packages

```
npm install eslint-plugin-import eslint-plugin-jsdoc eslint-plugin-prefer-arrow
```

To demonstrate how the linter works, I made two changes to a TypeScript file, as shown in Listing 9-14.

Listing 9-14. Making Changes in the app.component.ts File in the src/app Folder

```
import { Component } from '@angular/core';

debugger

@Component({
  selector: 'approot',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'example';
}
```

I added a debugger statement and changed the value of the selector property in the Component decorator. These changes illustrate the range of issues that can be detected by the linter. The debugger statement can cause problems when the application is deployed because it can halt code execution.

The change to the selector value breaks the style convention for Angular applications, where the selector should be specified in kebab-case, meaning that each word is separated by a hyphen. This is only a convention, however, and it doesn't prevent the application from working. (However, since I have changed only the app.component.ts file and not made a corresponding change in the HTML file, the application will build but not run as expected. I explain the relationship between the selector property and the HTML file in the next section of this chapter.)

Run the command shown in Listing 9-15 in the example folder to run the linter.

Listing 9-15. Running the Linter

```
ng lint
```

The linter inspects the files in the project and reports any problems that it encounters. The changes in Listing 9-14 result in the following messages:

...

Linting "example"...

```
C:\example\src\app\app.component.ts
 3:1  error  Unexpected 'debugger' statement no-debugger
 6:13 error  The selector should be kebab-case
            (https://angular.io/guide/styleguide#style-05-02)
            @angular-eslint/component-selector
```

```
2 problems (2 errors, 0 warnings)
Lint errors found in the listed files.
```

...

Linting isn't integrated into the regular build process and is performed manually. The most common use for linting is to check for potential problems before committing changes to a version control system, although some project teams make broader use of the linting facility by integrating it into other processes.

You may find that there are individual statements that cause the linter to report an error but that you are not able to change. Rather than disable the rule entirely, you can add a comment to the code that tells the linter to ignore the next line, like this:

```
...
// eslint-disable-next-line
...
```

If you have a file that is full of problems but you cannot make changes—often because there are constraints applied from some other part of the application—then you can disable linting for the entire file by adding this comment at the top of the page:

```
...
/* eslint-disable */
...
```

These comments allow you to ignore code that doesn't conform to the rules but that cannot be changed, while still linting the rest of the project.

You can also disable rules for the entire project in the `.eslintrc` file. Each rule has a name, which is included in the linter's error report. The name of the rule that checks for the `debugger` statement, for example, is `no-debugger`:

```
...
3:1  error  Unexpected 'debugger' statement no-debugger
...
```

The rules section of the `.eslintrc` file can be used to disable rules, as shown in Listing 9-16.

UNDERSTANDING THE LINTER RULES

The linter rules that are provided by the ESLint package are described at <https://eslint.org/docs/rules>, and each description includes examples of code that will pass and fail the rule. Rules whose names begin with `@angular` are defined by the `@angular-eslint` package and are described at <https://github.com/angular-eslint/angular-eslint/tree/master/packages/eslint-plugin/docs/rules>.

Listing 9-16. Disabling a Rule in the `.eslintrc` File in the example Folder

```
...
"rules": {
  "@angular-eslint/directive-selector": [
    "error",
    {
      "type": "attribute", "prefix": "app", "style": "camelCase"
    }
  ],
  "@angular-eslint/component-selector": [
    "error",
    {
      "type": "element", "prefix": "app", "style": "kebab-case"
    }
}
```

```
],
"no-debugger": "off"
}
...

```

Save the configuration file and run the `ng lint` command in the example folder. The new configuration disables the no-debugger rule, so the only warning is for the selector naming convention:

```
Linting "example"...
C:\example\src\app\app.component.ts
  6:13  error  The selector should be kebab-case (https://angular.io/guide/
styleguide#style-05-02)  @angular-eslint/component-selector
1 problem (1 error, 0 warnings)
Lint errors found in the listed files.
```

To address the remaining linter warning, I have changed the selector property back to its original value, as shown in Listing 9-17. I have also commented out the debugger statement, even though it will no longer be detected by the linter.

Listing 9-17. Changing a Property in the app.component.ts File in the src/app Folder

```
import { Component } from '@angular/core';

//debugger

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'example';
}
```

THE JOY AND MISERY OF LINTING

Linters can be a powerful tool for good, especially in a development team with mixed levels of skill and experience. Linters can detect common problems and subtle errors that lead to unexpected behavior or long-term maintenance issues. A good example is the difference between the JavaScript `==` and `===` operators, where a linter can warn when the wrong type of comparison has been performed. I like this kind of linting, and I like to run my code through the linting process after I have completed a major application feature or before I commit my code into version control.

But linters can also be a tool of division and strife. In addition to detecting coding errors, linters can be used to enforce rules about indentation, brace placement, the use of semicolons and spaces, and dozens of other style issues. Most developers have style preferences—I certainly do: I like four spaces for indentation, and I like opening braces to be on the same line and the expression they relate to. I know that some programmers have different preferences, just as I know those people are plain wrong and will one day see the light and start formatting their code correctly.

Linters allow people with strong views about formatting to enforce them on others, generally under the banner of being “opinionated,” which can tend toward “obnoxious.” The logic is that developers waste time arguing about different coding styles and everyone is better off being forced to write in the same way, which is typically the way preferred by the person with the strong views and ignores the fact that developers will just argue about something else because arguing is fun.

I especially dislike linting of formatting, which I see as divisive and unnecessary. I often help readers when they can’t get book examples working (my email address is adam@adam-freeman.com if you need help), and I see all sorts of coding style every week. But rather than forcing readers to code my way, I just get my code editor to reformat the code to the format that I prefer, which is a feature that every capable editor provides.

My advice is to use linting sparingly and focus on the issues that will cause real problems. Leave formatting decisions to the individuals and rely on code editor reformatting when you need to read code written by a team member who has different preferences.

Understanding How an Angular Application Works

Angular can seem like magic when you first start using it, and it is easy to become wary of making changes to the project files for fear of breaking something. Although there are lots of files in an Angular application, they all have a specific purpose, and they work together to do something far from magic: display HTML content to the user. In this section, I explain how the example Angular application works and how each part works toward the end result.

If you stopped the Angular development tools to run the linter in the previous section, run the command shown in Listing 9-18 in the example folder to start them again.

Listing 9-18. Starting the Angular Development Tools

```
ng serve
```

Once the initial build is complete, use a browser to request `http://localhost:4200`, and you will see the content shown in Figure 9-6.

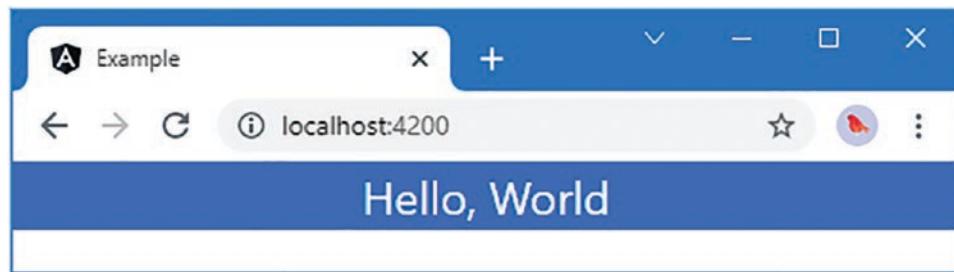


Figure 9-6. Running the example application

In the sections that follow, I explain how the files in the project are combined to produce the response shown in the figure.

Understanding the HTML Document

The starting point for running the application is the `index.html` file, which is found in the `src` folder. When the browser sent the request to the development HTTP server, it received this file, which contains the following elements:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Example</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
  <link rel="preconnect" href="https://fonts.gstatic.com">
  <link href="https://fonts.googleapis.com/css2?family=Roboto:wght@300;400;500&display=swap" rel="stylesheet">
  <link href="https://fonts.googleapis.com/icon?family=Material+Icons" rel="stylesheet">
</head>
<body>
  <app-root></app-root>
</body>
</html>
```

The header contains link elements for font files, which are required by the Angular Material package. The most important part of the file is the `app-root` element in the document body, whose purpose will become clear shortly.

The contents of the `index.html` file are modified as they are sent to the browser to include `script` elements for JavaScript files, like this:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Example</title>
  <base href="/">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" type="image/x-icon" href="favicon.ico">
  <link rel="preconnect" href="https://fonts.gstatic.com">
  <link href="https://fonts.googleapis.com/css2?
    family=Roboto:wght@300;400;500&display=swap" rel="stylesheet">
  <link href="https://fonts.googleapis.com/icon?family=Material+Icons" rel="stylesheet">
<link rel="stylesheet" href="styles.css"></head>
<body>
  <app-root></app-root>
  <script src="runtime.js" type="module"></script>
  <script src="polyfills.js" type="module"></script>
  <script src="styles.js" defer></script>
  <script src="vendor.js" type="module"></script>
  <script src="main.js" type="module"></script>
</body>
</html>
```

Understanding the Application Bootstrap

Browsers execute JavaScript files in the order in which their `script` elements appear, starting with the `runtime.js` file, which contains the code that processes the contents of the other JavaScript files.

Next comes the `polyfills.js` file, which contains code that provides implementations of features that the browser doesn't support, and then the `styles.js` file, which contains the CSS styles the application needs. The `vendor.js` file contains the third-party code the application requires, including the Angular framework. This file can be large during development because it contains all the Angular features, even if they are not required by the application. An optimization process is used to prepare an application for deployment, as described later in this chapter.

The final file is the `main.js` bundle, which contains the custom application code. The name of the bundle is taken from the entry point for the application, which is the `main.ts` file in the `src` folder. Once the other bundle files have been processed, the statements in the `main.ts` file are executed to initialize Angular and run the application. Here is the content of the `main.ts` file as it is created by the `ng new` command:

```
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';
import { environment } from './environments/environment';

if (environment.production) {
  enableProdMode();
}

platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.error(err));
```

The `import` statements declare dependencies on other JavaScript modules, providing access to Angular features (the dependencies on `@angular` modules, which are included in the `vendor.js` file) and the custom code in the application (the `AppModule` dependency). The final import is for environment settings, which are used to create different configuration settings for development, test, and production platforms, such as this code:

```
...
if (environment.production) {
  enableProdMode();
}
...
```

Angular has a production mode that disables some useful checks that are performed during development and that are described in later chapters. Production mode is enabled by calling the `enableProdMode` function, which is imported from the `@angular/core` module.

To work out whether production mode should be enabled, a check is performed to see whether `environment.production` is true. This check corresponds to the contents of the `environment.prod.ts` file in the `src/environments` folder, which sets this value and is applied when the application is built in preparation for deployment. The result is that production mode will be enabled if the application has been built for production but disabled the rest of the time.

The remaining statement in the `main.ts` file is responsible for starting the application.

```
...
platformBrowserDynamic().bootstrapModule(AppModule).catch(err => console.error(err));
...
```

The `platformBrowserDynamic` function initializes the Angular platform for use in a web browser and is imported from the `@angular/platform-browser-dynamic` module. Angular has been designed to run in a range of different environments, and calling the `platformBrowserDynamic` function is the first step in starting an application in a browser.

The next step is to call the `bootstrapModule` method, which accepts the Angular root module for the application, which is `AppModule` by default; `AppModule` is imported from the `app.module.ts` file in the `src/app` folder and described in the next section. The `bootstrapModule` method provides Angular with the entry point into the application and represents the bridge between the functionality provided by the `@angular` modules and the custom code and content in the project. The final part of this statement uses the `catch` keyword to handle any bootstrapping errors by writing them to the browser's JavaScript console.

Understanding the Root Angular Module

The term *module* does double duty in an Angular application and refers to both a JavaScript module and an Angular module. JavaScript modules are used to track dependencies in the application and ensure that the browser receives only the code it requires. Angular modules are used to configure a part of the Angular application.

Every application has a *root* Angular module, which is responsible for describing the application to Angular. For applications created with the `ng new` command, the root module is called `AppModule`, and it is defined in the `app.module.ts` file in the `src/app` folder, which contains the following code:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    BrowserAnimationsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

The `AppModule` class doesn't define any members, but it provides Angular with essential information through the configuration properties of its `@NgModule` decorator. I describe the different properties that are used to configure an Angular module in later chapters, but the one that is of interest now is the

`bootstrap` property, which tells Angular that it should load a component called `AppComponent` as part of the application startup process. Components are the main building block in Angular applications, and the content provided by the component called `AppComponent` will be displayed to the user.

Understanding the Angular Component

The component called `AppComponent`, which is selected by the root Angular module, is defined in the `app.component.ts` file in the `src/app` folder. Here are the contents of the `app.component.ts` file, which I edited earlier in the chapter to demonstrate linting:

```
import { Component } from '@angular/core';

//debugger

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'example';
}
```

The properties for the `@Component` decorator configure its behavior. The `selector` property tells Angular that this component will be used to replace an HTML element called `app-root`. The `templateUrl` and `styleUrls` properties tell Angular that the HTML content that the component wants to present to the user can be found in a file called `app.component.html` and that the CSS styles to apply to the HTML content are defined in a file called `app.component.css` (although the CSS file is empty in new projects).

Here is the content of the `app.component.html` file, which I edited earlier in the chapter to demonstrate hot reloading and the use of CSS styles:

```
<div class="bg-primary text-white text-center">
  Hello, World
</div>
```

This file contains regular HTML elements, but, as you will learn, Angular features are applied by using custom HTML elements or by adding attributes to regular HTML elements.

Understanding Content Display

When the application starts, Angular processes the `index.html` file, locates the element that matches the root component's `selector` property, and replaces it with the contents of the files specified by the root component's `templateUrl` and `styleUrls` properties. This is done using the Domain Object Model (DOM) API provided by the browser for JavaScript applications, and the changes can be seen only by right-clicking in the browser window and selecting `Inspect` from the pop-up menu, producing the following result:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
```

```

<title>Example</title>
<base href="/">
<meta name="viewport" content="width=device-width, initial-scale=1">
<link rel="icon" type="image/x-icon" href="favicon.ico">
<link rel="preconnect" href="https://fonts.gstatic.com">
<link href="https://fonts.googleapis.com/css2?
    family=Roboto:wght@300;400;500&display=swap" rel="stylesheet">
<link href="https://fonts.googleapis.com/icon?family=Material+Icons"
    rel="stylesheet">
<link rel="stylesheet" href="styles.css">
<style>
/*# sourceMappingURL=data:application/json;
base64,eyJ2ZXJzaW9uIjozLCJzb3VyY2VzIjpXSwibmFtZXMiOltdLCJtY
XBwaW5ncyI6IiIsImZpbGUiOiJhcHAuY29tcG9uZW50LmNzcyJ9 */
</style>
</head>
<body>
<app-root _ngcontent-ogl-c11="" ng-version="13.0.3">
    <div _ngcontent-ogl-c11="" class="bg-primary text-white text-center">
        Hello, World
    </div>
</app-root>
<script src="runtime.js" type="module"></script>
<script src="polyfills.js" type="module"></script>
<script src="styles.js" defer=""></script>
<script src="vendor.js" type="module"></script>
<script src="main.js" type="module"></script>
</body>

```

The `app-root` element contains the `div` element from the component's template, and the attributes are added by Angular during the initialization process.

The `style` elements represent the contents of the `styles.css` file in the `app` folder, the `app.component.css` file in the `src/app` folder, and the Angular Material and Bootstrap stylesheets added using the `angular.json` file.

The combination of the dynamically generated `div` element and the `class` attributes I specified in the template produces the result shown in Figure 9-7.

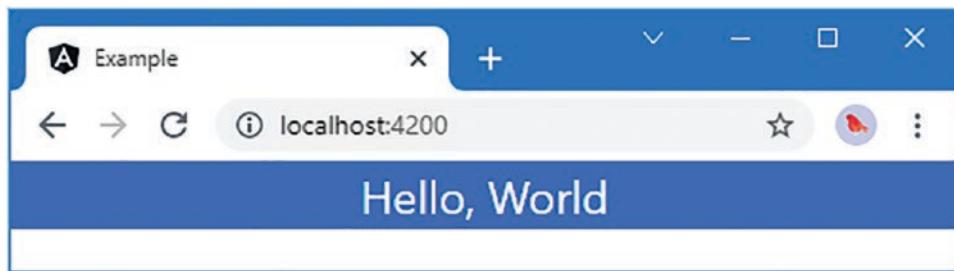


Figure 9-7. Displaying a component's content

Understanding the Production Build Process

During development, the emphasis is on fast compilation so that the results can be displayed as quickly as possible in the browser, leading to a good iterative development process. During development with the `ng serve` command, the compilers and the bundler don't apply any optimizations, which is why the bundle files are so large. The size doesn't matter because the browser is running on the same machine as the server and will load immediately.

Before an application is deployed, it is built using an optimizing process. To run this type of build, run the command shown in Listing 9-19 in the example folder.

Listing 9-19. Performing the Production Build

```
ng build
```

The `ng build` command performs the production compilation process, and the bundles it produces are smaller and contain only the code that is required by the application.

Note The `angular.json` command defines default build modes for commands, and the default configuration uses development mode for the `ng serve` command and production mode for the `ng build` command. You can edit the configuration file to change these settings or override them on the command line with the `--configuration` argument.

You can see the details of the bundles that are produced in the messages generated by the compiler.

```
...
Initial Chunk Files      | Names          | Size
styles.2bbd6476e9a18d40.css | styles         | 230.36 kB
main.dcffd5dba104918c.js  | main          | 171.02 kB
polyfills.221f478e3706b78e.js | polyfills     | 36.19 kB
runtime.a2e8a93eb7a8cc89.js | runtime        | 1.04 kB
                                | Initial Total | 438.62 kB
...

```

Features such as hot reloading are not added to the bundles, and the large `vendor.js` bundle is no longer produced. Instead, the `main.js` bundle contains the application and just the parts of third-party code it relies on.

UNDERSTANDING AHEAD-OF-TIME COMPILATION

The development build process leaves the decorators, which describe the building blocks of an Angular application, in the output. These are then transformed into API calls by the Angular runtime in the browser, which is known as *just-in-time* (JIT) compilation. The production build process enables a feature named *ahead-of-time* (AOT) compilation, which transforms the decorators so that it doesn't have to be done every time the application runs.

Combined with the other build optimizations, the result is an Angular application that loads faster and starts up faster. The drawback is that the additional compilation requires time, which can be frustrating if you enable optimizing builds during development.

Running the Production Build

To test the production build, run the command shown in Listing 9-20 in the example folder.

Listing 9-20. Running the Production Build

```
npx http-server@14.0.0 dist/example --port 5000
```

This command will download and execute version 14.0.0 of the `http-server` package, which provides a simple, self-contained HTTP server. The command tells the `http-server` package to serve the contents of the `dist/example` folder and listen for requests on port 5000. Open a new web browser and request `http://localhost:5000`; you will see the production version of the example application, as shown in Figure 9-8 (although, unless you examine the HTTP requests sent by the browser to get the bundle files, you won't see any differences from the development version shown in earlier figures).

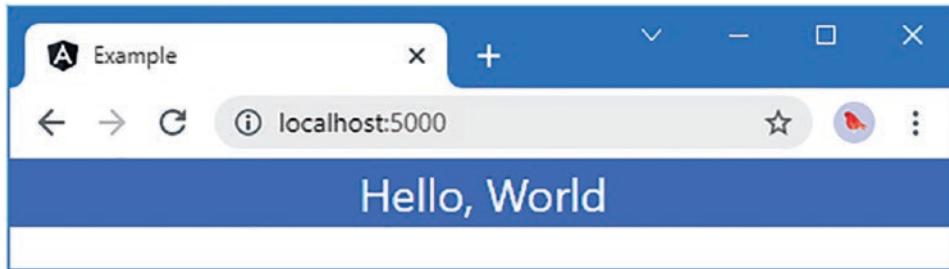


Figure 9-8. Running the production build

Once you have tested the production build, stop the HTTP server using `Control+C`.

Starting Development in an Angular Project

You have seen how the initial building blocks of an Angular application fit together and how the bootstrap process results in content being displayed to the user. In this section, I add a simple data model to the project, which is the typical starting point for most developers, and add features to the component, beyond the static content added earlier in the chapter.

Creating the Data Model

Of all the building blocks in an application, the data model is the one for which Angular is the least prescriptive. Elsewhere in the application, Angular requires specific decorators to be applied or parts of the API to be used, but the only requirement for the model is that it provides access to the data that the application requires; the details of how this is done and what that data looks like is left to the developer.

This can feel a little odd, and it can be difficult to know how to begin, but, at its heart, the model can be broken into three parts.

- One or more classes that describe the data in the model
- A data source that loads and saves data, typically to a server
- A repository that allows the data in the model to be manipulated

In the following sections, I create a simple model, which provides the functionality that I need to describe Angular features in the chapters that follow.

Creating the Descriptive Model Class

Descriptive classes, as the name suggests, describe the data in the application. In a real project, there will usually be a lot of classes to fully describe the data that the application operates on. To get started for this chapter, I am going to create a single, simple class as the foundation for the data model. I added a file named `product.model.ts` to the `src/app` folder with the code shown in Listing 9-21.

The name of the file follows the Angular descriptive naming convention. The `product` and `model` parts of the name tell you that this is the part of the data model that relates to products, and the `.ts` extension denotes a TypeScript file. You don't have to follow this convention, but Angular projects usually contain a lot of files, and cryptic names make it difficult to navigate around the source code.

Listing 9-21. The Contents of the `product.model.ts` File in the `src/app` Folder

```
export class Product {

    constructor(public id?: number,
        public name?: string,
        public category?: string,
        public price?: number) { }

}
```

The `Product` class defines properties for a product identifier, the name of the product, its category, and the price. The properties are defined as optional constructor arguments, which is a useful approach if you are creating objects using an HTML form, which I demonstrate in Chapter 12.

Creating the Data Source

The data source provides the application with the data. The most common type of data source uses HTTP to request data from a web service, which I describe in Chapter 23. For this chapter, I need something simpler that I can reset to a known state each time the application is started to ensure that you get the expected results from the examples. I added a file called `datasource.model.ts` to the `src/app` folder with the code shown in Listing 9-22.

Listing 9-22. The Contents of the `datasource.model.ts` File in the `src/app` Folder

```
import { Product } from "./product.model";

export class SimpleDataSource {
    private data: Product[];

    constructor() {
        this.data = new Array<Product>(
            new Product(1, "Kayak", "Watersports", 275),
            new Product(2, "Lifejacket", "Watersports", 48.95),
            new Product(3, "Soccer Ball", "Soccer", 19.50),
            new Product(4, "Corner Flags", "Soccer", 34.95),
            new Product(5, "Thinking Cap", "Chess", 16));
    }
}
```

```

    getData(): Product[] {
        return this.data;
    }
}

```

The data in this class is hardwired, which means that any changes that are made in the application will be lost when the browser is reloaded. This is far from useful in a real application, but it is ideal for book examples.

Creating the Model Repository

The final step to complete the simple model is to define a repository that will provide access to the data from the data source and allow it to be manipulated in the application. I added a file called `repository.model.ts` in the `src/app` folder and used it to define the class shown in Listing 9-23.

Listing 9-23. The Contents of the `repository.model.ts` File in the `src/app` Folder

```

import { Product } from "./product.model";
import { SimpleDataSource } from "./datasource.model";

export class Model {
    private dataSource: SimpleDataSource;
    private products: Product[];
    private locator = (p: Product, id: number | any) => p.id == id;

    constructor() {
        this.dataSource = new SimpleDataSource();
        this.products = new Array<Product>();
        this.dataSource.getData().forEach(p => this.products.push(p));
    }

    getProducts(): Product[] {
        return this.products;
    }

    getProduct(id: number): Product | undefined {
        return this.products.find(p => this.locator(p, id));
    }

    saveProduct(product: Product) {
        if (product.id == 0 || product.id == null) {
            product.id = this.generateID();
            this.products.push(product);
        } else {
            let index = this.products.findIndex(p => this.locator(p, product.id));
            this.products.splice(index, 1, product);
        }
    }
}

```

```

deleteProduct(id: number) {
    let index = this.products.findIndex(p => this.locator(p, id));
    if (index > -1) {
        this.products.splice(index, 1);
    }
}

private generateID(): number {
    let candidate = 100;
    while (this.getProduct(candidate) != null) {
        candidate++;
    }
    return candidate;
}
}

```

The Model class defines a constructor that gets the initial data from the data source class and provides access to it through a set of methods. These methods are typical of those defined by a repository and are described in Table 9-8.

Table 9-8. The Types of Web Forms Code Nuggets

Name	Description
getProducts	This method returns an array containing all the Product objects in the model.
getProduct	This method returns a single Product object based on its ID.
saveProduct	This method updates an existing Product object or adds a new one to the model.
deleteProduct	This method removes a Product object from the model based on its ID.

The implementation of the repository may seem odd because the data objects are stored in a standard JavaScript array, but the methods defined by the Model class present the data as though it were a collection of Product objects indexed by the `id` property. There are two main considerations when writing a repository for model data. The first is that it should present the data that will be displayed as efficiently as possible. For the example application, this means presenting all the data in the model in a form that can be iterated, such as an array. This is important because the iteration can happen often, as I explain in later chapters. The other operations of the Model class are inefficient, but they will be used less often.

The second consideration is being able to present unchanged data for Angular to work with. I explain why this is important in Chapter 11, but in terms of implementing the repository, it means that the `getProducts` method should return the same object when it is called multiple times unless one of the other methods or another part of the application has made a change to the data that the `getProducts` method provides. If a method returns a different object each time it is returned, even if they are different arrays containing the same objects, then Angular will report an error. Taking both points into account means that the best way to implement the repository is to store the data in an array and accept the inefficiencies.

Creating a Component and Template

Templates contain the HTML content that a component wants to present to the user. Templates can range from a single HTML element to a complex block of content.

To create a template, I added a file called `template.html` to the `src/app` folder and added the HTML elements shown in Listing 9-24.

Listing 9-24. The Contents of the template.html File in the src/app Folder

```
<div class="bg-info text-white p-2">
    There are {{model.getProducts().length}} products in the model
</div>
```

Most of this template is standard HTML, but the part between the double brace characters (the {{ and }} in the div element) is an example of a data binding. When the template is displayed, Angular will process its content, discover the binding, and evaluate the expression that it contains to produce the content that will be displayed by the data binding.

The logic and data required to support the template are provided by its component, which is a TypeScript class to which the @Component decorator has been applied. To provide a component for the template, I added a file called component.ts to the src/app folder and defined the class shown in Listing 9-25.

Listing 9-25. The Contents of the component.ts File in the src/app Folder

```
import { Component } from "@angular/core";
import { Model } from "./repository.model";

@Component({
    selector: "app",
    templateUrl: "template.html"
})
export class ProductComponent {
    model: Model = new Model();
}
```

The @Component decorator configures the component. The selector property specifies the HTML element that the directive will be applied to, which is app. The templateUrl property in the @Component directive specifies the content that will be used as the contents of the app element, and, for this example, this property specifies the template.html file.

The component class, which is ProductComponent for this example, is responsible for providing the template with the data and logic needed for its bindings. The ProductComponent class defines a single property, called model, which provides access to a Model object.

The app element I used for the component's selector isn't the same element that the ng new command uses when it creates a project and that is expected in the index.html file. In Listing 9-26, I have modified the index.html file to introduce an app element to match the component's selector.

Listing 9-26. Changing the Custom Element in the index.html File in the app Folder

```
<!doctype html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>Example</title>
    <base href="/">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <link rel="icon" type="image/x-icon" href="favicon.ico">
    <link rel="preconnect" href="https://fonts.gstatic.com">
    <link href="https://fonts.googleapis.com/css2?
        family=Roboto:wght@300;400;500&display=swap" rel="stylesheet">
```

```

<link href="https://fonts.googleapis.com/icon?family=Material+Icons"
      rel="stylesheet">
</head>
<body>
  <app></app>
</body>
</html>

```

This isn't something you need to do in a real project, but it further demonstrates that Angular applications fit together in simple and predictable ways and that you can change any part.

Configuring the Root Angular Module

The component that I created in the previous section won't be part of the application until I register it with the root Angular module. In Listing 9-27, I have used the `import` keyword to import the component, and I have used the `@NgModule` configuration properties to register the component.

Listing 9-27. Registering a Component in the `app.module.ts` File in the `src/app` Folder

```

import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

import { ProductComponent } from './component';

@NgModule({
  declarations: [ProductComponent],
  imports: [
    BrowserModule,
    BrowserAnimationsModule
  ],
  providers: [],
  bootstrap: [ProductComponent]
})
export class AppModule { }

```

I used the name `ProductComponent` in the `import` statement, and I added this name to the `declarations` array, which configures the set of components and other features in the application. I also changed the value of the `bootstrap` property so that the new component is the one that is used when the application starts.

Run the command shown in Listing 9-28 in the example folder to start the Angular development tools.

Listing 9-28. Starting the Angular Development Tools

```
ng serve
```

Once the initial build process is complete, use a web browser to request `http://localhost:4200`, which will produce the response shown in Figure 9-9.

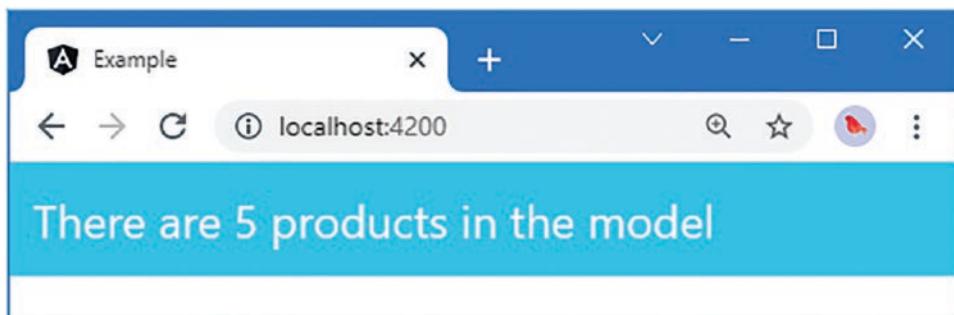


Figure 9-9. The effect of a new component and template

The standard Angular bootstrap sequence is performed, but the custom component and template that I created in the previous section are used, rather than the ones set up when the project was created.

Summary

In this chapter, I created an Angular project and used it to introduce the tools that it contains and explain how a simple Angular application works. In the next chapter, I start digging into the Angular features, starting with data bindings.

CHAPTER 10



Using Data Bindings

The example application in the previous chapter contains a simple template that was displayed to the user and that contained a data binding that showed how many objects were in the data model. In this chapter, I describe the basic data bindings that Angular provides and demonstrate how they can be used to produce dynamic content. In later chapters, I describe more advanced data bindings and explain how to extend the Angular binding system with custom features. Table 10-1 puts data bindings in context.

Table 10-1. Putting Data Bindings in Context

Question	Answer
What are they?	Data bindings are expressions embedded into templates and are evaluated to produce dynamic content in the HTML document.
Why are they useful?	Data bindings provide the link between the HTML elements in the HTML document and in template files with the data and code in the application.
How are they used?	Data bindings are applied as attributes on HTML elements or as special sequences of characters in strings.
Are there any pitfalls or limitations?	Data bindings contain simple JavaScript expressions that are evaluated to generate content. The main pitfall is including too much logic in a binding because such logic cannot be properly tested or used elsewhere in the application. Data binding expressions should be as simple as possible and rely on components (and other Angular features such as pipes) to provide complex application logic.
Are there any alternatives?	No. Data bindings are an essential part of Angular development.

Table 10-2 summarizes the chapter.

Table 10-2. Chapter Summary

Problem	Solution	Listing
Displaying data dynamically in the HTML document	Define a data binding	1–4
Configuring an HTML element	Use a standard property or attribute binding	5, 8
Setting the contents of an element	Use a string interpolation binding	6, 7
Configuring the classes to which an element is assigned	Use a class binding	9–13
Configuring the individual styles applied to an element	Use a style binding	14–17
Manually triggering a data model update	Use the browser's JavaScript console	18, 19

Preparing for This Chapter

For this chapter, I continue using the example project from Chapter 9. To prepare for this chapter, I added a method to the component class, as shown in Listing 10-1.

Tip You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/Apress/pro-angular-5ed>. See Chapter 1 for how to get help if you have problems running the examples.

Listing 10-1. Adding a Method in the component.ts File in the src/app Folder

```
import { Component } from "@angular/core";
import { Model } from "./repository.model";

@Component({
  selector: "app",
  templateUrl: "template.html"
})
export class ProductComponent {
  model: Model = new Model();

  getClasses(): string {
    return this.model.getProducts().length == 5 ? "bg-success" : "bg-warning";
  }
}
```

Run the following command in the example folder to start the Angular development tools:

```
ng serve
```

Open a new browser and navigate to `http://localhost:4200` to see the content, shown in Figure 10-1, that will be displayed.

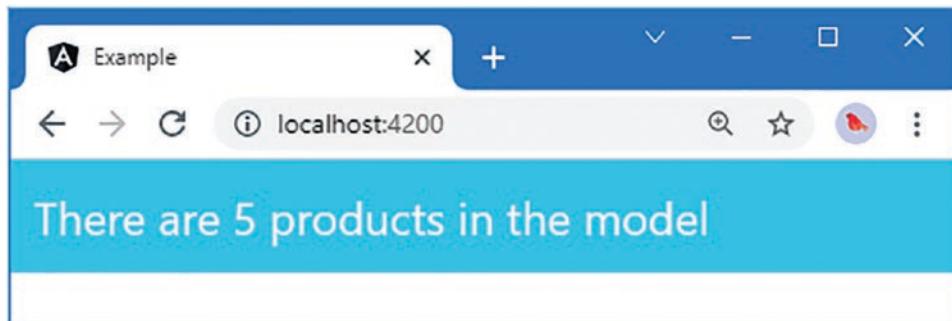


Figure 10-1. Running the example application

Understanding One-Way Data Bindings

One-way data bindings are used to generate content for the user and are the basic feature used in Angular templates. The term *one-way* refers to the fact that the data flows in one direction, meaning that data flows *from* the component *to* the data binding so that it can be displayed in a template.

Tip There are other types of Angular data binding, which I describe in later chapters. *Event bindings* flow in the other direction, from the elements in the template into the rest of the application, and they allow user interaction. *Two-way bindings* allow data to flow in both directions and are most often used in forms. See Chapters 11 and 12 for details of other bindings.

To get started with one-way data bindings, I have replaced the content of the template, as shown in Listing 10-2.

Listing 10-2. The Contents of the template.html File in the src/app Folder

```
<div [ngClass]="getClasses()">
    Hello, World.
</div>
```

When you save the changes to the template, the development tools will rebuild the application and trigger a browser reload, displaying the output shown in Figure 10-2.

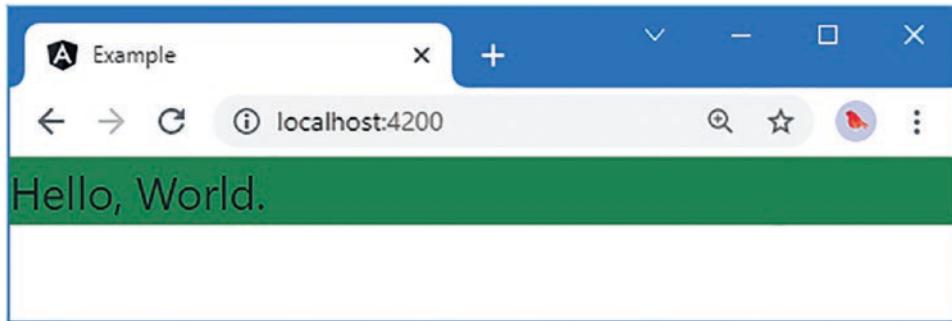


Figure 10-2. Using a one-way data binding

This is a simple example, but it shows the basic structure of a data binding, which is illustrated in Figure 10-3.

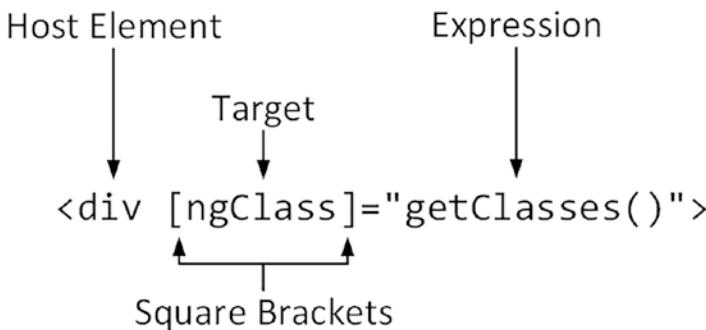


Figure 10-3. The anatomy of a data binding

A data binding has these four parts:

- The *host element* is the HTML element that the binding will affect, by changing its appearance, content, or behavior.
- The *square brackets* tell Angular that this is a one-way data binding. When Angular sees square brackets in a data binding, it will evaluate the expression and pass the result to the binding's *target* so that it can modify the host element.
- The *target* specifies what the binding will do. There are two different types of target: a *directive* or a *property binding*.
- The *expression* is a fragment of JavaScript that is evaluated using the template's component to provide context, meaning that the component's property and methods can be included in the expression, like the `getClasses` method in the example binding.

Looking at the binding in Listing 10-2, you can see that the host element is a `div` element, meaning that's the element that the binding is intended to modify. The expression invokes the component's `getClasses` method, which was defined at the start of the chapter. This method returns a string containing a Bootstrap CSS class based on the number of objects in the data model.

```
...
getClasses(): string {
  return this.model.getProducts().length == 5 ? "bg-success" : "bg-warning";
}
...
```

If there are five objects in the data model, then the method returns `bg-success`, which is a Bootstrap class that applies a green background. Otherwise, the method returns `bg-warning`, which is a Bootstrap class that applies an amber background.

The target for the data binding is a *directive*, which is a class that is specifically written to support a data binding. Angular comes with some useful built-in directives, and you can create your own to provide custom functionality. The names of the built-in directives start with `ng`, which tells you that the `ngClass` target is one of the built-in directives. The target usually gives an indication of what the directive does, and as its name suggests, the `ngClass` directive will add or remove the host element from the class or classes whose names are returned when the expression is evaluated.

Putting it all together, the data binding will add the `div` element to the `bg-success` or `bg-warning` classes based on the number of items in the data model.

Since there are five objects in the model when the application starts (because the initial data is hard-coded into the `SimpleDataSource` class created in Chapter 9), the `getClasses` method returns `bg-success` and produces the result shown in Figure 10-3, adding a green background to the `div` element.

Understanding the Binding Target

When Angular processes the target of a data binding, it starts by checking to see whether it matches a directive. Most applications will rely on a mix of the built-in directives provided by Angular and custom directives that provide application-specific features. You can usually tell when a directive is the target of a data binding because the name will be distinctive and give some indication of what the directive is for. The built-in directives can be recognized by the `ng` prefix. The binding in Listing 10-2 gives you a hint that the target is a built-in directive that is related to the class membership of the host element. For quick reference, Table 10-3 describes the basic built-in Angular directives and where they are described in this book. (There are other directives described in later chapters, but these are the simplest and the ones you will use most often.)

Table 10-3. The Basic Built-in Angular Directives

Name	Description
ngClass	This directive is used to assign host elements to classes, as described in the “Setting Classes and Styles” section.
ngStyle	This directive is used to set individual styles, as described in the “Setting Classes and Styles” section.
ngIf	This directive is used to insert content in the HTML document when its expression evaluates as <code>true</code> , as described in Chapter 11.
ngFor	This directive inserts the same content into the HTML document for each item in a data source, as described in Chapter 11.
ngSwitch ngSwitchCase ngSwitchDefault	These directives are used to choose between blocks of content to insert into the HTML document based on the value of the expression, as described in Chapter 11.
ngTemplateOutlet	This directive is used to repeat a block of content, as described in Chapter 11.

Understanding Property Bindings

If the binding target doesn’t correspond to a directive, then Angular checks to see whether the target can be used to create a property binding. There are five different types of property binding, which are listed in Table 10-4, along with the details of where they are described in detail.

Table 10-4. The Angular Property Bindings

Name	Description
[property]	This is the standard property binding, which is used to set a property on the JavaScript object that represents the host element in the Document Object Model (DOM), as described in the “Using the Standard Property and Attribute Bindings” section.
[attr.name]	This is the attribute binding, which is used to set the value of attributes on the host HTML element for which there are no DOM properties, as described in the “Using the Attribute Binding” section.
[class.name]	This is the special class property binding, which is used to configure class membership of the host element, as described in the “Using the Class Bindings” section.
[style.name]	This is the special style property binding, which is used to configure style settings of the host element, as described in the “Using the Style Bindings” section.

Understanding the Expression

The expression in a data binding is a fragment of JavaScript code that is evaluated to provide a value for the target. The expression has access to the properties and methods defined by the component, which is how the binding in Listing 10-2 can invoke the `getClasses` method to provide the `ngClass` directive with the name of the class that the host element should be added to.

Expressions are not restricted to calling methods or reading properties from the component; they can also perform most standard JavaScript operations. As an example, Listing 10-3 shows an expression that has a literal string value being concatenated with the result of the `getClasses` method.

Listing 10-3. Performing an Operation in the template.html File in the src/app Folder

```
<div [ngClass]="'text-white p-2 ' + getClasses()" >
  Hello, World.
</div>
```

The expression is enclosed in double quotes, which means that the string literal has to be defined using single quotes. The JavaScript concatenation operator is the `+` character, and the result from the expression will be the combination of both strings, like this:

```
text-white p-2 bg-success
```

The effect is that the `ngClass` directive will add the host element to four classes: `text-white`, `m-2`, and `p-2`, which Bootstrap uses to set the text color and add margin and padding around an element's content; and `bg-success`, which sets the background color. Figure 10-4 shows the combination of these classes.

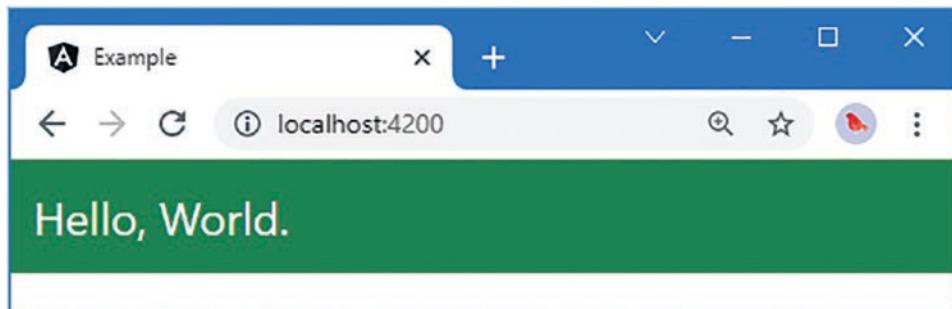


Figure 10-4. Combining classes in a JavaScript expression

It is easy to get carried away when writing expressions and include complex logic in the template. This can cause problems because the expressions are not checked by the TypeScript compiler nor can they be easily unit tested, which means that bugs are more likely to remain undetected until the application has been deployed. To avoid this issue, expressions should be as simple as possible and, ideally, used only to retrieve data from the component and format it for display. All the complex retrieval and processing logic should be defined in the component or the model, where it can be compiled and tested.

Understanding the Brackets

The square brackets (the `[` and `]` characters) tell Angular that this is a one-way data binding that has an expression that should be evaluated. Angular will still process the binding if you omit the brackets and the target is a directive, but the expression won't be evaluated, and the content between the quote characters will be passed to the directive as a literal value. Listing 10-4 adds an element to the template with a binding that doesn't have square brackets.

Listing 10-4. Omitting the Brackets in a Data Binding in the template.html File in the src/app Folder

```
<div [ngClass]="'text-white p-2 ' + getClasses()>
  Hello, World.
</div>
<div ngClass="'text-white p-2 ' + getClasses()>
  Hello, World.
</div>
```

If you examine the HTML element in the browser's DOM viewer (by right-clicking in the browser window and selecting Inspect or Inspect Element from the pop-up menu), you will see that its `class` attribute has been set to the literal string, like this:

```
class="'text-white p-2 ' + getClasses()"
```

The browser will try to process the classes to which the host element has been assigned, but the element's appearance won't be as expected since the classes won't correspond to the names used by Bootstrap. This is a common mistake to make, so it is the first thing to check whether a binding doesn't have the effect you expected.

The square brackets are not the only ones that Angular uses in data bindings. For quick reference, Table 10-5 provides the complete set of brackets, the meaning of each, and where they are described in detail.

Table 10-5. The Angular Brackets

Name	Description
[target]="expr"	The square brackets indicate a one-way data binding where data flows from the expression to the target. The different forms of this type of binding are the topic of this chapter.
{{expression}}	This is the string interpolation binding, which is described in the “Using the String Interpolation Binding” section.
(target) ="expr"	The round brackets indicate a one-way binding where the data flows from the target to the destination specified by the expression. This is the binding used to handle events, as described in Chapter 12.
[(target)] ="expr"	This combination of brackets—known as the <i>banana-in-a-box</i> —indicates a two-way binding, where data flows in both directions between the target and the destination specified by the expression, as described in Chapter 12.

Understanding the Host Element

The host element is the simplest part of a data binding. Data bindings can be applied to any HTML element in a template, and an element can have multiple bindings, each of which can manage a different aspect of the element's appearance or behavior. You will see elements with multiple bindings in later examples.

Using the Standard Property and Attribute Bindings

If the target of a binding doesn't match a directive, Angular will try to apply a property binding. The sections that follow describe the most common property bindings: the standard property binding and the attribute binding.

Using the Standard Property Binding

The browser uses the Document Object Model to represent the HTML document. Each element in the HTML document, including the host element, is represented using a JavaScript object in the DOM. Like all JavaScript objects, the ones used to represent HTML elements have properties. These properties are used to manage the state of the element so that the value property, for example, is used to set the contents of an input element. When the browser parses an HTML document, it encounters each new HTML element, creates an object in the DOM to represent it, and uses the element's attributes to set the initial values for the object's properties.

The standard property binding lets you set the value of a property for the object that represents the host element, using the result of an expression. For example, setting the target of a binding to value will set the content of an input element, as shown in Listing 10-5.

Listing 10-5. Using the Standard Property Binding in the template.html File in the src/app Folder

```
<div [ngClass]="'text-white p-2 ' + getClasses()>
  Hello, World.
</div>
<div class="form-group m-2">
  <label>Name:</label>
  <input class="form-control" [value]="model.getProduct(1)?.name ?? 'None'" />
</div>
```

The new binding in this example specifies that the value property should be bound to the result of an expression that calls a method on the data model to retrieve a data object from the repository by specifying a key. It is possible that there is no data object with that key, in which case the repository method will return null.

To guard against using null for the host element's value property, the binding uses the null conditional operator (the ? character) to safely navigate the result returned by the method, like this:

```
...
<input class="form-control" [value]="model.getProduct(1)?.name ?? 'None'" />
...
```

If the result from the getProduct method isn't null, then the expression will read the value of the name property and use it as the result. But if the result from the method is null, then the name property won't be read, and the nullish coalescing operator (the ?? characters) will set the result to None instead.

GETTING TO KNOW THE HTML ELEMENT PROPERTIES

Using property bindings can require some work figuring out which property you need to set because there are inconsistencies in the HTML specification. The name of most properties matches the name of the attribute that sets their initial value so that if you are used to setting the value attribute on an input element, for example, then you can achieve the same effect by setting the value property. But some property names don't match their attribute names, and some properties are not configured by attributes at all.

The Mozilla Foundation provides a useful reference for all the objects that are used to represent HTML elements in the DOM at <https://developer.mozilla.org/en-US/docs/Web/API>. For each element, Mozilla provides a summary of the properties that are available and what each is used for. Start with HTMLElement (<https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement>), which provides the functionality common to all elements. You can then branch out into the objects that are for specific elements, such as HTMLInputElement, which is used to represent input elements.

When you save the changes to the template, the browser will reload and display an `input` element whose content is the `name` property of the data object with the key of `1` in the model repository, as shown in Figure 10-5.

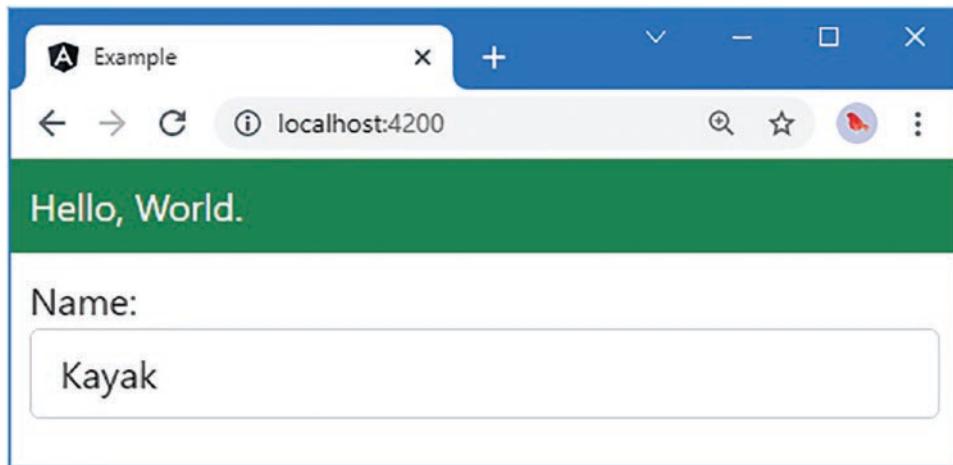


Figure 10-5. Using the standard property binding

Using the String Interpolation Binding

Angular provides a special version of the standard property binding, known as the *string interpolation binding*, that is used to include expression results in the text content of host elements. To understand why this special binding is useful, it helps to think about how the content of an element is set using the standard property binding. The `textContent` property is used to set the content of HTML elements, which means that the content of an element can be set using a data binding like the one shown in Listing 10-6.

Listing 10-6. Setting an Element's Content in the template.html File in the src/app Folder

```
<div [ngClass]="'text-white p-2 ' + getClasses()"
      [textContent]="'Name: ' + (model.getProduct(1)?.name ?? 'None')">
</div>
<div class="form-group m-2">
  <label>Name:</label>
  <input class="form-control" [value]="model.getProduct(1)?.name ?? 'None'" />
</div>
```

The expression in the new binding concatenates a literal string with the results of a method call to set the content of the `div` element.

The expression in this example is awkward to write, requiring careful attention to quotes, spaces, and brackets to ensure that the expected result is displayed in the output. The problem becomes worse for more complex bindings, where multiple dynamic values are interspersed among blocks of static content.

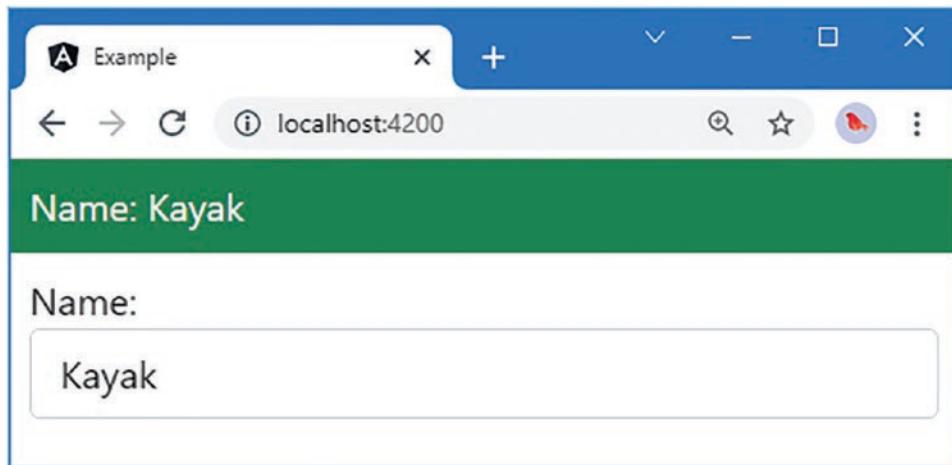
The string interpolation binding simplified this process by allowing fragments of expressions to be defined within the content of an element, as shown in Listing 10-7.

Listing 10-7. Using the String Interpolation Binding in the template.html File in the src/app Folder

```
<div [ngClass]="'text-white p-2 ' + getClasses()">
  Name: {{ model.getProduct(1)?.name ?? 'None' }}
</div>
<div class="form-group m-2">
  <label>Name:</label>
  <input class="form-control" [value]="model.getProduct(1)?.name ?? 'None'" />
</div>
```

The string interpolation binding is denoted using pairs of curly brackets ({{ and }}). A single element can contain multiple string interpolation bindings.

Angular combines the content of the HTML element with the contents of the brackets to create a binding for the `textContent` property. The result is the same as Listing 10-6, which is shown in Figure 10-6, but the process of writing the binding is simpler and less error-prone.

**Figure 10-6.** Using the string interpolation binding

Using the Attribute Binding

There are some oddities in the HTML and DOM specifications that mean that not all HTML element attributes have equivalent properties in the DOM API. For these situations, Angular provides the *attribute binding*, which is used to set an attribute on the host element rather than setting the value of the JavaScript object that represents it in the DOM.

The most often used attribute without a corresponding property is `colspan`, which is used to set the number of columns that a `td` element will occupy in a table. Listing 10-8 shows using the attribute binding to set the `colspan` element based on the number of objects in the data model.

Listing 10-8. Using an Attribute Binding in the template.html File in the src/app Folder

```
<div [ngClass]="'text-white p-2 ' + getClasses()">
  Name: {{ model.getProduct(1)?.name ?? 'None' }}
</div>
```

```

<div class="form-group m-2">
  <label>Name:</label>
  <input class="form-control" [value]="model.getProduct(1)?.name ?? 'None'" />
</div>
<table class="table mt-2">
  <tr>
    <th>1</th><th>2</th><th>3</th><th>4</th><th>5</th>
  </tr>
  <tr>
    <td [attr.colspan]="model.getProducts().length">
      {{model.getProduct(1)?.name ?? 'None'}}
    </td>
  </tr>
</table>

```

The attribute binding is applied by defining a target that prefixes the name of the attribute with attr. (the term attr, followed by a period). In the listing, I have used the attribute binding to set the value of the colspan element on one of the td elements in the table, like this:

```

...
<td [attr.colspan]="model.getProducts().length">
...

```

Angular will evaluate the expression and set the value of the colspan attribute to the result. Since the data model is hardwired to start with five data objects, the effect is that the colspan attribute creates a table cell that spans five columns, as shown in Figure 10-7.

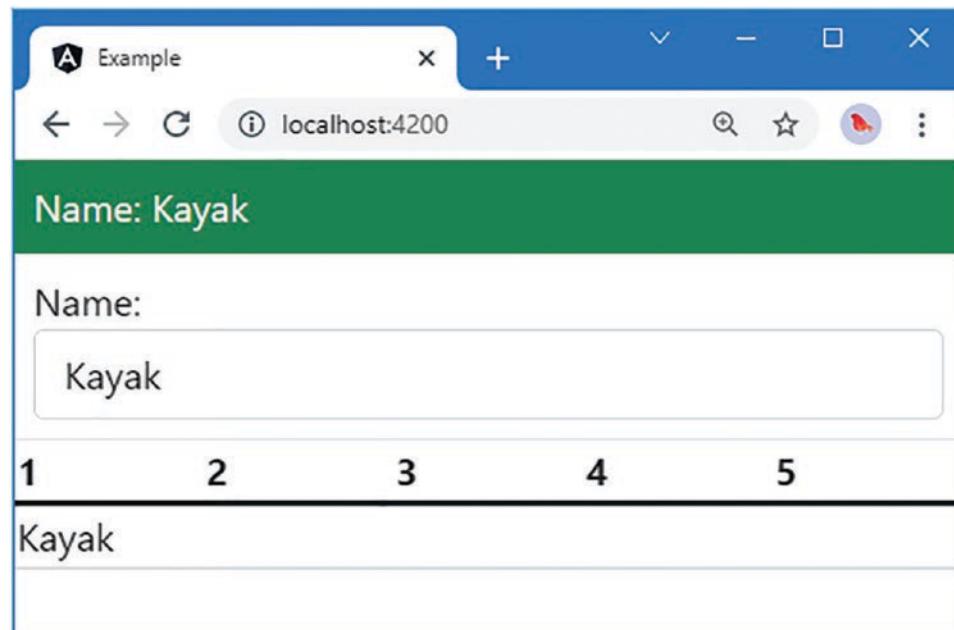


Figure 10-7. Using an attribute binding

Setting Classes and Styles

Angular provides special support in property bindings for assigning the host element to classes and for configuring individual style properties. I describe these bindings in the sections that follow, along with details of the `ngClass` and `ngStyle` directives, which provide closely related features.

Using the Class Bindings

There are three different ways in which you can use data bindings to manage the class memberships of an element: the standard property binding, the special class binding, and the `ngClass` directive. All three are described in Table 10-6, and each works in a slightly different way and is useful in different circumstances, as described in the sections that follow.

Table 10-6. The Angular Class Bindings

Example	Description
<code><div [class]="expr"></div></code>	This binding evaluates the expression and uses the result to replace any existing class memberships.
<code><div [class.myClass]="expr"></div></code>	This binding evaluates the expression and uses the result to set the element's membership of <code>myClass</code> .
<code><div [ngClass]="map"></div></code>	This binding sets class membership of multiple classes using the data in a map object.

Setting All of an Element's Classes with the Standard Binding

The standard property binding can be used to set all of an element's classes in a single step, which is useful when you have a method or property in the component that returns all of the classes to which an element should belong in a single string, with the names separated by spaces. Listing 10-9 shows the revision of the `getClasses` method in the component that returns a different string of class names based on the `price` property of a `Product` object.

Listing 10-9. Providing All Classes in a Single String in the component.ts File in the src/app Folder

```
import { Component } from "@angular/core";
import { Model } from "./repository.model";

@Component({
  selector: "app",
  templateUrl: "template.html"
})
export class ProductComponent {
  model: Model = new Model();

  getClasses(key: number): string {
    let product = this.model.getProduct(key);
    return "p-2 " + ((product?.price ?? 0) < 50 ? "bg-info" : "bg-warning");
  }
}
```

The result from the `getClasses` method will include the `p-2` class, which adds padding around the host element's content, for all `Product` objects. If the value of the `price` property is less than 50, the `bg-info` class will be included in the result, and if the value is 50 or more, the `bg-warning` class will be included (these classes set different background colors). You must ensure that the names of the classes are separated by spaces.

Listing 10-10 replaces the contents of the `template.html` file to show the standard property binding being used to set the `class` property of host elements using the component's `getClasses` method.

Listing 10-10. Setting Class Memberships in the template.html File in the src/app Folder

```
<div class="text-white">
  <div [class]="getClasses(1)">
    The first product is {{model.getProduct(1)?.name}}.
  </div>
  <div [class]="getClasses(2)">
    The second product is {{model.getProduct(2)?.name}}
  </div>
</div>
```

When the standard property binding is used to set the `class` property, the result of the expression replaces any previous classes that an element belonged to, which means that it can be used only when the binding expression returns all the classes that are required, as in this example, producing the result shown in Figure 10-8.

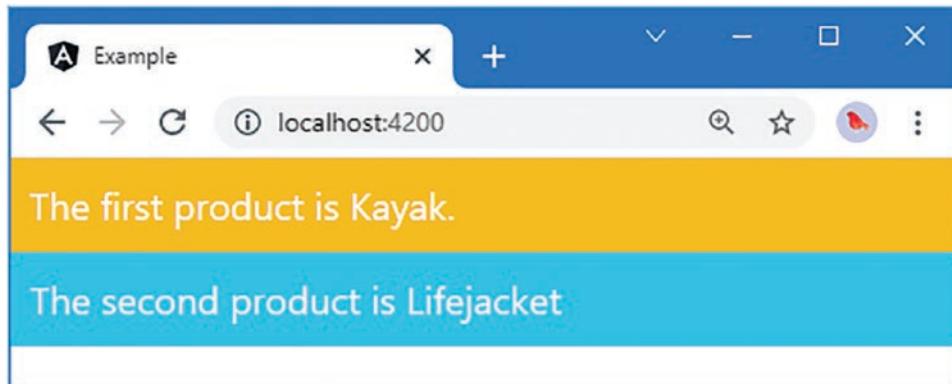


Figure 10-8. Setting class memberships

Setting Individual Classes Using the Special Class Binding

The special class binding provides finer-grained control than the standard property binding and allows membership of a single class to be managed using an expression. This is useful if you want to build on the existing class memberships of an element, rather than replace them entirely. Listing 10-11 shows the use of the special class binding.

Listing 10-11. Using the Special Class Binding in the template.html File in the src/app Folder

```
<div class="text-white">
  <div [class]="getClasses(1)">
    The first product is {{model.getProduct(1)?.name}}.
  </div>
  <div class="p-2"
    [class.bg-success]="(model.getProduct(2)?.price ?? 0) < 50"
    [class.bg-info]="(model.getProduct(2)?.price ?? 0) >= 50">
    The second product is {{model.getProduct(2)?.name}}
  </div>
</div>
```

The special class binding is specified with a target that combines the term `class`, followed by a period, followed by the name of the class whose membership is being managed. In the listing, there are two special class bindings, which manage the membership of the `bg-success` and `bg-info` classes.

The special class binding will add the host element to the specified class if the result of the expression is *truthy* (as described in the “Understanding Truthy and Falsy” sidebar). In this case, the host element will be a member of the `bg-success` class if the `price` property is less than 50 and a member of the `bg-info` class if the `price` property is 50 or more.

These bindings act independently from one another and do not interfere with any existing classes that an element belongs to, such as the `p-2` class, which Bootstrap uses to add padding around an element’s content.

UNDERSTANDING TRUTHY AND FALSY

As explained in Chapter 3, JavaScript has an odd feature, where the result of an expression can be *truthy* or *falsy*, providing a pitfall for the unwary. The following results are always *falsy*:

- The `false` (`boolean`) value
- The `0` (`number`) value
- The empty string (`" "`)
- `null`
- `undefined`
- `Nan` (a special number value)

All other values are *truthy*, which can be confusing. For example, “`false`” (a string whose content is the word `false`) is *truthy*. The best way to avoid confusion is to only use expressions that evaluate to the Boolean values `true` and `false`.

Setting Classes Using the `ngClass` Directive

The `ngClass` directive is a more flexible alternative to the standard and special property bindings and behaves differently based on the type of data that is returned by the expression, as described in Table 10-7.

Table 10-7. The Expression Result Types Supported by the ngClass Directive

Name	Description
String	The host element is added to the classes specified by the string. Multiple classes are separated by spaces.
Array	Each object in the array is the name of a class that the host element will be added to.
Object	Each property on the object is the name of one or more classes, separated by spaces. The host element will be added to the class if the value of the property is truthy.

The string and array features are useful, but it is the ability to use an object (known as a *map*) to create complex class membership policies that make the ngClass directive especially useful. Listing 10-12 shows the addition of a component method that returns a map object.

UNDERSTANDING THE NULLISH OPERATOR PRECEDENCE PITFALL

Care must be taken when using the nullish operator when it is combined with other JavaScript operations, especially when the results are combined to form strings. Here is an example of a problem statement:

```
...
return "p-2 " + (product?.price ?? 0 < 50 ? "bg-info" : "bg-warning");
...
```

The problem arises because the nullish operator has a lower precedence than the less than operator. Here is the same statement with the addition of parentheses that show how the statement is evaluated:

```
...
return "p-2 " + ((product?.price ?? (0 < 50)) ? "bg-info" : "bg-warning");
...
```

The effect is that the less than operator is applied only when the product.price property is null, and, even then, it is used only to determine if 0 is less than 50. Since JavaScript comparisons work on truthiness, the outcome from the ternary operator is always true: the product.price property will be truthy when it is not null, and the $0 < 50$ expression is truthy when the product.price property is null. The effect is that the statement always returns the string "p-2 bg-info".

The solution is to use parentheses to group related terms together and avoid relying on JavaScript operator precedence, like this:

```
...
return "p-2 " + ((product?.price ?? 0) < 50 ? "bg-info" : "bg-warning");
...
```

This ensures that the expression evaluated by the ternary operator behaves as intended.

Listing 10-12. Returning a Class Map Object in the component.ts File in the src/app Folder

```
import { Component } from "@angular/core";
import { Model } from "./repository.model";
```

```

@Component({
  selector: "app",
  templateUrl: "template.html"
})
export class ProductComponent {
  model: Model = new Model();

  getClasses(key: number): string {
    let product = this.model.getProduct(key);
    return "p-2 " + ((product?.price ?? 0) < 50 ? "bg-info" : "bg-warning");
  }

  getClassMap(key: number): Object {
    let product = this.model.getProduct(key);
    return {
      "text-center bg-danger": product?.name === "Kayak",
      "bg-info": (product?.price ?? 0) < 50
    };
  }
}

```

The getClassMap method returns an object with properties whose values are one or more class names, with values based on the property values of the Product object whose key is specified as the method argument. As an example, when the key is 1, the method returns this object:

```

...
{
  "text-center bg-danger":true,
  "bg-info":false
}
...

```

The first property will assign the host element to the text-center class (which Bootstrap uses to center the text horizontally) and the bg-danger class (which sets the element's background color). The second property evaluates to false, which means that the host element will not be added to the bg-info class. It may seem odd to specify a property that doesn't result in an element being added to a class, but, as you will see shortly, the value of expressions is automatically updated to reflect changes in the application, and being able to define a map object that specifies memberships this way can be useful.

Listing 10-13 shows the getClassMap and the map objects it returns used as the expression for data bindings that target the ngClass directive.

Listing 10-13. Using the ngClass Directive in the template.html File in the src/app Folder

```

<div class="text-white">
  <div class="p-2" [ngClass]="getClassMap(1)">
    The first product is {{model.getProduct(1)?.name}}.
  </div>
  <div class="p-2" [ngClass]="getClassMap(2)">
    The second product is {{model.getProduct(2)?.name}}.
  </div>

```

```
<div class="p-2" [ngClass]="{'bg-success': (model.getProduct(3)?.price ?? 0) < 50,
  'bg-info': (model.getProduct(3)?.price ?? 0) >= 50}">
  The third product is {{model.getProduct(3)?.name}}
</div>
</div>
```

The first two div elements have bindings that use the getClassMap method. The third div element shows an alternative approach, which is to define the map in the template. For this element, membership of the bg-info and bg-warning classes is tied to the value of the price property of a Product object, as shown in Figure 10-9. Care should be taken with this technique because the expression contains JavaScript logic that cannot be readily tested.

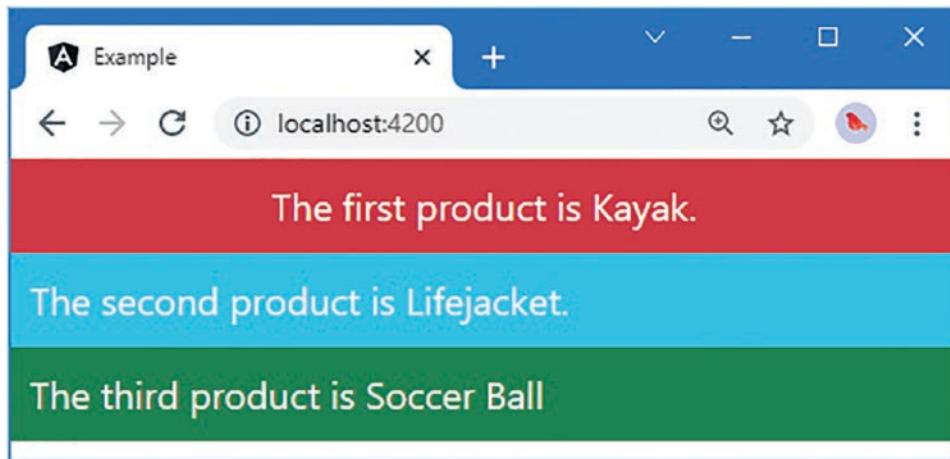


Figure 10-9. Using the ngClass directive

Using the Style Bindings

There are three different ways in which you can use data bindings to set style properties of the host element: the standard property binding, the special style binding, and the ngStyle directive. All three are described in Table 10-8 and demonstrated in the sections that follow.

Table 10-8. The Angular Style Bindings

Example	Description
<div [style.myStyle]="expr"></div>	This is the standard property binding, which is used to set a single style property to the result of the expression.
<div [style.myStyle.units]="expr"></div>	This is the special style binding, which allows the units for the style value to be specified as part of the target.
<div [ngStyle]="map"></div>	This binding sets multiple style properties using the data in a map object.

Setting a Single Style Property

The standard property binding and the special style bindings are used to set the value of a single style property. The difference between these bindings is that the standard property binding must include the units required for the style, while the special binding allows for the units to be included in the binding target. To demonstrate the difference, Listing 10-14 adds two new properties to the component.

Listing 10-14. Adding Properties in the component.ts File in the src/app Folder

```
import { Component } from "@angular/core";
import { Model } from "./repository.model";

@Component({
  selector: "app",
  templateUrl: "template.html"
})
export class ProductComponent {
  model: Model = new Model();

  getClasses(key: number): string {
    let product = this.model.getProduct(key);
    return "p-2 " + ((product?.price ?? 0) < 50 ? "bg-info" : "bg-warning");
  }

  getClassMap(key: number): Object {
    let product = this.model.getProduct(key);
    return {
      "text-center bg-danger": product?.name == "Kayak",
      "bg-info": (product?.price ?? 0) < 50
    };
  }
}

fontSizeWithUnits: string = "30px";
fontSizeWithoutUnits: string= "30";
}
```

The `fontSizeWithUnits` property returns a value that includes a quantity and the units that quantity is expressed in: 30 pixels. The `fontSizeWithoutUnits` property returns just the quantity, without any unit information. Listing 10-15 replaces the contents of the `template.html` file to show how these properties can be used with the standard and special bindings.

Caution Do not try to use the standard property binding to target the `style` property to set multiple style values. The object returned by the `style` property of the JavaScript object that represents the host element in the DOM is read-only. Some browsers will ignore this and allow changes to be made, but the results are unpredictable and cannot be relied on. If you want to set multiple style properties, then create a binding for each of them or use the `ngStyle` directive.

Listing 10-15. Using Style Bindings in the template.html File in the src/app Folder

```
<div class="text-white">
  <div class="p-2 bg-warning">
    The <span [style.fontSize]="fontSizeWithUnits">first</span>
    product is {{model.getProduct(1)?.name}}.
  </div>
  <div class="p-2 bg-info">
    The <span [style.fontSize.px]="fontSizeWithoutUnits">second</span>
    product is {{model.getProduct(2)?.name}}.
  </div>
</div>
```

The target for the binding is `style.fontSize`, which sets the size of the font used for the host element's content. The expression for this binding uses the `fontSizeWithUnits` property, whose value includes the units, `px` for pixels, required to set the font size.

The target for the special binding is `style.fontSize.px`, which tells Angular that the value of the expression specifies the number in pixels. This allows the binding to use the component's `fontSizeWithoutUnits` property, which doesn't include units.

Tip You can specify style properties using the JavaScript property name format (`[style.fontSize]`) or using the CSS property name format (`[style.font-size]`).

The result of both bindings is the same, which is to set the font size of the span elements to 30 pixels, producing the result shown in Figure 10-10.

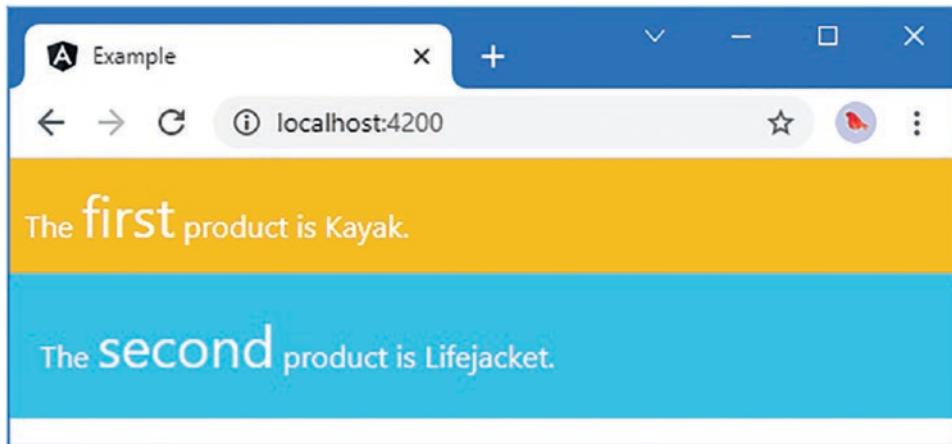


Figure 10-10. Setting individual style properties

Setting Styles Using the `ngStyle` Directive

The `ngStyle` directive allows multiple style properties to be set using a map object, similar to the way that the `ngClass` directive works. Listing 10-16 shows the addition of a component method that returns a map containing style settings.

Listing 10-16. Creating a Style Map Object in the component.ts File in the src/app Folder

```
import { Component } from "@angular/core";
import { Model } from "./repository.model";

@Component({
  selector: "app",
  templateUrl: "template.html"
})
export class ProductComponent {
  model: Model = new Model();

  getClasses(key: number): string {
    let product = this.model.getProduct(key);
    return "p-2 " + ((product?.price ?? 0) < 50 ? "bg-info" : "bg-warning");
  }

  // getClassMap(key: number): Object {
  //   let product = this.model.getProduct(key);
  //   return {
  //     "text-center bg-danger": product?.name == "Kayak",
  //     "bg-info": (product?.price ?? 0) < 50
  //   };
  // }

  // fontSizeWithUnits: string = "30px";
  // fontSizeWithoutUnits: string= "30";

  getStyles(key: number) {
    let product = this.model.getProduct(key);
    return {
      fontSize: "30px",
      "margin.px": 100,
      color: (product?.price?? 0) > 50 ? "red" : "green"
    };
  }
}
```

The map object returned by the `getStyle` method shows that the `ngStyle` directive is able to support both of the formats that can be used with property bindings, including either the units in the value or the property name. Here is the map object that the `getStyles` method produces when the value of the `key` argument is 1:

```
...
{
  "fontSize":"30px",
  "margin.px":100,
  "color":"red"
}
...
```

Listing 10-17 shows data bindings in the template that use the `ngStyle` directive and whose expressions call the `getStyles` method.

Listing 10-17. Using the `ngStyle` Directive in the template.html File in the src/app Folder

```
<div class="text-white">
  <div class="p-2 bg-warning">
    The <span [ngStyle]="getStyles(1)">first</span>
    product is {{model.getProduct(1)?.name}}.
  </div>
  <div class="p-2 bg-info">
    The <span [ngStyle]="getStyles(2)">second</span>
    product is {{model.getProduct(2)?.name}}.
  </div>
</div>
```

The result is that each `span` element receives a tailored set of styles, based on the argument passed to the `getStyles` method, as shown in Figure 10-11.

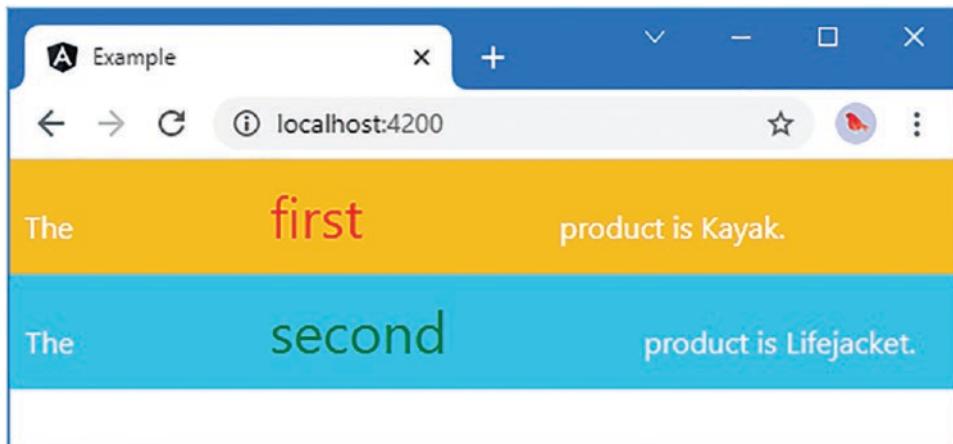


Figure 10-11. Using the `ngStyle` directive

Updating the Data in the Application

When you start out with Angular, it can seem like a lot of effort to deal with the data bindings, remembering which binding is required in different situations. You might be wondering if it is worth the effort.

Bindings are worth understanding because their expressions are re-evaluated when the data they depend on changes. As an example, if you are using a string interpolation binding to display the value of a property, then the binding will automatically update when the value of the property is changed.

To provide a demonstration, I am going to jump ahead and show you how to take manual control of the updating process. This is not a technique that is required in normal Angular development, but it provides a solid demonstration of why bindings are so important. Listing 10-18 shows some changes to the component that enables the demonstration.

Listing 10-18. Preparing the Component in the component.ts File in the src/app Folder

```
import { ApplicationRef, Component } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";

@Component({
  selector: "app",
  templateUrl: "template.html"
})
export class ProductComponent {
  model: Model = new Model();

  constructor(ref: ApplicationRef) {
    (<any>window).appRef = ref;
    (<any>window).model = this.model;
  }

  getProductByPosition(position: number): Product {
    return this.model.getProducts()[position];
  }

  getClassesByPosition(position: number): string {
    let product = this.getProductByPosition(position);
    return "p-2 " + ((product?.price ?? 0) < 50 ? "bg-info" : "bg-warning");
  }

  // getClasses(key: number): string {
  //   let product = this.model.getProduct(key);
  //   return "p-2 " + ((product?.price ?? 0) < 50 ? "bg-info" : "bg-warning");
  // }

  // getStyles(key: number) {
  //   let product = this.model.getProduct(key);
  //   return {
  //     fontSize: "30px",
  //     "margin.px": 100,
  //     color: (product?.price?? 0) > 50 ? "red" : "green"
  //   };
  // }
}
```

I have imported the `ApplicationRef` type from the `@angular/core` module. When Angular performs the bootstrapping process, it creates an `ApplicationRef` object to represent the application. Listing 10-18 adds a constructor to the component that receives an `ApplicationRef` object as an argument, using the Angular dependency injection feature, which I describe in Chapter 17. Without going into detail now, declaring a constructor argument like this tells Angular that the component wants to receive the `ApplicationRef` object when a new instance is created.

Within the constructor, there are two statements that make a demonstration possible but would undermine many of the benefits of using TypeScript and Angular if used in a real project.

```
...
(<any>window).appRef = ref;
(<any>window).model = this.model;
...
```

These statements define variables in the global namespace and assign the ApplicationRef and Model objects to them. It is good practice to keep the global namespace as clear as possible, but exposing these objects allows them to be manipulated through the browser's JavaScript console, which is important for this example.

The other methods added to the constructor allow a Product object to be retrieved from the repository based on its position, rather than by its key, and to generate a class map that differs based on the value of the price property.

Listing 10-19 shows the corresponding changes to the template, which uses the ngClass directive to set class memberships and the string interpolation binding to display the value of the Product.name property.

Listing 10-19. Preparing for Changes in the template.html File in the src/app Folder

```
<div class="text-white">
  <div [ngClass]="getClassesByPosition(0)">
    The first product is {{getProductByPosition(0).name}}.
  </div>
  <div [ngClass]="getClassesByPosition(1)">
    The second product is {{getProductByPosition(1).name}}
  </div>
</div>
```

Save the changes to the component and template. Once the browser has reloaded the page, enter the following statement into the browser's JavaScript console and press Return:

```
model.products.shift()
```

This statement calls the shift method on the array of Product objects in the model, which removes the first item from the array and returns it. You won't see any changes yet because Angular doesn't know that the model has been modified. To tell Angular to check for changes, enter the following statement into the browser's JavaScript console and press Return:

```
appRef.tick()
```

The tick method starts the Angular change detection process, where Angular looks at the data in the application and the expressions in the data binding and processes any changes. The data bindings in the template use specific array indexes to display data, and now that an object has been removed from the model, the bindings will be updated to display new values, as shown in Figure 10-12.

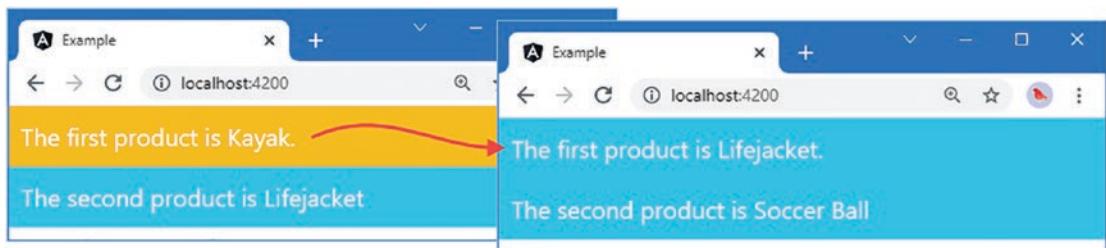


Figure 10-12. Manually updating the application model

It is worth taking a moment to think about what happened when the change detection process ran. Angular re-evaluated the expressions on the bindings in the template and updated their values. In turn, the `ngClass` directive and the string interpolation binding reconfigured their host elements by changing their class memberships and displaying new content.

This happens because Angular data bindings are *live*, meaning that the relationship between the expression, the target, and the host element continues to exist after the initial content is displayed to the user and dynamically reflects changes to the application state. This effect is, I admit, much more impressive when you don't have to make changes using the JavaScript console. I explain how Angular allows the user to trigger changes using events and forms in Chapter 12.

Summary

In this chapter, I described the structure of Angular data bindings and showed you how they are used to create relationships between the data in the application and the HTML elements that are displayed to the user. I introduced the property bindings and described how two of the built-in directives—`ngClass` and `ngStyle`—are used. In the next chapter, I explain how more of the built-in directives work.

CHAPTER 11



Using the Built-in Directives

In this chapter, I describe the built-in directives that are responsible for some of the most commonly required functionality for creating web applications: selectively including content, choosing between different fragments of content, and repeating content. I also describe some limitations that Angular puts on the expressions that are used for one-way data bindings and the directives that provide them. Table 11-1 puts the built-in template directives in context.

Table 11-1. Putting the Built-in Directives in Context

Question	Answer
What are they?	The built-in directives described in this chapter are responsible for selectively including content, selecting between fragments of content, and repeating content for each item in an array. There are also directives for setting an element's styles and class memberships, as described in Chapter 11.
Why are they useful?	The tasks that can be performed with these directives are the most common and fundamental in web application development, and they provide the foundation for adapting the content shown to the user based on the data in the application.
How are they used?	The directives are applied to HTML elements in templates. There are examples throughout this chapter (and in the rest of the book).
Are there any pitfalls or limitations?	The syntax for using the built-in template directives requires you to remember that some of them (including <code>ngIf</code> and <code>ngFor</code>) must be prefixed with an asterisk, while others (including <code>ngClass</code> , <code>ngStyle</code> , and <code>ngSwitch</code>) must be enclosed in square brackets. I explain why this is required in the “Understanding Micro-Template Directives” sidebar, but it is easy to forget and get an unexpected result.
Are there any alternatives?	You could write your own custom directives—a process that I described in Chapters 13 and 14—but the built-in directives are well-written and comprehensively tested. For most applications, using the built-in directives is preferable, unless they cannot provide exactly the functionality that is required.

Table 11-2 summarizes the chapter.

Table 11-2. Chapter Summary

Problem	Solution	Listing
Conditionally displaying content based on a data binding expression	Use the <code>ngIf</code> directive	1-3
Choosing between different content based on the value of a data binding expression	Use the <code>ngSwitch</code> directive	4, 5
Generating a section of content for each object produced by a data binding expression	Use the <code>ngFor</code> directive	6-12
Repeating a block of content	Use the <code>ngTemplateOutlet</code> directive	13-14
Apply a directive without using an HTML element	Use the <code>ng-container</code> element	15
Preventing template errors	Avoid modifying the application state as a side effect of a data binding expression	16-20
Avoiding context errors	Ensure that data binding expressions use only the properties and methods provided by the template's component	21-23

Preparing the Example Project

This chapter relies on the example project that was created in Chapter 9 and modified in Chapter 10. To prepare for the topic of this chapter, Listing 11-1 shows changes to the component class that remove features that are no longer required and adds new methods and a property.

Tip You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/Apress/pro-angular-5ed>. See Chapter 1 for how to get help if you have problems running the examples.

Listing 11-1. Changes in the component.ts File in the src/app Folder

```
import { ApplicationRef, Component } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";

@Component({
  selector: "app",
  templateUrl: "template.html"
})
export class ProductComponent {
  model: Model = new Model();
```

```

constructor(ref: ApplicationRef) {
    (<any>window).appRef = ref;
    (<any>window).model = this.model;
}

getProductByPosition(position: number): Product {
    return this.model.getProducts()[position];
}

// getClassesByPosition(position: number): string {
//     let product = this.getProductByPosition(position);
//     return "p-2 " + ((product?.price ?? 0) < 50 ? "bg-info" : "bg-warning");
// }

getProduct(key: number): Product | undefined {
    return this.model.getProduct(key);
}

getProducts(): Product[] {
    return this.model.getProducts();
}

getProductCount(): number {
    return this.getProducts().length;
}

targetName: string = "Kayak";
}

```

Listing 11-2 shows the contents of the template file, which displays the number of products in the data model by calling the component's new getProductCount method.

Listing 11-2. The Contents of the template.html File in the src/app Folder

```

<div class="text-white">
    <div class="bg-info p-2">
        There are {{getProductCount()}} products.
    </div>
</div>

```

Run the following command from the command line in the example folder to start the TypeScript compiler and the development HTTP server:

```
ng serve
```

Open a new browser window and navigate to `http://localhost:4200` to see the content shown in Figure 11-1.

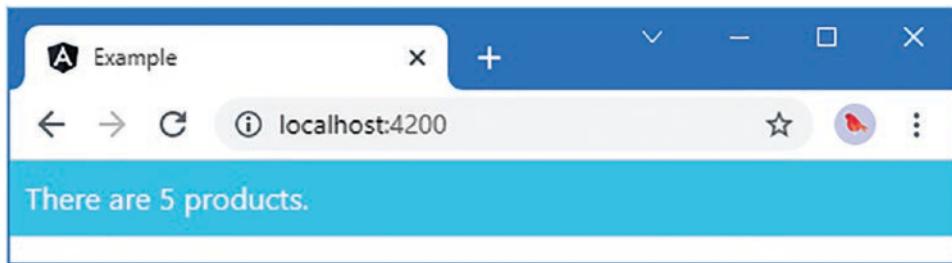


Figure 11-1. Running the example application

Using the Built-in Directives

Angular comes with a set of built-in directives that provide features commonly required in web applications. Table 11-3 describes the directives that are available, which I demonstrate in the sections that follow (except for the `ngClass` and `ngStyle` directives, which are covered in Chapter 10).

Table 11-3. The Built-in Directives

Example	Description
<code><div *ngIf="expr"></div></code>	The <code>ngIf</code> directive is used to include an element and its content in the HTML document if the expression evaluates as true. The asterisk before the directive name indicates that this is a micro-template directive, as described in the “Understanding Micro-Template Directives” sidebar.
<code><div [ngSwitch]="expr"> ... </div></code>	The <code>ngSwitch</code> directive is used to choose between multiple elements to include in the HTML document based on the result of an expression, which is then compared to the result of the individual expressions defined using <code>ngSwitchCase</code> directives. If none of the <code>ngSwitchCase</code> values matches, then the element to which the <code>ngSwitchDefault</code> directive has been applied will be used. The asterisks before the <code>ngSwitchCase</code> and <code>ngSwitchDefault</code> directives indicate they are micro-template directives, as described in the “Understanding Micro-Template Directives” sidebar.
<code><div *ngFor="#item of expr"> </div></code>	The <code>ngFor</code> directive is used to generate the same set of elements for each object in an array. The asterisk before the directive name indicates that this is a micro-template directive, as described in the “Understanding Micro-Template Directives” sidebar.
<code><div ngClass="expr"></div></code>	The <code>ngClass</code> directive is used to manage class membership, as described in Chapter 10.

(continued)

Table 11-3. (continued)

Example	Description
<div ngStyle="expr"></div>	The ngStyle directive is used to manage styles applied directly to elements (as opposed to applying styles through classes), as described in Chapter 10.
<ng-template [ngTemplateOutlet]="myTemp1"> </ngtemplate>	The ngTemplateOutlet directive is used to repeat a block of content in a template.

Using the ngIf Directive

ngIf is the simplest of the built-in directives and is used to include a fragment of HTML in the document when an expression evaluates as true, as shown in Listing 11-3.

Listing 11-3. Using the ngIf Directive in the template.html File in the src/app Folder

```
<div class="text-white">
  <div class="bg-info p-2">
    There are {{getProductCount()}} products.
  </div>

  <div *ngIf="getProductCount() > 4" class="bg-info p-2 mt-1">
    There are more than 4 products in the model
  </div>

  <div *ngIf="getProductByPosition(0).name != 'Kayak'" class="bg-info p-2 mt-1">
    The first product isn't a Kayak
  </div>
</div>
```

The ngIf directive has been applied to two div elements, with expressions that check the number of Product objects in the model and whether the name of the first Product is Kayak.

The first expression evaluates as true, which means that div element and its content will be included in the HTML document; the second expression evaluates as false, which means that the second div element will be excluded. Figure 11-2 shows the result.

Note The ngIf directive adds and removes elements from the HTML document, rather than just showing or hiding them. Use the property or style bindings, described in Chapter 10, if you want to leave elements in place and control their visibility, either by setting the hidden element property to true or by setting the display style property to none.

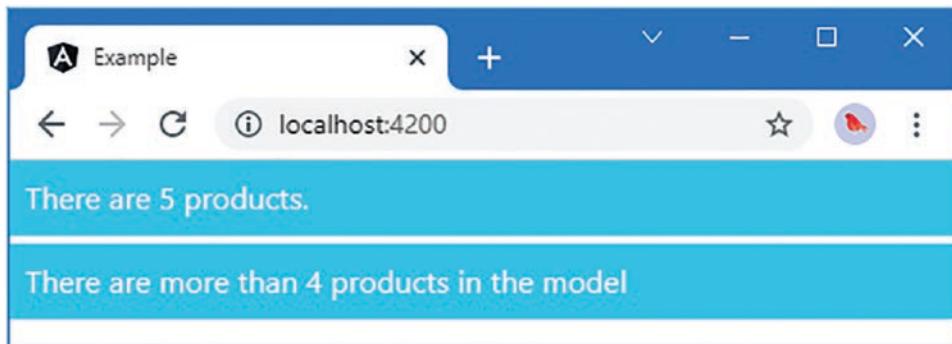


Figure 11-2. Using the `ngIf` directive

UNDERSTANDING MICRO-TEMPLATE DIRECTIVES

Some directives, such as `ngFor`, `ngIf`, and the nested directives used with `ngSwitch`, are prefixed with an asterisk, as in `*ngFor`, `*ngIf`, and `*ngSwitch`. The asterisk is shorthand for using directives that rely on content provided as part of the template, known as a *micro-template*. Directives that use micro-templates are known as *structural directives*, a description that I revisit in Chapter 14 when I show you how to create them.

Listing 11-3 applied the `ngIf` directive to `div` elements, telling the directive to use the `div` element and its content as the micro-template for each of the objects that it processes. Behind the scenes, Angular expands the micro-template and the directive like this:

```
...
<ng-template ngIf="model.getProductCount() > 4">
  <div class="bg-info p-2 mt-1">
    There are more than 4 products in the model
  </div>
</ng-template>
...
```

You can use either syntax in your templates, but if you use the compact syntax, then you must remember to use the asterisk. I explain how to create your own micro-template directives in Chapter 12.

Like all directives, the expression used for `ngIf` will be re-evaluated to reflect changes in the data model. Run the following statements in the browser's JavaScript console to remove the first data object and to run the change detection process:

```
model.products.shift()
appRef.tick()
```

The effect of modifying the model is to remove the first `div` element because there are too few `Product` objects now and to add the second `div` element because the `name` property of the first `Product` in the array is no longer `Kayak`. Figure 11-3 shows the change.

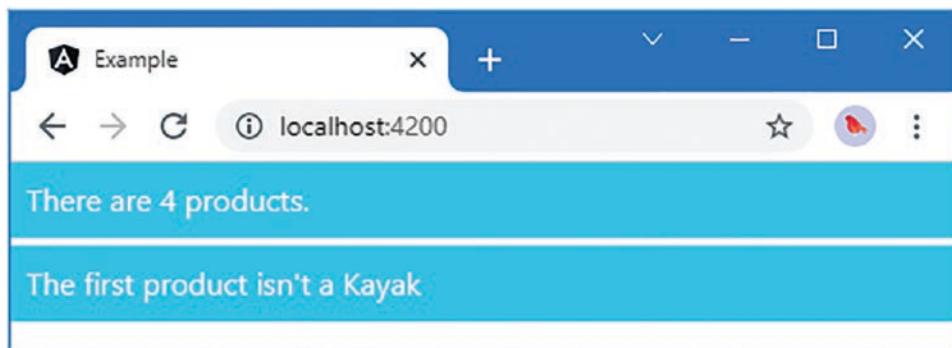


Figure 11-3. The effect of reevaluating directive expressions

Using the ngSwitch Directive

The `ngSwitch` directive selects one of several elements based on the expression result, similar to a JavaScript `switch` statement. Listing 11-4 shows the `ngSwitch` directive being used to choose an element based on the number of objects in the model.

Listing 11-4. Using the `ngSwitch` Directive in the template.html File in the src/app Folder

```
<div class="text-white">
  <div class="bg-info p-2">
    There are {{getProductCount()}} products.
  </div>

  <div class="bg-info p-2 mt-1" [ngSwitch]="getProductCount()">
    <span *ngSwitchCase="2">There are two products</span>
    <span *ngSwitchCase="5">There are five products</span>
    <span *ngSwitchDefault>This is the default</span>
  </div>
</div>
```

The `ngSwitch` directive syntax can be confusing to use. The element that the `ngSwitch` directive is applied to is always included in the HTML document, and the directive name isn't prefixed with an asterisk. It must be specified within square brackets, like this:

```
...
<div class="bg-info p-2 mt-1" [ngSwitch]="getProductCount()">
  ...

```

Each of the inner elements, which are `span` elements in this example, is a micro-template, and the directives that specify the target expression result are prefixed with an asterisk, like this:

```
...
<span *ngSwitchCase="5">There are five products</span>
  ...

```

The `ngSwitchCase` directive is used to specify an expression result. If the `ngSwitch` expression evaluates to the specified result, then that element and its contents will be included in the HTML document. If the expression doesn't evaluate to the specified result, then the element and its contents will be excluded from the HTML document.

The `ngSwitchDefault` directive is applied to a fallback element—equivalent to the `default` label in a JavaScript `switch` statement—which is included in the HTML document if the expression result doesn't match any of the results specified by the `ngSwitchCase` directives.

For the initial data in the application, the directives in Listing 11-4 produce the following HTML:

```
...
<div class="bg-info p-2 mt-1" ng-reflect-ng-switch="5">
  <span>There are five products</span>
</div>
...

```

The `div` element, to which the `ngSwitch` directive has been applied, is always included in the HTML document. For the initial data in the model, the `span` element whose `ngSwitchCase` directive has a result of 5 is also included, producing the result shown on the left of Figure 11-4.

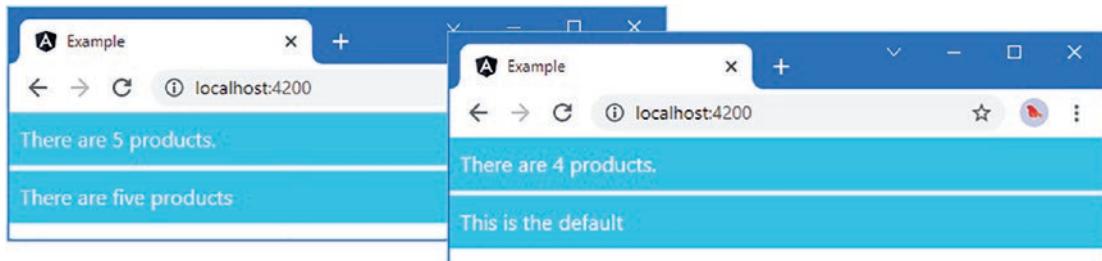


Figure 11-4. Using the `ngSwitch` directive

The `ngSwitch` binding responds to changes in the data model, which you can test by executing the following statements in the browser's JavaScript console:

```
model.products.shift()
appRef.tick()
```

These statements remove the first item from the model and force Angular to run the change detection process. Neither of the results for the two `ngSwitchCase` directives matches the result from the `getProductCount` expression, so the `ngSwitchDefault` element is included in the HTML document, as shown on the right of Figure 11-4.

Avoiding Literal Value Problems

A common problem arises when using the `ngSwitchCase` directive to specify literal string values, and care must be taken to get the right result, as shown in Listing 11-5.

Listing 11-5. Component and String Literal Values in the template.html File in the src/app Folder

```
<div class="text-white">
  <div class="bg-info p-2">
    There are {{getProductCount()}} products.
  </div>

  <div class="bg-info p-2 mt-1" [ngSwitch]="getProduct(1)?.name">
    <span *ngSwitchCase="targetName">Kayak</span>
    <span *ngSwitchCase="'Lifejacket'">Lifejacket</span>
    <span *ngSwitchDefault>Other Product</span>
  </div>
</div>
```

The values assigned to the `ngSwitchCase` directives are also expressions, which means you can invoke methods, perform simple inline operations, and read property values, just as you would for the basic data bindings.

As an example, this expression tells Angular to include the `span` element to which the directive has been applied when the result of evaluating the `ngSwitch` expression matches the value of the `targetName` property defined by the component:

```
...
<span *ngSwitchCase="targetName">Kayak</span>
...
```

If you want to compare a result to a specific string, then you must double quote it, like this:

```
...
<span *ngSwitchCase="'Lifejacket'">Lifejacket</span>
...
```

This expression tells Angular to include the `span` element when the value of the `ngSwitch` expression is equal to the literal string value `Lifejacket`, producing the result shown in Figure 11-5.

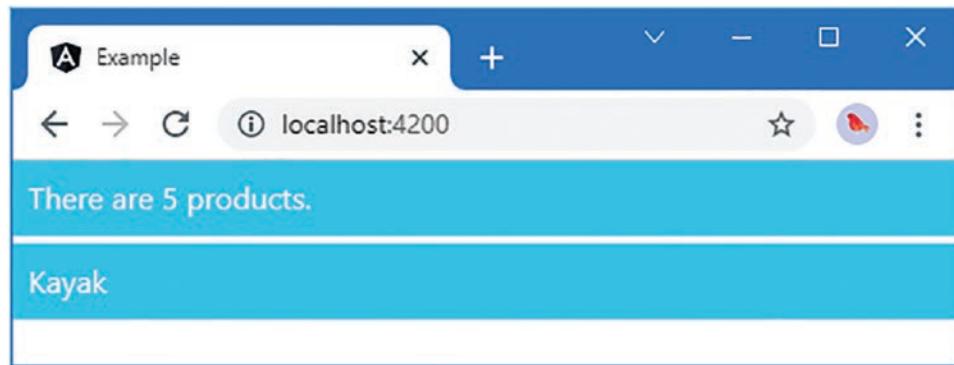


Figure 11-5. Using expressions and literal values with the `ngSwitch` directive

Using the ngFor Directive

The ngFor directive repeats a section of content for each object in an array, providing the template equivalent of a `foreach` loop. In Listing 11-6, I have used the ngFor directive to populate a table by generating a row for each Product object in the model.

Listing 11-6. Using the ngFor Directive in the template.html File in the src/app Folder

```
<div class="text-white">
  <div class="bg-info p-2">
    There are {{getProductCount()}} products.
  </div>

  <div class="p-1">
    <table class="table table-sm table-bordered text-dark">
      <tr><th>Name</th><th>Category</th><th>Price</th></tr>
      <tr *ngFor="let item of getProducts()">
        <td>{{item.name}}</td>
        <td>{{item.category}}</td>
        <td>{{item.price}}</td>
      </tr>
    </table>
  </div>
</div>
```

The expression used with the ngFor directive is more complex than for the other built-in directives, but it will start to make sense when you see how the different parts fit together. Here is the directive that I used in the example:

```
...
<tr *ngFor="let item of getProducts()">
...

```

The asterisk before the name is required because the directive is using a micro-template, as described in the “Understanding Micro-Template Directives” sidebar. This will make more sense as you become familiar with Angular, but at first, you just have to remember that this directive requires an asterisk (or, as I often do, forget until you see an error displayed in the browser’s JavaScript console and *then* remember).

For the expression itself, there are two distinct parts, joined with the `of` keyword. The right-hand part of the expression provides the data source that will be enumerated.

```
...
<tr *ngFor="let item of getProducts()">
...

```

This example specifies the component’s `getProducts` method as the source of data, which allows content to be for each of the `Product` objects in the model. The right-hand side is an expression in its own right, which means you can prepare data or perform simple manipulation operations within the template.

The left-hand side of the ngFor expression defines a *template variable*, denoted by the `let` keyword, which is how data is passed between elements within an Angular template.

```
...
<tr *ngFor="let item of getProducts()">
...

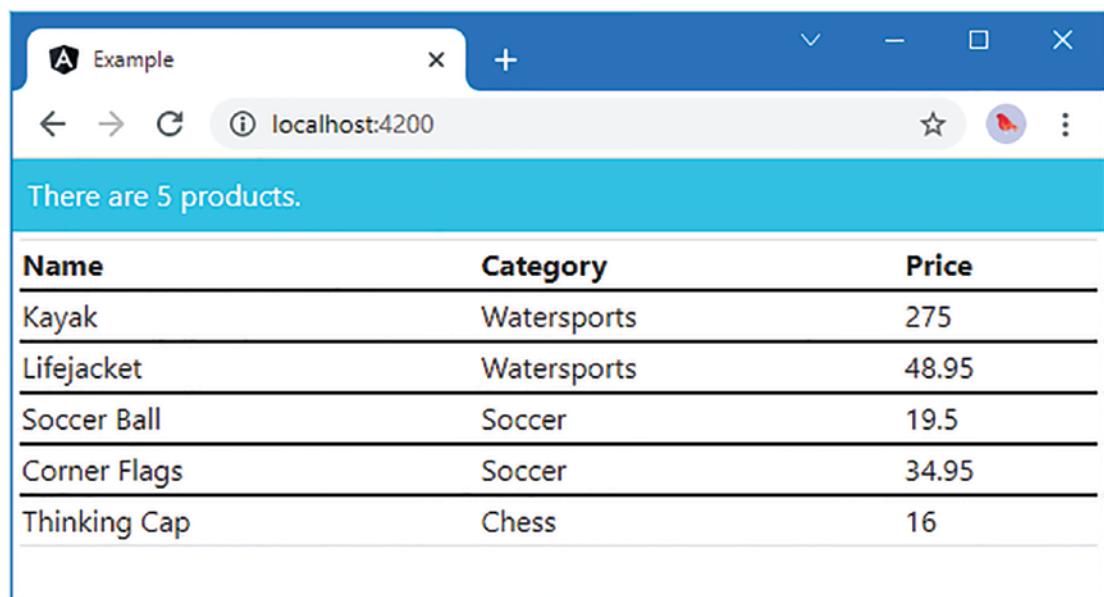
```

The `ngFor` directive assigns the variable to each object in the data source so that it is available for use by the nested elements. The local template variable in the example is called `item`, and it is used to access the `Product` object's properties for the `td` elements, like this:

```
...  
<td>{{item.name}}</td>  
...
```

Put together, the directive in the example tells Angular to enumerate the objects returned by the component's `getProducts` method, assign each of them to a variable called `item`, and then generate a `tr` element and its `td` children, evaluating the template expressions they contain.

For the example in Listing 11-6, the result is a table where the `ngFor` directive is used to generate table rows for each of the `Product` objects in the model and where each table row contains `td` elements that display the value of the `Product` object's `name`, `category`, and `price` properties, as shown in Figure 11-6.

A screenshot of a web browser window titled "Example". The address bar shows "localhost:4200". The page content starts with a blue header bar containing the text "There are 5 products.". Below this is a table with a light gray header row and five data rows. The table has three columns: "Name", "Category", and "Price".

Name	Category	Price
Kayak	Watersports	275
Lifejacket	Watersports	48.95
Soccer Ball	Soccer	19.5
Corner Flags	Soccer	34.95
Thinking Cap	Chess	16

Figure 11-6. Using the `ngFor` directive to create table rows

Using Other Template Variables

The most important template variable is the one that refers to the data object being processed, which is `item` in the previous example. But the `ngFor` directive supports a range of other values that can also be assigned to variables and then referred to within the nested HTML elements, as described in Table 11-4 and demonstrated in the sections that follow.

Table 11-4. The ngFor Local Template Values

Name	Description
index	This number value is assigned to the position of the current object.
count	This number value is assigned the number of elements in the data source.
odd	This boolean value returns true if the current object has an odd-numbered position in the data source.
even	This boolean value returns true if the current object has an even-numbered position in the data source.
first	This boolean value returns true if the current object is the first one in the data source.
last	This boolean value returns true if the current object is the last one in the data source.

Using the Index and Count Value

The index value is set to the position of the current data object and is incremented for each object in the data source. The count value is set to the number of data values in the data source.

In Listing 11-7, I have defined a table that is populated using the ngFor directive and that assigns the index and count values to local template variables, which are then used in a string interpolation binding.

Listing 11-7. Using the Index Value in the template.html File in the src/app Folder

```
<div class="text-white">
  <div class="bg-info p-2">
    There are {{getProductCount()}} products.
  </div>

  <div class="p-1">
    <table class="table table-sm table-bordered text-dark">
      <tr><th></th><th>Name</th><th>Category</th><th>Price</th></tr>
      <tr *ngFor="let item of getProducts(); let i = index; let c = count">
        <td>{{ i + 1 }} of {{ c }}</td>
        <td>{{item.name}}</td>
        <td>{{item.category}}</td>
        <td>{{item.price}}</td>
      </tr>
    </table>
  </div>
</div>
```

A new term is added to the ngFor expression, separated using a semicolon (the ; character). The new expressions uses the let keyword to assign the index value to a local template variable called i and the count value to a local template variable named c, like this:

```
...
<tr *ngFor="let item of getProducts(); let i = index; let c = count">
...
...
```

This allows the values to be accessed within the nested elements using bindings, like this:

```
...
<td>{{ i + 1 }} of {{ c }}</td>
...
```

The index value is zero-based, and adding 1 to the template variable creates a simple counter, producing the result shown in Figure 11-7.

	Name	Category	Price
1 of 5	Kayak	Watersports	275
2 of 5	Lifejacket	Watersports	48.95
3 of 5	Soccer Ball	Soccer	19.5
4 of 5	Corner Flags	Soccer	34.95
5 of 5	Thinking Cap	Chess	16

Figure 11-7. Using the index value

Using the Odd and Even Values

The odd value is true when the index value for a data item is odd. Conversely, the even value is true when the index value for a data item is even. In general, you only need to use either the odd or even value since they are boolean values where odd is true when even is false, and vice versa. In Listing 11-8, the odd value is used to manage the class membership of the tr elements in the table.

Listing 11-8. Using the odd Value in the template.html File in the src/app Folder

```
<div class="text-white">
    <div class="bg-info p-2">
        There are {{getProductCount()}} products.
    </div>

    <div class="p-1">
        <table class="table table-sm table-bordered text-dark">
            <tr><th></th><th>Name</th><th>Category</th><th>Price</th></tr>
            <tr *ngFor="let item of getProducts(); let i = index;
                let c = count; let odd = odd">
```

```

    class="text-white" [class.bg-primary]="odd"
    [class.bg-info]!="odd">
      <td>{{ i + 1 }} of {{ c }}</td>
      <td>{{item.name}}</td>
      <td>{{item.category}}</td>
      <td>{{item.price}}</td>
    </tr>
  </table>
</div>
</div>

```

I have used a semicolon and added another term to the ngFor expression that assigns the odd value to a local template variable that is also called odd.

```

...
<tr *ngFor="let item of getProducts(); let i = index;
  let c = count; let odd = odd"
  class="text-white" [class.bg-primary]="odd"
  [class.bg-info]!="odd">
...

```

This may seem redundant, but you cannot access the ngFor values directly and must use a local variable even if it has the same name. I use the class binding and the odd variable to assign alternate rows to the bg-primary and bg-info classes, which are Bootstrap background color classes that stripe the table rows, as shown in Figure 11-8.

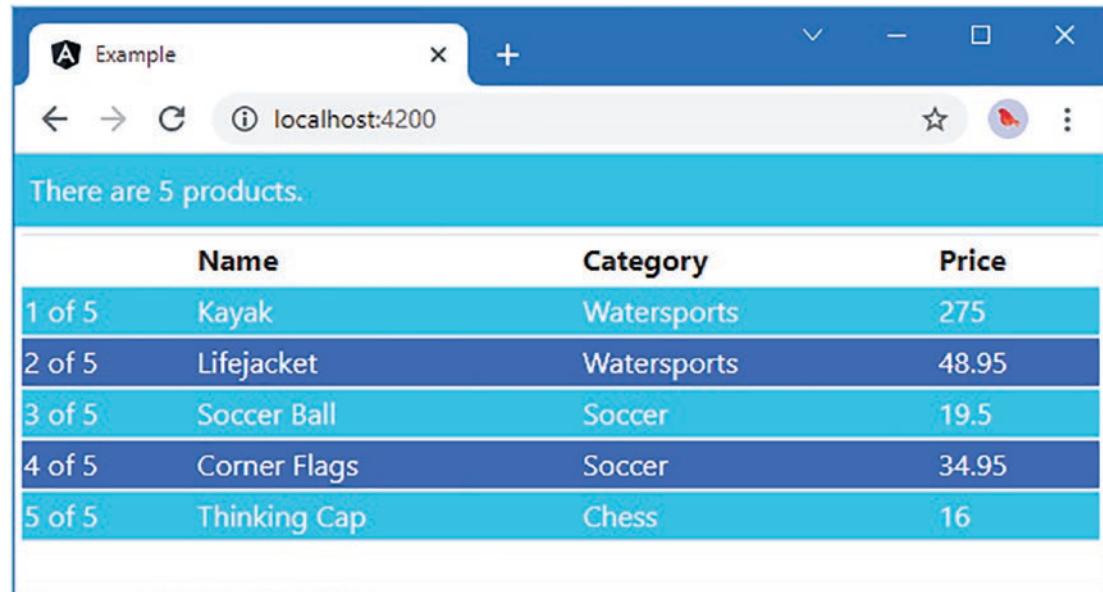


Figure 11-8. Using the odd value

EXPANDING THE *NGFOR DIRECTIVE

Notice that in Listing 11-8, I can use the template variable in expressions applied to the same `tr` element that defines it. This is possible because `ngFor` is a micro-template directive—denoted by the `*` that precedes the name—and so Angular expands the HTML so that it looks like this:

```
...
<table class="table table-sm table-bordered text-dark">
  <tr><th></th><th>Name</th><th>Category</th><th>Price</th></tr>
  <ng-template ngFor let-item [ngForOf]="getProducts()">
    <let-i="index" let-c="count" let-odd="odd">
      <tr class="text-white" [class.bg-primary]="odd" [class.bg-info]!="odd">
        <td>{{ i + 1 }} of {{ c }}</td>
        <td>{{item.name}}</td>
        <td>{{item.category}}</td>
        <td>{{item.price}}</td>
      </tr>
    </ng-template>
  </table>
  ...

```

You can see that the `ng-template` element defines the variables, using the somewhat awkward `let-name` attributes, which are then accessed by the `tr` and `td` elements within it. As with so much in Angular, what appears to happen by magic turns out to be straightforward once you understand what is going on behind the scenes, and I explain these features in detail in Chapter 14. A good reason to use the `*ngFor` syntax is that it provides a more elegant way to express the directive expression, especially when there are multiple template variables.

Using the First and Last Values

The `first` value is `true` only for the first object in the sequence provided by the data source and is `false` for all other objects. Conversely, the `last` value is `true` only for the last object in the sequence. Listing 11-9 uses these values to treat the first and last objects differently from the others in the sequence.

Listing 11-9. Using the first and last Values in the template.html File in the src/app Folder

```
<div class="text-white">
  <div class="bg-info p-2">
    There are {{getProductCount()}} products.
  </div>

  <div class="p-1">
    <table class="table table-sm table-bordered text-dark">
      <tr><th></th><th>Name</th><th>Category</th><th>Price</th></tr>
      <tr *ngFor="let item of getProducts(); let i = index;
        let c = count; let odd = odd; let first = first;
        let last = last">
```

```

    class="text-white" [class.bg-primary]="odd"
    [class.bg-info]!="odd"
    [class.bg-warning]="first || last">
      <td>{{ i + 1 }} of {{ c }}</td>
      <td>{{item.name}}</td>
      <td>{{item.category}}</td>
      <td *ngIf="!last">{{item.price}}</td>
    </tr>
  </table>
</div>
</div>

```

The new terms in the ngFor expression assign the `first` and `last` values to template variables called `first` and `last`. These variables are then used by a class binding on the `tr` element, which assigns the element to the `bg-warning` class when either is true, and are used by the `ngIf` directive on one of the `td` elements, which will exclude the element for the last item in the data source, producing the effect shown in Figure 11-9.

There are 5 products.			
	Name	Category	Price
1 of 5	Kayak	Watersports	275
2 of 5	Lifejacket	Watersports	48.95
3 of 5	Soccer Ball	Soccer	19.5
4 of 5	Corner Flags	Soccer	34.95
5 of 5	Thinking Cap	Chess	

Figure 11-9. Using the `first` and `last` values

Minimizing Element Operations

When there is a change to the data model, the `ngFor` directive evaluates its expression and updates the elements that represent its data objects. The update process can be expensive, especially if the data source is replaced with one that contains different objects representing the same data. Replacing the data source may seem like an odd thing to do, but it happens often in web applications, especially when the data is retrieved from a web service, like the ones I describe in Chapter 23. The same data values are represented by new objects, which presents an efficiency problem for Angular. To demonstrate the problem, I added a method to the component that replaces one of the `Product` objects in the data model, as shown in Listing 11-10.

Listing 11-10. Replacing an Object in the repository.model.ts File in the src/app Folder

```

import { Product } from "./product.model";
import { SimpleDataSource } from "./datasource.model";

export class Model {
    private dataSource: SimpleDataSource;
    private products: Product[];
    private locator = (p: Product, id: number | any) => p.id == id;

    constructor() {
        this.dataSource = new SimpleDataSource();
        this.products = new Array<Product>();
        this.dataSource.getData().forEach(p => this.products.push(p));
    }

    // ...methods omitted for brevity...

    swapProduct() {
        let p = this.products.shift();
        if (p != null) {
            this.products.push(new Product(p.id, p.name, p.category, p.price));
        }
    }
}

```

The swapProduct method removes the first object from the array and adds a new object that has the same values for the id, name, category, and price properties. This is an example of data values being represented by a new object.

Run the following statements using the browser's JavaScript console to modify the data model and run the change-detection process:

```
model.swapProduct()
appRef.tick()
```

When the ngFor directive examines its data source, it sees it has two operations to perform to reflect the change to the data. The first operation is to destroy the HTML elements that represent the first object in the array. The second operation is to create a new set of HTML elements to represent the new object at the end of the array.

Angular has no way of knowing that the data objects it is dealing with have the same values and that it could perform its work more efficiently by simply moving the existing elements within the HTML document.

This problem affects only two elements in this example, but the problem is much more severe when the data in the application is refreshed from an external data source, such as a web service, where all the data model objects can be replaced each time that a response is received. Since it is not aware that there have been few real changes, the ngFor directive has to destroy all of its HTML elements and create new ones, which can be an expensive and time-consuming operation.

To improve the efficiency of an update, you can define a component method that will help Angular determine when two different objects represent the same data, as shown in Listing 11-11.

Listing 11-11. Adding the Object Comparison Method in the component.ts File in the src/app Folder

```
import { ApplicationRef, Component } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";

@Component({
    selector: "app",
    templateUrl: "template.html"
})
export class ProductComponent {
    model: Model = new Model();

    // ...constructor and methods omitted for brevity...

    targetName: string = "Kayak";

    getKey(index: number, product: Product) {
        return product.id;
    }
}
```

The method has to define two parameters: the position of the object in the data source and the data object. The result of the method uniquely identifies an object, and two objects are considered to be equal if they produce the same result.

Two Product objects will be considered equal if they have the same id value. Telling the ngFor expression to use the comparison method is done by adding a trackBy term to the expression, as shown in Listing 11-12.

Listing 11-12. Providing an Equality Method in the template.html File in the src/app Folder

```
<div class="text-white">
    <div class="bg-info p-2">
        There are {{getProductCount()}} products.
    </div>

    <div class="p-1">
        <table class="table table-sm table-bordered text-dark">
            <tr><th></th><th>Name</th><th>Category</th><th>Price</th></tr>
            <tr *ngFor="let item of getProducts(); let i = index;">
                let c = count; let odd = odd; let first = first;
                let last = last; trackBy:getKey"
                class="text-white" [class.bg-primary]="odd"
                [class.bg-info]="!odd"
                [class.bg-warning]="first || last">
                    <td>{{ i + 1 }} of {{ c }}</td>
                    <td>{{item.name}}</td>
                    <td>{{item.category}}</td>
                    <td *ngIf="!last">{{item.price}}</td>
            </tr>
        </table>
    </div>
</div>
```

With this change, the `ngFor` directive will know that the `Product` that is removed from the array using the `swapProduct` method defined in Listing 11-12 is equivalent to the one that is added to the array, even though they are different objects. Rather than delete and create elements, the existing elements can be moved, which is a much simpler and quicker task to perform.

Changes can still be made to the elements—such as by the `ngIf` directive, which will remove one of the `td` elements because the new object will be the last item in the data source, but even this is faster than treating the objects separately.

TESTING THE EQUALITY METHOD

Checking whether the equality method has an effect is a little tricky. The best way that I have found requires using the browser's F12 developer tools, in this case using the Chrome browser.

Once the application has loaded, right-click the `td` element that contains the word *Kayak* in the browser window and select `Inspect` from the pop-up menu. This will open the Developer Tools window and show the Elements panel.

Click the ellipsis button (marked ...) in the left margin and select `Add Attribute` from the menu. Add an `id` attribute with the value `old`. This will result in an element that looks like this:

```
<td id="old">Kayak</td>
```

Adding an `id` attribute makes it possible to access the object that represents the HTML element using the JavaScript console. Switch to the Console panel and enter the following statement:

```
window.old
```

When you hit Return, the browser will locate the element by its `id` attribute value and display the following result:

```
<td id="old">Kayak</td>
```

Now execute the following statements in the JavaScript console, hitting Return after each one:

```
model.swapProduct()  
appRef.tick()
```

Once the change to the data model has been processed, executing the following statement in the JavaScript console will determine whether the `td` element to which the `id` attribute was added has been moved or destroyed:

```
window.old
```

If the element has been moved, then you will see the element shown in the console, like this:

```
<td id="old">Kayak</td>
```

If the element has been destroyed, then there won't be an element whose `id` attribute is `old`, and the browser will display the word `undefined`.

Using the ngTemplateOutlet Directive

The `ngTemplateOutlet` directive is used to repeat a block of content at a specified location, which can be useful when you need to generate the same content in different places and want to avoid duplication. Listing 11-13 replaces the contents of the `template.html` file to show the `ngTemplateOutlet` directive in use.

Listing 11-13. Replacing the Contents of the `template.html` File in the `src/app` Folder

```
<ng-template #titleTemplate>
  <h4 class="p-2 bg-success text-white">Repeated Content</h4>
</ng-template>

<ng-template [ngTemplateOutlet]="titleTemplate"></ng-template>

<div class="bg-info p-2 m-2 text-white">
  There are {{getProductCount()}} products.
</div>

<ng-template [ngTemplateOutlet]="titleTemplate"></ng-template>
```

The first step is to define the template that contains the content that you want to repeat using the directive. This is done using the `ng-template` element and assigning it a name using a *reference variable*, like this:

```
...
<ng-template #titleTemplate let-title="title">
  <h4 class="p-2 bg-success text-white">Repeated Content</h4>
</ng-template>
...
```

When Angular encounters the reference variable, it sets its value to the element to which it has been defined, which is the `ng-template` element in this case. The second step is to insert the content into the HTML document, using the `ngTemplateOutlet` directive, like this:

```
...
<ng-template [ngTemplateOutlet]="titleTemplate"></ng-template>
...
```

The expression is the name of the reference variable that was assigned to the content that should be inserted. The directive replaces the host element with the contents of the specified `ng-template` element. Neither the `ng-template` element that contains the repeated content nor the one that is the host element for the binding is included in the HTML document. Figure 11-10 shows how the directive has used the repeated content.

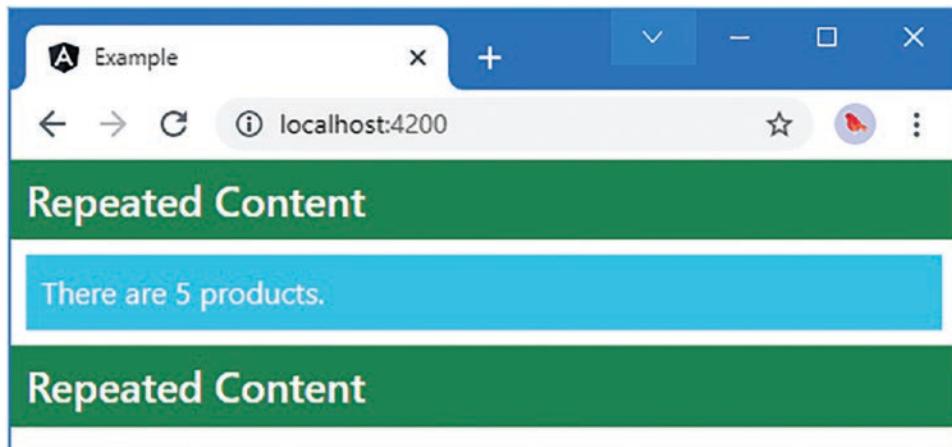


Figure 11-10. Using the `ngTemplateOutlet` directive

Providing Context Data

The `ngTemplateOutlet` directive can be used to provide the repeated content with a context object that can be used in data bindings defined within the `ng-template` element, as shown in Listing 11-14.

Listing 11-14. Providing Context Data in the template.html File in the src/app Folder

```
<ng-template #titleTemplate let-text="title">
    <h4 class="p-2 bg-success text-white">{{text}}</h4>
</ng-template>

<ng-template [ngTemplateOutlet]="titleTemplate"
    [ngTemplateOutletContext]="{title: 'Header'}">
</ng-template>

<div class="bg-info p-2 m-2 text-white">
    There are {{getProductCount()}} products.
</div>

<ng-template [ngTemplateOutlet]="titleTemplate"
    [ngTemplateOutletContext]="{title: 'Footer'}">
</ng-template>
```

To receive the context data, the `ng-template` element that contains the repeated content defines a `let-` attribute that specifies the name of a variable, similar to the expanded syntax used for the `ngFor` directive. The value of the expression assigns the `let-` variable a value, like this:

```
...
<ng-template #titleTemplate let-text="title">
...

```

The `let-` attribute in this example creates a variable called `text`, which is assigned a value by evaluating the expression `title`. To provide the data against which the expression is evaluated, the `ng-template` element to which the `ngTemplateOutletContext` directive has been applied provides a map object, like this:

```
...
<ng-template [ngTemplateOutlet]="titleTemplate"
  [ngTemplateOutletContext]="{title: 'Footer'}">
</ng-template>
...
```

The target of this new binding is `ngTemplateOutletContext`, which looks like another directive but is actually an example of an *input property*, which some directives use to receive data values and that I describe in detail in Chapter 13. The expression for the binding is a map object whose property name corresponds to the `let-` attribute on the other `ng-template` element. The result is that the repeated content can be tailored using bindings, as shown in Figure 11-11.

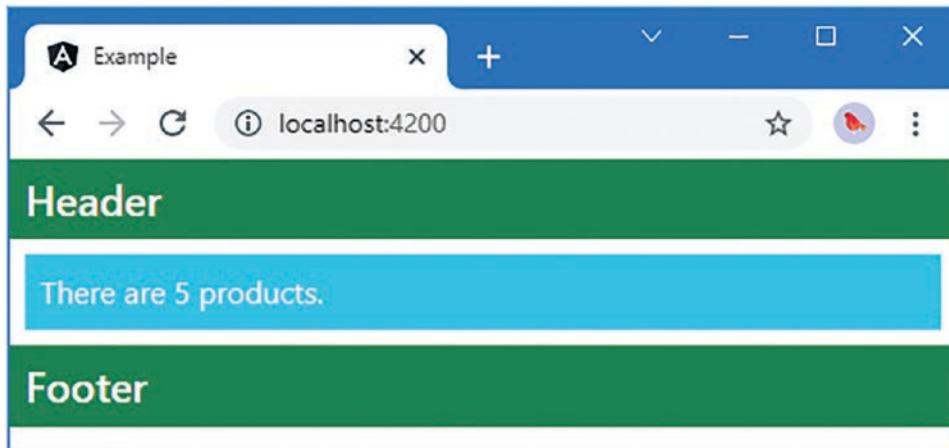


Figure 11-11. Providing context data for repeated content

Using Directives Without an HTML Element

The `ng-container` element can be used to apply directives without using an HTML element, which can be useful when you want to generate content without adding to the structure of the HTML document displayed by the browser, as shown in Listing 11-15, which replaces the contents of the `template.html` file.

Listing 11-15. Generating Content Without an Element in the template.html File in the src/app Folder

```
<div class="bg-info p-2 text-white">
  Product Names:
  <ng-container *ngFor="let item of getProducts(); let last = last">
    {{ item.name }}<ng-container *ngIf="!last">,</ng-container>
  </ng-container>
</div>
```

The `ng-container` element doesn't appear in the HTML displayed by the browser, which means that it can be used to generate content within elements. In this example, the `ng-container` element is used to apply the `ngFor` directive, and the content it produces contains a second `ng-container` element that applies the `ngIf` directive. The result is a string that introduces no new elements, as shown in Figure 11-12.

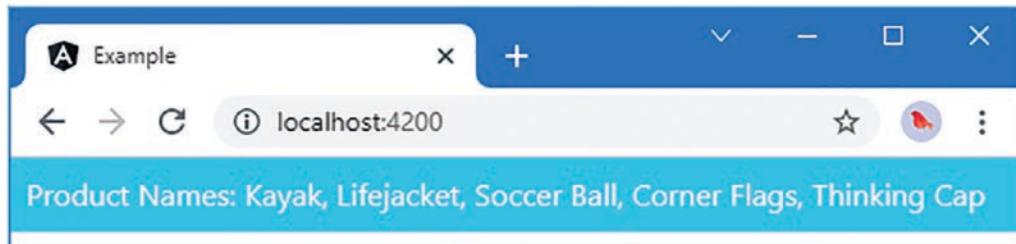


Figure 11-12. Using directives without an HTML element

Understanding One-Way Data Binding Restrictions

Although the expressions used in one-way data binding and directives look like JavaScript code, you can't use all the JavaScript—or TypeScript—language features. I explain the restrictions and the reasons for them in the sections that follow.

Using Idempotent Expressions

One-way data bindings must be *idempotent*, meaning that they can be evaluated repeatedly without changing the state of the application. To demonstrate why, I added a debugging statement to the component's `getProducts` method, as shown in Listing 11-16.

Note Angular *does* support modifying the application state, but it must be done using the techniques I describe in Chapter 12.

Listing 11-16. Adding a Statement in the component.ts File in the src/app Folder

```
...
getProducts(): Product[] {
  console.log("getProducts invoked");
  return this.model.getProducts();
}
...
```

When the changes are saved and the browser reloads the page, you will see a long series of messages like these in the browser's JavaScript console:

```
...
getProducts invoked
getProducts invoked
getProducts invoked
getProducts invoked
...
...
```

As the messages show, Angular evaluates the binding expression several times before displaying the content in the browser. If an expression modifies the state of an application, such as removing an object from a queue, you won't get the results you expect by the time the template is displayed to the user. To avoid this problem, Angular restricts the way that expressions can be used. In Listing 11-17, I added a counter property to the component to help demonstrate.

Listing 11-17. Adding a Property in the component.ts File in the src/app Folder

```
import { ApplicationRef, Component } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";

@Component({
    selector: "app",
    templateUrl: "template.html"
})
export class ProductComponent {
    model: Model = new Model();

    // ...members omitted for brevity...

    getKey(index: number, product: Product) {
        return product.id;
    }

    counter: number = 1;
}
```

In Listing 11-18, I added a binding whose expression increments the counter when it is evaluated.

Listing 11-18. Adding a Binding in the template.html File in the src/app Folder

```
<div class="bg-info p-2 text-white">
    Product Names:
    <ng-container *ngFor="let item of getProducts(); let last = last">
        {{ item.name}}<ng-container *ngIf="!last">,</ng-container>
    </ng-container>
</div>
```

```
<div class="bg-info p-2">
  Counter: {{counter = counter + 1}}
</div>
```

When the browser loads the page, you will see the following error:

```
...
Error: src/app/template.html:9:5 - error NG5002: Parser Error:
  Bindings cannot contain assignments at column 11 in
  [ Counter: {{counter = counter + 1}} ] in C:\example\src\app\template.html@8:4
9    Counter: {{counter = counter + 1}}
  ~~~~~
10   </div>
  ~~
src/app/component.ts:7:15
  7  templateUrl: "template.html"
  ~~~~~
  Error occurs in the template of component ProductComponent.
...
```

Angular will report an error if a data binding expression contains an operator that can be used to perform an assignment, such as `=`, `+=`, `-+`, `++`, and `--`. In addition, when Angular is running in development mode, it performs an additional check to make sure that one-way data bindings have not been modified after their expressions are evaluated. To demonstrate, Listing 11-19 adds a property to the component that removes and returns a `Product` object from the model array.

Listing 11-19. Modifying Data in the component.ts File in the src/app Folder

```
import { ApplicationRef, Component } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";

@Component({
  selector: "app",
  templateUrl: "template.html"
})
export class ProductComponent {
  model: Model = new Model();

  // ...members omitted for brevity...

  counter: number = 1;

  get nextProduct(): Product | undefined {
    return this.model.getProducts().shift();
  }
}
```

In Listing 11-20, you can see the data binding that I used to read the `nextProduct` property.

Listing 11-20. Binding to a Property in the template.html File in the src/app Folder

```
<div class="bg-info p-2 text-white">
  Product Names:
  <ng-container *ngFor="let item of getProducts(); let last = last">
    {{ item.name}}<ng-container *ngIf="!last">,</ng-container>
  </ng-container>
</div>

<div class="bg-info p-2 text-white">
  Next Product is {{nextProduct?.name}}
</div>
```

When the browser reloads, you will see the following error in the JavaScript console:

```
...
ERROR Error: NG0100: ExpressionChangedAfterItHasBeenCheckedError: Expression has changed
after it was checked. Previous value: 'Lifejacket'. Current value: 'Corner Flags'.. Find
more at https://angular.io/errors/NG0100
...
```

Understanding the Expression Context

When Angular evaluates an expression, it does so in the context of the template's component, which is how the template can access methods and properties without any kind of prefix, like this:

```
...
<div class="bg-info p-2 text-white">
  Next Product is {{nextProduct?.name}}
</div>
...
```

When Angular processes these expressions, the component provides the `nextProduct` property, which Angular incorporates into the HTML document. The component is said to provide the template's *expression context*.

The expression context means you can't access objects defined outside of the template's component, and in particular, templates can't access the global namespace. The global namespace is used to define common utilities, such as the `console` object, which defines the `log` method I have been using to write out debugging information to the browser's JavaScript console. The global namespace also includes the `Math` object, which provides access to some useful arithmetic methods, such as `min` and `max`.

To demonstrate this restriction, Listing 11-21 adds a string interpolation binding to the template that relies on the `Math.floor` method to round down a number value to the nearest integer.

Listing 11-21. Accessing the Global Namespace in the template.html File in the src/app Folder

```
<div class="bg-info p-2 text-white">
    Product Names:
    <ng-container *ngFor="let item of getProducts(); let last = last">
        {{ item.name }}<ng-container *ngIf="!last">,</ng-container>
    </ng-container>
</div>

<div class='bg-info p-2'>
    The rounded price is {{Math.floor(getProduct(1)?.price)}}
</div>
```

When Angular processes the template, it will produce the following error in the browser's JavaScript console:

```
error TS2339: Property 'Math' does not exist on type 'ProductComponent'.
```

The error message doesn't specifically mention the global namespace. Instead, Angular has tried to evaluate the expression using the component as the context and failed to find a `Math` property.

If you want to access functionality in the global namespace, then it must be provided by the component, acting on behalf of the template. In the case of the example, the component could just define a `Math` property that is assigned to the global object, but template expressions should be as clear and simple as possible, so a better approach is to define a method that provides the template with the specific functionality it requires, as shown in Listing 11-22.

Listing 11-22. Defining a Method in the component.ts File in the src/app Folder

```
import { ApplicationRef, Component } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";

@Component({
    selector: "app",
    templateUrl: "template.html"
})
export class ProductComponent {
    model: Model = new Model();

    // ...members omitted for brevity...

    get nextProduct(): Product | undefined {
        return this.model.getProducts().shift();
    }

    getProductPrice(index: number): number {
        return Math.floor(this.getProduct(index)?.price ?? 0);
    }
}
```

In Listing 11-23, I have changed the data binding in the template to use the newly defined method.

Listing 11-23. Access Global Namespace Functionality in the template.html File in the src/app Folder

```
<div class="bg-info p-2 text-white">
  Product Names:
  <ng-container *ngFor="let item of getProducts(); let last = last">
    {{ item.name }}<ng-container *ngIf="!last">,</ng-container>
  </ng-container>
</div>

<div class="bg-info p-2">
  The rounded price is {{getProductPrice(2)}}
</div>
```

When Angular processes the template, it will call the `getProductPrice` method and indirectly take advantage of the `Math` object in the global namespace, producing the result shown in Figure 11-13.

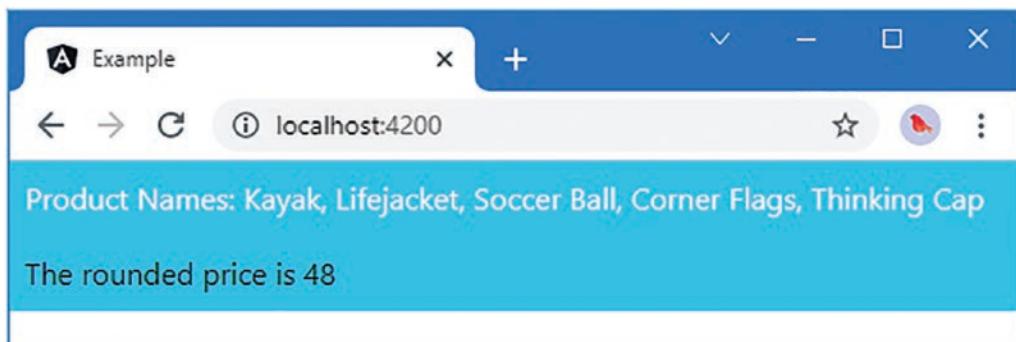


Figure 11-13. Accessing global namespace functionality

Summary

In this chapter, I explained how to use the built-in template directives. I showed you how to select content with the `ngIf` and `ngSwitch` directives and how to repeat content using the `ngFor` directive. I explained why some directive names are prefixed with an asterisk and described the limitations that are placed on template expressions used with these directives and with one-way data bindings in general. In the next chapter, I describe how data bindings are used for events and form elements.

CHAPTER 12



Using Events and Forms

In this chapter, I continue describing the basic Angular functionality, focusing on features that respond to user interaction. I explain how to create event bindings and how to use two-way bindings to manage the flow of data between the model and the template. One of the main forms of user interaction in a web application is the use of HTML forms, and I explain how event bindings and two-way data bindings are used to support them and validate the content that the user provides. Table 12-1 puts events and forms in context.

Table 12-1. Putting Event Bindings and Forms in Context

Question	Answer
What are they?	Event bindings evaluate an expression when an event is triggered, such as a user pressing a key, moving the mouse, or submitting a form. The broader form-related features build on this foundation to create forms that are automatically validated to ensure that the user provides useful data.
Why are they useful?	These features allow the user to change the state of the application, changing or adding to the data in the model.
How are they used?	Each feature is used differently. See the examples for details.
Are there any pitfalls or limitations?	In common with all Angular bindings, the main pitfall is using the wrong kind of bracket to denote a binding. Pay close attention to the examples in this chapter and check the way you have applied bindings when you don't get the results you expect.
Are there any alternatives?	No. These features are a core part of Angular.

Table 12-2 summarizes the chapter.

Table 12-2. Chapter Summary

Problem	Solution	Listing
Enabling forms support	Add the @angular/forms module to the application	1-3
Responding to an event	Use an event binding	4-6
Getting details of an event	Use the \$event object	7-9
Referring to elements in the template	Define template variables	10
Enabling the flow of data in both directions between the element and the component	Use a two-way data binding	11, 12
Capturing user input	Use an HTML form	13, 14
Validating the data provided by the user	Perform form validation	15-26

Preparing the Example Project

For this chapter, I will continue using the example project that I created in Chapter 9 and have been modifying in the chapters since.

Tip You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/Apress/pro-angular-5ed>. See Chapter 1 for how to get help if you have problems running the examples.

Importing the Forms Module

The features demonstrated in this chapter rely on the Angular forms module, which must be imported to the Angular module, as shown in Listing 12-1.

Listing 12-1. Declaring a Dependency in the app.module.ts File in the src/app Folder

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

import { ProductComponent } from './component';
import { FormsModule } from "@angular/forms";

@NgModule({
  declarations: [ProductComponent],
  imports: [
    BrowserModule,
```

```

    BrowserAnimationsModule,
  FormsModule
],
providers: [],
bootstrap: [ProductComponent]
})
export class AppModule { }

```

The imports property of the NgModule decorator specifies the dependencies of the application. Adding **FormsModule** to the list of dependencies enables the form features and makes them available for use throughout the application.

Preparing the Component and Template

Listing 12-2 removes the constructor and some of the methods from the component class and adds a new property, named selectedProduct.

Listing 12-2. Simplifying the Component in the component.ts File in the src/app Folder

```

import { Component } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";

@Component({
  selector: "app",
  templateUrl: "template.html"
})
export class ProductComponent {
  model: Model = new Model();

  getProduct(key: number): Product | undefined {
    return this.model.getProduct(key);
  }

  getProducts(): Product[] {
    return this.model.getProducts();
  }

  selectedProduct: string | undefined;
}

```

Listing 12-3 simplifies the component's template, leaving just a table that is populated using the `ngFor` directive.

Listing 12-3. Simplifying the Template in the template.html File in the src/app Folder

```

<div class="p-2">
  <table class="table table-sm table-bordered">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th></tr>
    <tr *ngFor="let item of getProducts(); let i = index">
      <td>{{i + 1}}</td>

```

```

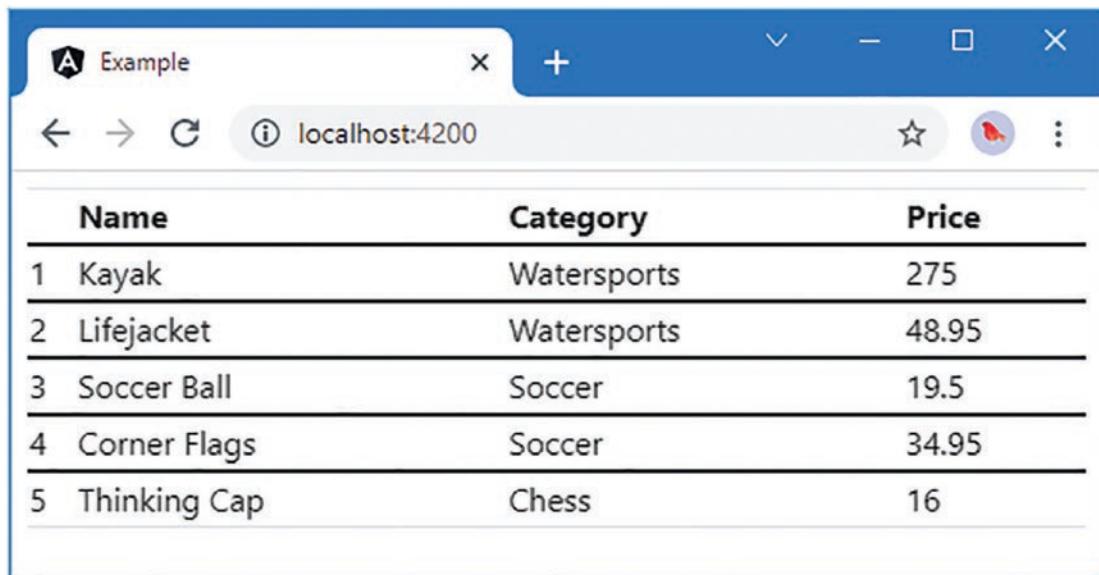
<td>{{item.name}}</td>
<td>{{item.category}}</td>
<td>{{item.price}}</td>
</tr>
</table>
</div>

```

To start the development server, open a command prompt, navigate to the example folder, and run the following command:

```
ng serve
```

Open a new browser window and navigate to <http://localhost:4200> to see the table shown in Figure 12-1.



The screenshot shows a web browser window titled "Example". The address bar indicates the URL is "localhost:4200". The main content is a table with five rows, each representing a product. The columns are labeled "Name", "Category", and "Price". The data is as follows:

	Name	Category	Price
1	Kayak	Watersports	275
2	Lifejacket	Watersports	48.95
3	Soccer Ball	Soccer	19.5
4	Corner Flags	Soccer	34.95
5	Thinking Cap	Chess	16

Figure 12-1. Running the example application

Using the Event Binding

The *event binding* is used to respond to the events sent by the host element. Listing 12-4 demonstrates the event binding, which allows a user to interact with an Angular application.

Listing 12-4. Using the Event Binding in the template.html File in the src/app Folder

```

<div class="p-2">
  <div class="bg-info text-white p-2">
    Selected Product: {{selectedProduct ?? '(None)'}}
  </div>

```

```


|           | Name          | Category          | Price          |
|-----------|---------------|-------------------|----------------|
| {{i + 1}} | {{item.name}} | {{item.category}} | {{item.price}} |


```

When you save the changes to the template, you can test the binding by moving the mouse pointer over the first column in the HTML table, which displays a series of numbers. As the mouse moves from row to row, the name of the product displayed in that row is displayed at the top of the page, as shown in Figure 12-2.

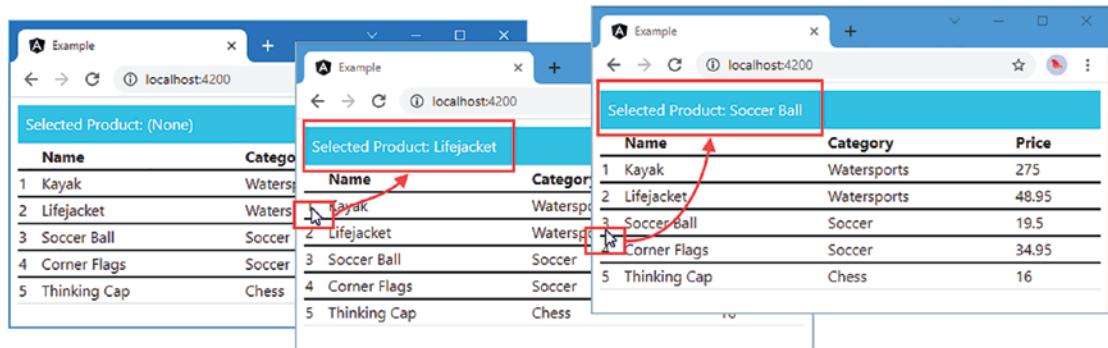


Figure 12-2. Using an event binding

This is a simple example, but it shows the structure of an event binding, which is illustrated in Figure 12-3.

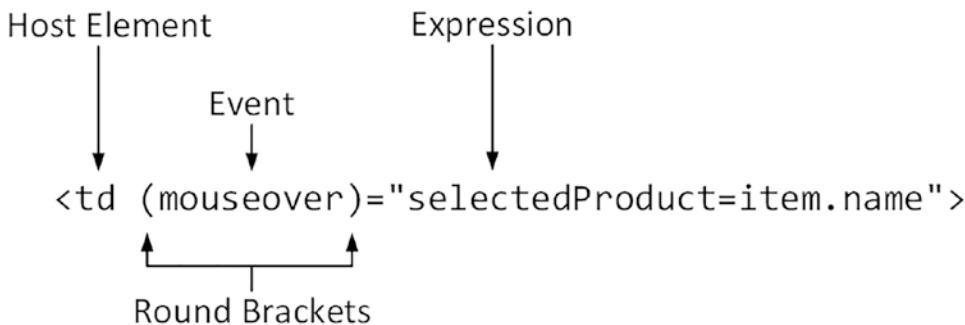


Figure 12-3. The anatomy of an event binding

An event binding has these four parts:

- The *host element* is the source of events for the binding.
- The *round brackets* tell Angular that this is an event binding, which is a form of one-way binding where data flows from the element to the rest of the application.
- The *event* specifies which event the binding is for.
- The *expression* is evaluated when the event is triggered.

Looking at the binding in Listing 12-4, you can see that the host element is a `td` element, meaning that this is the element that will be the source of events. The binding specifies the `mouseover` event, which is triggered when the mouse pointer moves over the part of the screen occupied by the host element.

Unlike one-way bindings, the expressions in event bindings can make changes to the state of the application and can contain assignment operators, such as `=`. The expression for the binding assigns the value of the `item.name` property to a variable called `selectedProduct`. The `selectedProduct` variable is used in a string interpolation binding at the top of the template, like this:

```
...
<div class="bg-info text-white p-2">
    Selected Product: {{selectedProduct ?? '(None)'}}
</div>
...
```

The value displayed by the string interpolation binding is updated when the value of the `selectedProduct` variable is changed by the event binding. Manually starting the change detection process using the `ApplicationRef.tick` method is no longer required because the bindings and directives in this chapter take care of the process automatically.

WORKING WITH DOM EVENTS

If you are unfamiliar with the events that an HTML element can send, then there is a good summary available at <https://developer.mozilla.org/en-US/docs/Web/Events>. There are a lot of events, however, and not all of them are supported widely or consistently in all browsers. A good place to start is the “DOM Events” and “HTML DOM Events” sections of the mozilla.org page, which define the basic interactions that a user has with an element (clicking, moving the pointer, submitting forms, and so on) and that can be relied on to work in most browsers.

If you use the less common events, then you should make sure they are available and work as expected in your target browsers. The excellent <http://caniuse.com> provides details of which features are implemented by different browsers, but you should also perform thorough testing.

The expression that displays the selected product uses the nullish coalescing operator to ensure that the user always sees a message, even when no product is selected. A neater approach is to define a method that performs this check, as shown in Listing 12-5.

Listing 12-5. Enhancing the Component in the component.ts File in the src/app Folder

```

import { Component } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";

@Component({
  selector: "app",
  templateUrl: "template.html"
})
export class ProductComponent {
  model: Model = new Model();

  getProduct(key: number): Product | undefined {
    return this.model.getProduct(key);
  }

  getProducts(): Product[] {
    return this.model.getProducts();
  }

  selectedProduct: string | undefined;

  getSelected(product: Product): boolean {
    return product.name == this.selectedProduct;
  }
}

```

I have defined a method called `getSelected` that accepts a `Product` object and compares its name to the `selectedProduct` property. In Listing 12-6, the `getSelected` method is used by a class binding to control membership of the `bg-info` class, which is a Bootstrap class that assigns a background color to an element.

Listing 12-6. Setting Class Membership in the template.html File in the src/app Folder

```

<div class="p-2">
  <div class="bg-info text-white p-2">
    Selected Product: {{selectedProduct ?? '(None)'}}
```

<tr *ngFor="let item of getProducts(); let i = index">				
[class.bg-info]="getSelected(item)">				
<td (mouseover)="selectedProduct=item.name">{{i + 1}}</td>				
<td>{{item.name}}</td>				
<td>{{item.category}}</td>				
<td>{{item.price}}</td>				
</tr>				
</table>				

</div>

The result is that `tr` elements are added to the `bg-info` class when the `selectedProduct` property value matches the `Product` object used to create them, which is changed by the event binding when the `mouseover` event is triggered, as shown in Figure 12-4.

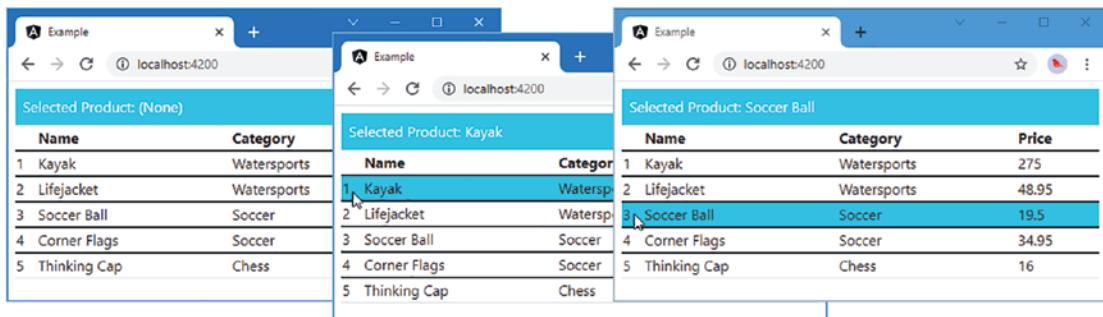


Figure 12-4. Highlighting table rows through an event binding

This example shows how user interaction drives new data into the application and starts the change-detection process, causing Angular to reevaluate the expressions used by the string interpolation and class bindings. This flow of data is what brings Angular applications to life: the bindings and directives described in Chapters 10 and 11 respond dynamically to changes in the application state, creating content generated and managed entirely within the browser.

Using Event Data

The previous example used the event binding to connect two pieces of data provided by the component: when the mouseevent is triggered, the binding's expression sets the `selectedProduct` property using a data value that was provided to the `ngFor` directive by the component's `getProducts` method.

The event binding can also be used to introduce new data into the application from the event itself, using details that are provided by the browser. Listing 12-7 adds an `input` element to the template and uses the event binding to listen for the `input` event, which is triggered when the content of the `input` element changes.

Listing 12-7. Using an Event Object in the template.html File in the src/app Folder

```

<div class="p-2">
  <div class="bg-info text-white p-2">
    Selected Product: {{selectedProduct ?? '(None)'}}
  </div>
  <table class="table table-sm table-bordered">
    <tr *ngFor="let item of getProducts(); let i = index"
       [class.bg-info]="getSelected(item)">
      <td (mouseover)="selectedProduct=item.name">{{i + 1}}</td>
      <td>{{item.name}}</td>
      <td>{{item.category}}</td>
      <td>{{item.price}}</td>
    </tr>
  </table>
  <div class="form-group">
    <label>Product Name</label>
    <input class="form-control"
          (input)="selectedProduct=$any($event).target.value" />
  </div>
</div>

```

When the browser triggers an event, it provides an Event object that describes it. There are different types of event objects for different categories of events (mouse events, keyboard events, form events, and so on), but all events share the three properties described in Table 12-3.

Table 12-3. The Properties Common to All DOM Event Objects

Name	Description
type	This property returns a string that identifies the type of event that has been triggered.
target	This property returns the object that triggered the event, which will generally be the object that represents the HTML element in the DOM.
timeStamp	This property returns a number that contains the time that the event was triggered, expressed as milliseconds since January 1, 1970.

The Event object is assigned to a template variable called \$event, and the binding expression in Listing 12-7 uses this variable to access the event and its target property, like this:

```
...
<input class="form-control" (input)="selectedProduct=$any($event).target.value" />
...
```

This expression highlights a limitation of the way that data types are checked in Angular templates.

When the input element is triggered, the browser's DOM API creates an InputEvent object, and it is this object that is assigned to the \$event variable. The InputEvent.target property returns an HTMLInputElement object, which is how the DOM represents the input element that triggered the event. The HTMLInputElement.value property returns the content of the input element. Putting these types together means that reading the value of \$event.target.value will produce the contents of the input element that triggered the event.

Unfortunately, Angular assumes that the \$event variable is always assigned an Event object, which defines the features common to all events. The Event.target property returns an InputTarget object, which defines just the methods required to set up event handlers and doesn't provide access to element-specific features.

TypeScript was designed to accommodate this sort of problem using type assertions, as I explained in Chapter 3. But Angular doesn't allow the use of the as keyword in template expressions, which means that I am unable to tell the Angular and TypeScript build tools that the \$event variable contains an InputEvent object.

Angular templates do support the special \$any function, which disables type checking by treating a value as the special any type:

```
...
<input class="form-control" (input)="selectedProduct=$any($event).target.value" />
...
```

By passing \$event to the \$any function, I can read the target.value property without causing a compiler error. Care must be taken when using the \$any function because it effectively disables the compiler's type checks, which can result in errors if the specified property or methods names do not exist at runtime.

The effect of the event binding is that the selectedProduct variable is assigned the contents of the input element after each keystroke. As the user types into the input element, the text that has been entered is displayed at the top of the browser window using the string interpolation binding.

The `ngClass` binding applied to the `tr` elements sets the background color of the table rows when the `selectedProduct` property matches the name of the product they represent. And, now that the value of the `selectedProduct` property is driven by the contents of the `input` element, typing the name of a product will cause the appropriate row to be highlighted, as shown in Figure 12-5.

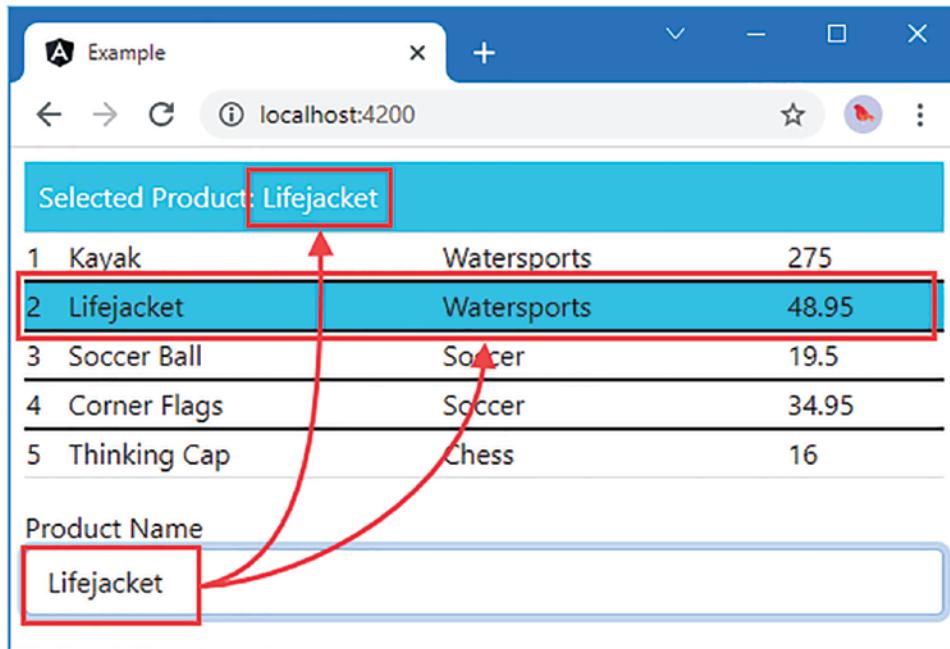


Figure 12-5. Using event data

Using different bindings to work together is at the heart of effective Angular development and makes it possible to create applications that respond immediately to user interaction and to changes in the data model.

Handling Events in the Component

Although type assertions cannot be performed in templates, they can be used in the component class, as shown in Listing 12-8, which provides a way to handle events without needing to use the `any` type.

Listing 12-8. Defining a Method in the component.ts File in the src/app Folder

```
import { Component } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";

@Component({
  selector: "app",
  templateUrl: "template.html"
})
```

```

export class ProductComponent {
  model: Model = new Model();

  getProduct(key: number): Product | undefined {
    return this.model.getProduct(key);
  }

  getProducts(): Product[] {
    return this.model.getProducts();
  }

  selectedProduct: string | undefined;

  getSelected(product: Product): boolean {
    return product.name == this.selectedProduct;
  }

  handleInputEvent(ev: Event) {
    if (ev.target instanceof HTMLInputElement) {
      this.selectedProduct = ev.target.value
    }
  }
}

```

The `handleInputEvent` method receives an `Event` object and uses the `instanceof` operator to determine if the event's `target` property returns an `HTMLInputElement`. If it does, then the `value` property is assigned to the `selectedProduct` property. Listing 12-9 updates the template to use the new method to handle events.

Listing 12-9. Handling an Event with a Method in the template.html File in the src/app Folder

```

<div class="p-2">
  <div class="bg-info text-white p-2">
    Selected Product: {{selectedProduct ?? '(None)'}}
  </div>
  <table class="table table-sm table-bordered">
    <tr *ngFor="let item of getProducts(); let i = index"
       [class.bg-info]="getSelected(item)">
      <td (mouseover)="selectedProduct=item.name">{{i + 1}}</td>
      <td>{{item.name}}</td>
      <td>{{item.category}}</td>
      <td>{{item.price}}</td>
    </tr>
  </table>
  <div class="form-group">
    <label>Product Name</label>
    <input class="form-control" (input)="handleInputEvent($event)" />
  </div>
</div>

```

The effect is the same as the previous example, but the event is handled without disabling type checking.

Using Template Reference Variables

In Chapter 11, I explained how template variables are used to pass data around within a template, such as defining a variable for the current object when using the `ngFor` directive. *Template reference variables* are a form of template variable that can be used to refer to elements *within* the template, as shown in Listing 12-10.

Listing 12-10. Using a Template Variable in the template.html File in the src/app Folder

```
<div class="p-2">
  <div class="bg-info text-white p-2">
    Selected Product: {{product.value ?? '(None)'}}
  </div>
  <table class="table table-sm table-bordered">
    <tr *ngFor="let item of getProducts(); let i = index"
       [class.bg-info]="product.value == item.name">
      <td (mouseover)="product.value = item.name ?? ''">{{i + 1}}</td>
      <td>{{item.name}}</td>
      <td>{{item.category}}</td>
      <td>{{item.price}}</td>
    </tr>
  </table>
  <div class="form-group">
    <label>Product Name</label>
    <input #product class="form-control" (input)="false" />
  </div>
</div>
```

Reference variables are defined using the `#` character, followed by the variable name. In the listing, I defined a variable called `product` like this:

```
...
<input #product class="form-control" (input)="false" />
...
```

When Angular encounters a reference variable in a template, it sets its value to the element to which it has been applied. For this example, the `product` reference variable is assigned the object that represents the `input` element in the DOM, the `HTMLInputElement` object. Reference variables can be used by other bindings in the same template. This is demonstrated by the string interpolation binding, which also uses the `product` variable, like this:

```
...
Selected Product: {{product.value ?? '(None)'}}
```

This binding displays the `value` property defined by the `HTMLInputElement` that has been assigned to the `product` variable or the string `(None)` if the `value` property returns `null` or `undefined`. Template variables can also be used to change the state of the element, as shown in this binding:

```
...
<td (mouseover)="product.value = item.name ?? ''">{{i + 1}}</td>
...
```

The event binding responds to the `mouseover` event by setting the `value` property on the `HTMLInputElement` that has been assigned to the `product` variable. The result is that moving the mouse over one of the `td` elements in the first table column will update the contents of the `input` element.

There is one awkward aspect to this example, which is the binding for the `input` event on the `input` element.

```
...
<input #product class="form-control" (input)="false" />
...
```

Angular won't update the data bindings in the template when the user edits the contents of the `input` element unless there is an event binding on that element. Setting the binding to `false` gives Angular something to evaluate just so the update process will begin and distribute the current contents of the `input` element throughout the template. This is a quirk of stretching the role of a template reference variable a little too far and isn't something you will need to do in most real projects. Most data bindings rely on variables defined by the template's component, as demonstrated in the previous section.

FILTERING KEY EVENTS

The `input` event is triggered every time the content in the `input` element is changed. This provides an immediate and responsive set of changes, but it isn't what every application requires, especially if updating the application state involves expensive operations.

The event binding has built-in support to be more selective when binding to keyboard events, which means that updates will be performed only when a specific key is pressed. Here is a binding that responds to every keystroke:

```
...
<input #product class="form-control" (keyup)="selectedProduct=product.value" />
...
```

The `keyup` event is a standard DOM event, and the result is that application is updated as the user releases each key while typing in the `input` element. I can be more specific about which key I am interested in by specifying its name as part of the event binding, like this:

```
...
<input #product class="form-control"
(keyup.enter)="selectedProduct=product.value" />
...
```

The key that the binding will respond to is specified by appending a period after the DOM event name, followed by the name of the key. This binding is for the Enter key, and the result is that the changes in the `input` element won't be pushed into the rest of the application until that key is pressed.

Using Two-Way Data Bindings

Bindings can be combined to create a two-way flow of data for a single element, allowing the HTML document to respond when the application model changes and also allowing the application to respond when the element emits an event, as shown in Listing 12-11.

Listing 12-11. Creating a Two-Way Binding in the template.html File in the src/app Folder

```
<div class="p-2">
  <div class="bg-info text-white p-2">
    Selected Product: {{ selectedProduct ?? '(None)' }}
  </div>
  <table class="table table-sm table-bordered">
    <tr *ngFor="let item of getProducts(); let i = index"
       [class.bg-info]="getSelected(item)">
      <td (mouseover)="selectedProduct=item.name">{{i + 1}}</td>
      <td>{{item.name}}</td>
      <td>{{item.category}}</td>
      <td>{{item.price}}</td>
    </tr>
  </table>
  <div class="form-group">
    <label>Product Name</label>
    <input class="form-control"
          (input)="selectedProduct=$any($event).target.value"
          [value]="selectedProduct ?? ''" />
  </div>
  <div class="form-group">
    <label>Product Name</label>
    <input class="form-control"
          (input)="selectedProduct=$any($event).target.value"
          [value]="selectedProduct ?? ''" />
  </div>
</div>
```

Each of the input elements has an event binding and a property binding. The event binding responds to the input event by updating the component's selectedProduct property. The property binding ties the value of the selectedProduct property to the element's value property.

The result is that the contents of the two input elements are synchronized, and editing one causes the other to be updated as well. And, since there are other bindings in the template that depend on the selectedProduct property, editing the contents of an input element also changes the data displayed by the string interpolation binding and changes the highlighted table row, as shown in Figure 12-6.

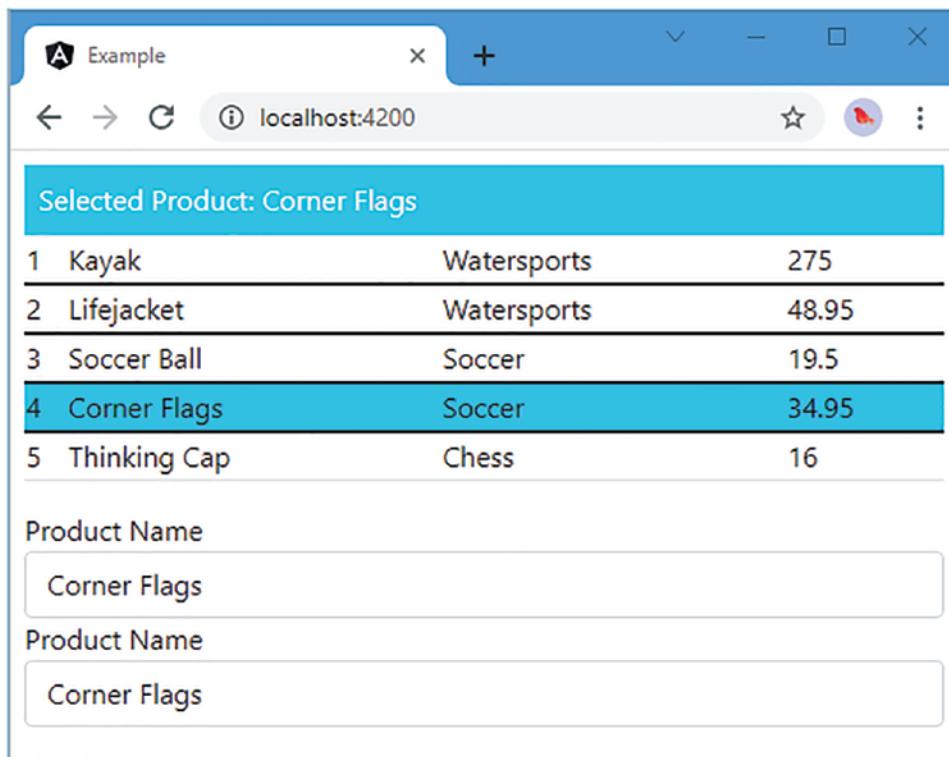


Figure 12-6. Creating a two-way data binding

This is an example that makes the most sense when you experiment with it in the browser. Enter some text into one of the `input` elements, and you will see the same text displayed in the other `input` element and in the `div` element whose content is managed by the string interpolation binding. If you enter the name of a product into one of the `input` elements, such as Kayak or Lifejacket, then you will also see the corresponding row in the table highlighted.

The event binding for the `mouseover` event still takes effect, which means as you move the mouse pointer over the first row in the table, the changes to the `selectedProduct` value will cause the `input` elements to display the product name.

Using the `ngModel` Directive

The `ngModel` directive is used to simplify two-way bindings so that you don't have to apply both an event and a property binding to the same element. Listing 12-12 shows how to replace the separate bindings with the `ngModel` directive.

Listing 12-12. Using the `ngModel` Directive in the template.html File in the src/app Folder

```
<div class="p-2">
  <div class="bg-info text-white p-2">
    Selected Product: {{ selectedProduct ?? '(None)' }}
  </div>
  <table class="table table-sm table-bordered">
```

```

<tr *ngFor="let item of getProducts(); let i = index"
    [class.bg-info]="getSelected(item)">
    <td (mouseover)="selectedProduct=item.name">{{i + 1}}</td>
    <td>{{item.name}}</td>
    <td>{{item.category}}</td>
    <td>{{item.price}}</td>
</tr>
</table>
<div class="form-group">
    <label>Product Name</label>
    <input class="form-control" [(ngModel)]="selectedProduct" />
</div>
<div class="form-group">
    <label>Product Name</label>
    <input class="form-control" [(ngModel)]="selectedProduct" />
</div>
</div>

```

Using the `ngModel` directive requires combining the syntax of the property and event bindings, as illustrated in Figure 12-7.

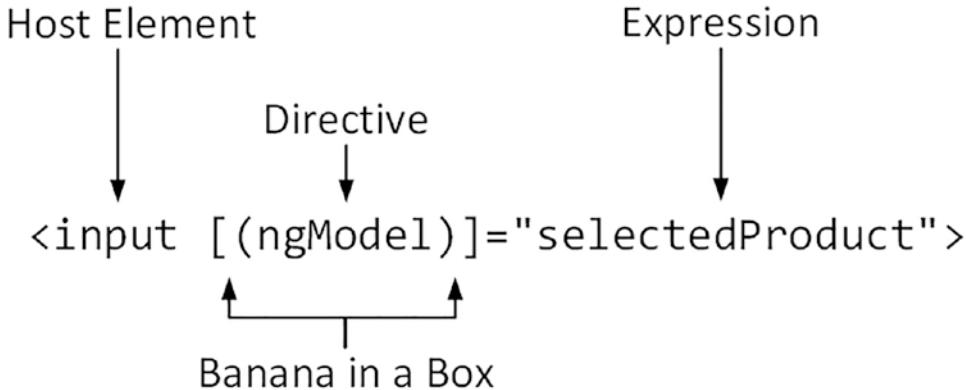


Figure 12-7. The anatomy of a two-way data binding

A combination of square and round brackets is used to denote a two-way data binding, with the round brackets placed inside the square ones: `[(and)]`. The Angular development team refers to this as the *banana-in-a-box* binding because that's what the brackets and parentheses look like when placed like this `[()]`. Well, sort of.

The target for the binding is the `ngModel` directive, which is included in Angular to simplify creating two-way data bindings on form elements, such as the `input` elements used in the example.

The expression for a two-way data binding is the name of a property, which is used to set up the individual bindings behind the scenes. When the contents of the `input` element change, the new content will be used to update the value of the `selectedProduct` property. Equally, when the value of the `selectedProduct` value changes, it will be used to update the contents of the element.

The `ngModel` directive knows the combination of events and properties that the standard HTML elements define. Behind the scenes, an event binding is applied to the `input` event, and a property binding is applied to the `value` property.

Tip You must remember to use both brackets and parentheses with the ngModel binding. If you use just parentheses—(ngModel)—then you are setting an event binding for an event called ngModel, which doesn't exist. The result is an element that won't be updated or won't update the rest of the application. You can use the ngModel directive with just square brackets—[ngModel]—and Angular will set the initial value of the element but won't listen for events, which means that changes made by the user won't be automatically reflected in the application model.

Working with Forms

Most web applications rely on forms for receiving data from users, and the two-way ngModel binding described in the previous section provides the foundation for using forms in Angular applications. In this section, I create a form that allows new products to be created and added to the application's data model and then describe some of the more advanced form features that Angular provides.

Adding a Form to the Example Application

Listing 12-13 shows some enhancements to the component that will be used when the form is created and removes some features that are no longer required.

Listing 12-13. Enhancing the Component in the component.ts File in the src/app Folder

```
import { Component } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";

@Component({
  selector: "app",
  templateUrl: "template.html"
})
export class ProductComponent {
  model: Model = new Model();

  getProduct(key: number): Product | undefined {
    return this.model.getProduct(key);
  }

  getProducts(): Product[] {
    return this.model.getProducts();
  }

  // selectedProduct: string | undefined;

  // getSelected(product: Product): boolean {
  //   return product.name == this.selectedProduct;
  // }

  // handleInputEvent(ev: Event) {
```

```

//      if (ev.target instanceof HTMLInputElement) {
//          this.selectedProduct = ev.target.value
//      }
// }

newProduct: Product = new Product();

get jsonProduct() {
    return JSON.stringify(this.newProduct);
}

addProduct(p: Product) {
    console.log("New Product: " + this.jsonProduct);
}
}

```

The listing adds a new property called `newProduct`, which will be used to store the data entered into the form by the user. There is also a `jsonProduct` property with a getter that returns a JSON representation of the `newProduct` property and that will be used in the template to show the effect of the two-way bindings. (I can't create a JSON representation of an object directly in the template because the `JSON` object is defined in the global namespace, which, as I explained in Chapter 11, cannot be accessed directly from template expressions.)

The final addition is an `addProduct` method that writes out the value of the `jsonProduct` method to the console; this will let me demonstrate some basic form-related features before adding support for updating the data model later in the chapter.

In Listing 12-14, the template content has been replaced with a series of `input` elements for each of the properties defined by the `Product` class.

Listing 12-14. Replacing the Contents of the template.html File in the src/app Folder

```

<div class="p-2">
    <div class="bg-info text-white mb-2 p-2">Model Data: {{jsonProduct}}</div>
    <div class="form-group">
        <label>Name</label>
        <input class="form-control" [(ngModel)]="newProduct.name" />
    </div>
    <div class="form-group">
        <label>Category</label>
        <input class="form-control" [(ngModel)]="newProduct.category" />
    </div>
    <div class="form-group">
        <label>Price</label>
        <input class="form-control" [(ngModel)]="newProduct.price" />
    </div>
    <button class="btn btn-primary mt-2" (click)="addProduct(newProduct)">
        Create
    </button>
</div>

```

Each `input` element is grouped with a `label` and contained in a `div` element, which is styled using the Bootstrap `form-group` class. Individual `input` elements are assigned to the Bootstrap `form-control` class to manage the layout and style.

The `ngModel` binding has been applied to each `input` element to create a two-way binding with the corresponding property on the component's `newProduct` object, like this:

```
...  
<input class="form-control" [(ngModel)]="newProduct.name" />  
...
```

There is also a `button` element, which has a binding for the `click` event that calls the component's `addProduct` method, passing in the `newProduct` value as an argument.

```
...  
<button class="btn btn-primary" (click)="addProduct(newProduct)">Create</button>  
...
```

Finally, a string interpolation binding is used to display a JSON representation of the component's `newProduct` property at the top of the template, like this:

```
...  
<div class="bg-info text-white mb-2 p-2">Model Data: {{jsonProduct}}</div>  
...
```

The overall result, illustrated in Figure 12-8, is a set of `input` elements that update the properties of a `Product` object managed by the component, which are reflected immediately in the JSON data.

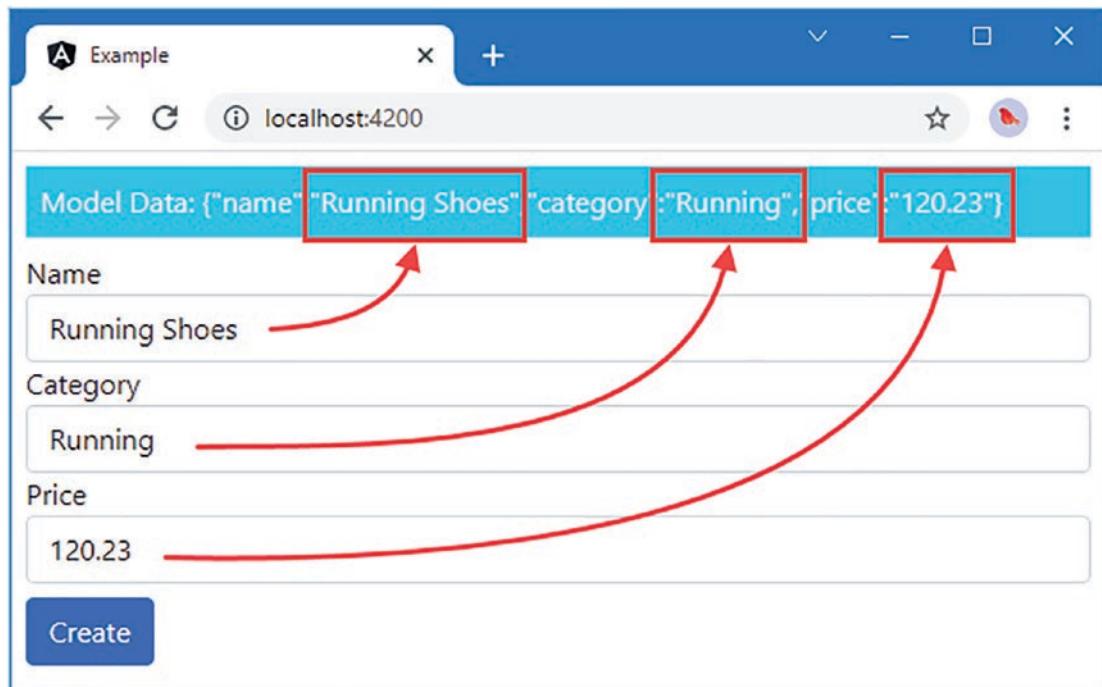


Figure 12-8. Using the form elements to create a new object in the data model

When the Create button is clicked, the JSON representation of the component's `newProduct` property is written to the browser's JavaScript console, producing a result like this:

```
New Product: {"name":"Running Shoes","category":"Running","price":120.23}
```

Adding Form Data Validation

At the moment, any data can be entered into the `input` elements in the form. Data validation is essential in web applications because users will enter a surprising range of data values, either in error or because they want to get to the end of the process as quickly as possible and enter garbage values to proceed.

Angular provides an extensible system for validating the content of form elements, based on the approach used by the HTML5 standard. Table 12-4 lists the attributes that you can add to `input` elements, each of which defines a validation rule.

Table 12-4. The Built-in Angular Validation Attributes

Attribute	Description
<code>email</code>	This attribute is used to specify a well-formatted email address.
<code>required</code>	This attribute is used to specify a value that must be provided.
<code>minlength</code>	This attribute is used to specify a minimum number of characters.
<code>maxlength</code>	This attribute is used to specify a maximum number of characters. This type of validation cannot be applied directly to form elements because it conflicts with the HTML5 attribute of the same name. It can be used with model-based forms, which are described later in the chapter.
<code>min</code>	This attribute is used to specify a minimum value.
<code>max</code>	This attribute is used to specify a maximum value.
<code>pattern</code>	This attribute is used to specify a regular expression that the value provided by the user must match.

You may be familiar with these attributes because they are part of the HTML specification, but Angular builds on these properties with some additional features. Listing 12-15 removes all but one of the `input` elements to demonstrate the process of adding validation to the form as simply as possible. (I restore the missing elements at the end of the chapter.)

Listing 12-15. Adding Form Validation in the template.html File in the src/app Folder

```
<div class="p-2">
    <div class="bg-info text-white mb-2 p-2">Model Data: {{jsonProduct}}</div>

    <form (ngSubmit)="addProduct(newProduct)">
        <div class="form-group">
            <label>Name</label>
            <input class="form-control"
                name="name"
                [(ngModel)]="newProduct.name"
```

```

required
minlength="5"
pattern="^[A-Za-z ]+$" />
</div>
<button class="btn btn-primary mt-2" type="submit">
    Create
</button>
</form>
</div>

```

Angular requires elements being validated to define the `name` attribute, which is used to identify the element in the validation system. Since this `input` element is being used to capture the value of the `Product.name` property, the `name` attribute on the element has been set to `name`.

This listing adds three of the validation attributes to the `input` element. The `required` attribute specifies that the user must provide a value, the `minlength` attribute specifies that there should be at least three characters, and the `pattern` attribute specifies that only alphabetic characters and spaces are allowed.

Finally, notice that a `form` element has been added to the template. Although you can use `input` elements independently, the Angular validation features work only when there is a `form` element present, and Angular will report an error if you add the `ngControl` directive to an element that is not contained in a `form`.

When using a `form` element, the convention is to use an event binding for a special event called `ngSubmit` like this:

```

...
<form (ngSubmit)="addProduct(newProduct)">
...

```

The `ngSubmit` binding handles the `form` element's `submit` event. You can achieve the same effect by binding to the `click` event on individual button elements within the `form` if you prefer.

Styling Elements Using Validation Classes

Once you have saved the template changes in Listing 12-15 and the browser has reloaded the HTML, right-click the `input` element in the browser window and select Inspect or Inspect Element from the pop-up window. The browser will display the HTML representation of the element in the Developer Tools window, and you will see that the `input` element has been added to three classes, like this:

```

...
<input name="name" required="" minlength="5" pattern="^[A-Za-z ]+$"
       class="form-control ng-pristine ng-invalid ng-touched" ng-reflect-required=""
       ng-reflect-minlength="5" ng-reflect-pattern="^[A-Za-z ]+$" ng-reflect-name="name">
...

```

The classes to which an `input` element is assigned provide details of its validation state. There are three pairs of validation classes, which are described in Table 12-5. Elements will always be members of one class from each pair, for a total of three classes. The same classes are applied to the `form` element to show the overall validation status of all the elements it contains. As the status of the `input` element changes, the `ngControl` directive switches the classes automatically for both the individual elements and the `form` element.

Table 12-5. The Angular Form Validation Classes

Name	Description
ng-untouched	An element is assigned to the ng-untouched class if it has not been visited by the user, which is typically done by tabbing through the form fields. Once the user has visited an element, it is added to the ng-touched class.
ng-pristine	An element is assigned to the ng-pristine class if its contents have not been changed by the user and to the ng-dirty class otherwise. Once the contents have been edited, an element remains in the ng-dirty class, even if the user then returns to the previous contents.
ng-valid	An element is assigned to the ng-valid class if its contents meet the criteria defined by the validation rules that have been applied to it and to the ng-invalid class otherwise.
ng-pending	Elements are assigned to the ng-pending class when their contents are being validated asynchronously. See Chapters 21 and 22 for details.

These classes can be used to style form elements to provide the user with validation feedback. Add the styles shown in Listing 12-16 to the styles.css file in the src folder.

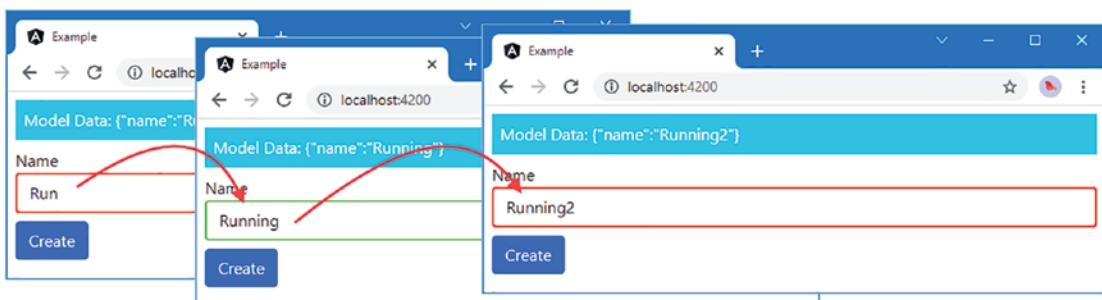
Listing 12-16. Defining Validation Feedback Styles in the styles.css File in the src/app Folder

```
html, body { height: 100%; }
body { margin: 0; font-family: Roboto, "Helvetica Neue", sans-serif; }

input.ng-dirty.ng-invalid { border: 2px solid #ff0000; }
input.ng-dirty.ng-valid { border: 2px solid #6bc502; }
```

These styles set green and red borders for input elements whose content has been edited and is valid (and so belong to both the ng-dirty and ng-valid classes) and whose content is invalid (and so belong to the ng-dirty and ng-invalid classes). Using the ng-dirty class means that the appearance of the elements won't be changed until after the user has entered some content.

Angular validates the contents and changes the class memberships of the input elements after each keystroke or focus change. The browser detects the changes to the elements and applies the styles dynamically, which provides users with validation feedback as they enter data into the form, as shown in Figure 12-9.

**Figure 12-9.** Providing validation feedback

As I start to type, the `input` element is shown as invalid because there are not enough characters to satisfy the `minlength` attribute. Once there are five characters, the border is green, indicating that the data is valid. When I type the 2 character, the border turns red again because the `pattern` attribute is set to allow only letters and spaces.

Tip If you look at the JSON data at the top of the page in Figure 12-9, you will see that the data bindings are still being updated, even when the data values are not valid. Validation runs alongside data bindings, and you should not act on form data without checking that the overall form is valid, as described in the “Validating the Entire Form” section.

Displaying Field-Level Validation Messages

Using colors to provide validation feedback tells the user that something is wrong but doesn’t provide any indication of what the user should do about it. The `ngModel` directive provides access to the validation status of the elements it is applied to, which can be used to display guidance to the user. Listing 12-17 adds validation messages for each of the attributes applied to the `input` element using the support provided by the `ngModel` directive.

Listing 12-17. Adding Validation Messages in the template.html File in the src/app Folder

```
<div class="p-2">
    <div class="bg-info text-white mb-2 p-2">Model Data: {{jsonProduct}}</div>

    <form (ngSubmit)="addProduct(newProduct)">
        <div class="form-group">
            <label>Name</label>
            <input class="form-control"
                name="name"
                [(ngModel)]="newProduct.name"
                #name="ngModel"
                required
                minlength="5"
                pattern="^[A-Za-z ]+$" />
            <ul class="text-danger list-unstyled mt-1">
                *ngIf="name.dirty && name.invalid">
                <li *ngIf="name.errors?.['required']">
                    You must enter a product name
                </li>
                <li *ngIf="name.errors?.['pattern']">
                    Product names can only contain letters and spaces
                </li>
                <li *ngIf="name.errors?.['minlength']">
                    Product names must be at least
                    {{ name.errors?.['minlength'].requiredLength }} characters
                </li>
            </ul>
        </div>
        <button class="btn btn-primary mt-2" type="submit">
```

```

    Create
    </button>
  </form>
</div>
```

To get validation working, I have to create a template reference variable to access the validation state in expressions, which I do like this:

```

...
<input class="form-control" name="name" [(ngModel)]="newProduct.name"
  #name="ngModel" required minlength="5" pattern="^[A-Za-z ]+$/>
...
```

I create a template reference variable called `name` and set its value to `ngModel`. This use of an `ngModel` value is a little confusing: it is a feature provided by the `ngModel` directive to give access to the validation status. This will make more sense once you have read Chapters 13 and 16, in which I explain how to create custom directives and you see how they provide access to their features. For this chapter, it is enough to know that to display validation messages, you need to create a template reference variable and assign it `ngModel` to access the validation data for the `input` element. The object that is assigned to the template reference variable defines the properties that are described in Table 12-6. All of the properties described in the table are nullable.

Table 12-6. The Validation Object Properties

Name	Description
<code>path</code>	This property returns the name of the element.
<code>valid</code>	This property returns <code>true</code> if the element's contents are valid and <code>false</code> otherwise.
<code>invalid</code>	This property returns <code>true</code> if the element's contents are invalid and <code>false</code> otherwise.
<code>pristine</code>	This property returns <code>true</code> if the element's contents have not been changed.
<code>dirty</code>	This property returns <code>true</code> if the element's contents have been changed.
<code>touched</code>	This property returns <code>true</code> if the user has visited the element.
<code>untouched</code>	This property returns <code>true</code> if the user has not visited the element.
<code>errors</code>	This property returns a <code>ValidationErrors</code> object whose properties correspond to each attribute for which there is a validation error.
<code>value</code>	This property returns the <code>value</code> of the element, which is used when defining custom validation rules, as described in the “Creating Custom Form Validators” section.

Listing 12-17 displays the validation messages in a list. The list should be shown only if there is at least one validation error, so I applied the `ngIf` directive to the `ul` element, with an expression that uses the `dirty` and `invalid` properties, like this:

```

...
<ul class="text-danger list-unstyled mt-1" *ngIf="name.dirty && name.invalid">
  ...

```

Within the `ul` element, there is an `li` element that corresponds to each validation error that can occur. Each `li` element has an `ngIf` directive that uses the `errors` property described in Table 12-6, like this:

```
...
<li *ngIf="name.errors?.['required']">
  You must enter a product name
</li>
...
```

The `errors.[required]` property will be defined only if the element's contents have failed the required validation check, which ties the visibility of the `li` element to the outcome of that validation check.

Each property defined by the `errors` object returns an object whose properties provide details of why the content has failed the validation check for its attribute, which can be used to make the validation messages more helpful to the user. Table 12-7 describes the error properties provided for each attribute.

Table 12-7. The Angular Form Validation Error Description Properties

Name	Description
<code>email</code>	This property returns <code>true</code> if the <code>email</code> attribute has been applied to the <code>input</code> element. This is not especially useful because this can be deduced from the fact that the property exists.
<code>required</code>	This property returns <code>true</code> if the <code>required</code> attribute has been applied to the <code>input</code> element. This is not especially useful because this can be deduced from the fact that the property exists.
<code>minlength.requiredLength</code>	This property returns the number of characters required to satisfy the <code>minlength</code> attribute.
<code>minlength.actualLength</code>	This property returns the number of characters entered by the user.
<code>maxlength.requiredLength</code>	This property returns the number of characters required to satisfy the <code>maxlength</code> attribute.
<code>maxlength.actualLength</code>	This property returns the number of characters entered by the user.
<code>min.actual</code>	This property returns the value entered by the user.
<code>min.min</code>	This property returns the minimum value required to satisfy the <code>min</code> attribute.
<code>max.actual</code>	This property returns the value entered by the user.
<code>max.max</code>	This property returns the minimum value required to satisfy the <code>max</code> attribute.
<code>pattern.requiredPattern</code>	This property returns the regular expression that has been specified using the <code>pattern</code> attribute.
<code>pattern.actualValue</code>	This property returns the contents of the element.

These properties are not displayed directly to the user, who is unlikely to understand an error message that includes a regular expression, although they can be useful during development to figure out validation problems. The exception is the `minlength.requiredLength` property, which can be useful for avoiding the duplication of the value assigned to the `minlength` attribute on the element, like this:

```
...
<li *ngIf="name.errors?.['minlength']">
  Product names must be at least
  {{ name.errors?.['minlength'].requiredLength }} characters
</li>
...

```

The overall result is a set of validation messages that are shown as soon as the user starts editing the `input` element and that change to reflect each new keystroke, as illustrated in Figure 12-10.

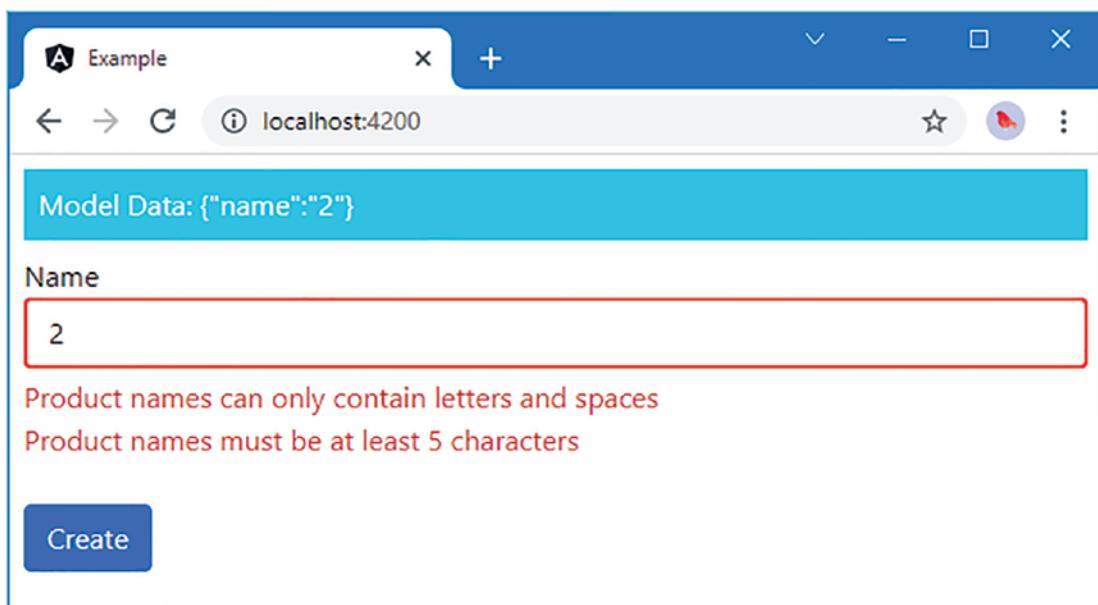


Figure 12-10. Displaying validation messages

Using the Component to Display Validation Messages

Including separate elements for all possible validation errors quickly becomes verbose in complex forms. A better approach is to add logic to the component to prepare the validation messages in a method, which can then be displayed to the user through the `ngFor` directive in the template. Listing 12-18 shows the addition of a component method that accepts the validation state for an `input` element and produces an array of validation messages.

Listing 12-18. Generating Validation Messages in the component.ts File in the src/app Folder

```

import { Component } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";
import { NgModel, ValidationErrors } from "@angular/forms";

@Component({
  selector: "app",
  templateUrl: "template.html"
})
export class ProductComponent {
  model: Model = new Model();

  getProduct(key: number): Product | undefined {
    return this.model.getProduct(key);
  }

  getProducts(): Product[] {
    return this.model.getProducts();
  }

  newProduct: Product = new Product();

  get jsonProduct() {
    return JSON.stringify(this.newProduct);
  }

  addProduct(p: Product) {
    console.log("New Product: " + this.jsonProduct);
  }

  getMessages(errs : ValidationErrors | null, name: string) : string[] {
    let messages: string[] = [];
    for (let errorName in errs) {
      switch (errorName) {
        case "required":
          messages.push(`You must enter a ${name}`);
          break;
        case "minlength":
          messages.push(`A ${name} must be at least
            ${errs['minlength'].requiredLength}
            characters`);
          break;
        case "pattern":
          messages.push(`The ${name} contains
            illegal characters`);
          break;
      }
    }
    return messages;
  }
}

```

```

getValidationMessages(state: NgModel, thingName?: string) {
    let thing: string = state.path?.[0] ?? thingName;
    return this.getMessages(state.errors, thing)
}
}

```

The `getValidationMessages` and `getMessages` methods use the properties described in Table 12-6 to produce validation messages for each error, returning them in a string array. To make this code as widely applicable as possible, the method accepts a value that describes the data item that an `input` element is intended to collect from the user, which is then used to generate error messages, like this:

```

...
messages.push(`You must enter a ${name}`);
...

```

This is an example of the JavaScript string interpolation feature, which allows strings to be defined like templates, without having to use the `+` operator to include data values. Note that the template string is denoted with backtick characters (the ``` character and not the regular JavaScript `'` character). The `getValidationMessages` method defaults to using the `path` property as the descriptive string if an argument isn't received when the method is invoked, like this:

```

...
let thing: string = state.path?.[0] ?? thingName;
...

```

Listing 12-19 shows how the `getValidationMessages` can be used in the template to generate validation error messages for the user without needing to define separate elements and bindings for each one.

Listing 12-19. Getting Validation Messages in the template.html File in the src/app Folder

```

<div class="p-2">
    <div class="bg-info text-white mb-2 p-2">Model Data: {{jsonProduct}}</div>

    <form (ngSubmit)="addProduct(newProduct)">
        <div class="form-group">
            <label>Name</label>
            <input class="form-control"
                name="name"
                [(ngModel)]="newProduct.name"
                #name="ngModel"
                required
                minlength="5"
                pattern="^[\w\W]{5,}$" />
            <ul class="text-danger list-unstyled mt-1"
                *ngIf="name.dirty && name.invalid">
                <li *ngFor="let error of getValidationMessages(name)">
                    {{error}}
                </li>
            </ul>
        </div>
    </div>

```

```

<button class="btn btn-primary mt-2" type="submit">
    Create
</button>
</form>
</div>

```

There is no visual change, but the same method can be used to produce validation messages for multiple elements, which results in a simpler template that is easier to read and maintain.

Validating the Entire Form

Displaying validation error messages for individual fields is useful because it helps emphasize where problems need to be fixed. But it can also be useful to validate the entire form. Care must be taken not to overwhelm the user with error messages until they try to submit the form, at which point a summary of any problems can be useful. In preparation, Listing 12-20 adds two new members to the component.

Listing 12-20. Enhancing the Component in the component.ts File in the src/app Folder

```

import { Component } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";
import { NgModel, ValidationErrors, NgForm } from "@angular/forms";

@Component({
    selector: "app",
    templateUrl: "template.html"
})
export class ProductComponent {
    model: Model = new Model();

    // ...other methods omitted for brevity...

    formSubmitted: boolean = false;

    submitForm(form: NgForm) {
        this.formSubmitted = true;
        if (form.valid) {
            this.addProduct(this.newProduct);
            this.newProduct = new Product();
            form.resetForm();
            this.formSubmitted = false;
        }
    }
}

```

The `formSubmitted` property will be used to indicate whether the form has been submitted and will be used to prevent validation of the entire form until the user has tried to submit.

The `submitForm` method will be invoked when the user submits the form and receives an `NgForm` object as its argument. This object represents the form and defines the set of validation properties; these properties are used to describe the overall validation status of the form so that, for example, the `invalid` property will be true if there are validation errors on any of the elements contained by the form. In addition to the

validation property, `NgForm` provides the `resetForm` method, which resets the validation status of the form and returns it to its original and pristine state.

The effect is that the whole form will be validated when the user performs a submit, and if there are no validation errors, a new object will be added to the data model before the form is reset so that it can be used again. Listing 12-21 shows the changes required to the template to take advantage of these new features and implement form-wide validation.

Listing 12-21. Performing Form-Wide Validation in the template.html File in the src/app Folder

```
<div class="p-2">
  <form #form="ngForm" (ngSubmit)="submitForm(form)">

    <div class="bg-danger text-white p-2 mb-2"
        *ngIf="formSubmitted && form.invalid">
      There are problems with the form
    </div>

    <div class="form-group">
      <label>Name</label>
      <input class="form-control"
            name="name"
            [(ngModel)]="newProduct.name"
            #name="ngModel"
            required
            minlength="5"
            pattern="^[A-Za-z ]+$" />
      <ul class="text-danger list-unstyled mt-1"
          *ngIf="(formSubmitted || name.dirty) && name.invalid">
        <li *ngFor="let error of getValidationMessages(name)">
          {{error}}
        </li>
      </ul>
    </div>
    <button class="btn btn-primary mt-2" type="submit">
      Create
    </button>
  </form>
</div>
```

The `form` element now defines a reference variable called `form`, which has been assigned to `ngForm`. This is how the `ngForm` directive provides access to its functionality, through a process that I describe in Chapter 13. For now, however, it is important to know that the validation information for the entire form can be accessed through the `form` reference variable.

The listing also changes the expression for the `ngSubmit` binding so that it calls the `submitForm` method defined by the controller, passing in the template variable, like this:

```
...
<form ngForm="productForm" #form="ngForm" (ngSubmit)="submitForm(form)">
  ...

```

It is this object that is received as the argument of the `submitForm` method and that is used to check the validation status of the form and to reset the form so that it can be used again.

Listing 12-21 also adds a div element that uses the `formSubmitted` property from the component along with the `valid` property (provided by the `form` template variable) to show a warning message when the form contains invalid data, but only after the form has been submitted.

In addition, the `ngIf` binding has been updated to display the field-level validation messages so that they will be shown when the form has been submitted, even if the element itself hasn't been edited. The result is a validation summary that is shown only when the user submits the form with invalid data, as illustrated by Figure 12-11.

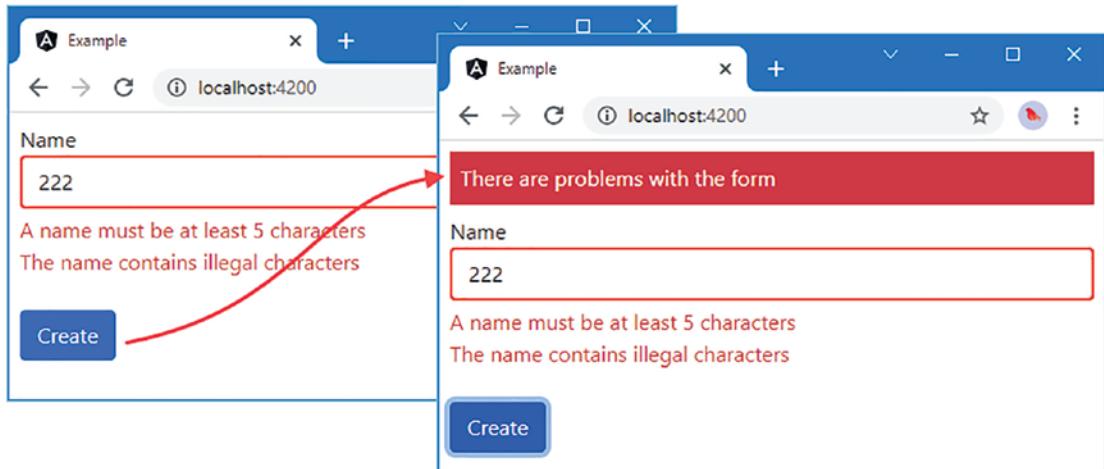


Figure 12-11. Displaying a validation summary message

Displaying Summary Validation Messages

In a complex form, it can be helpful to provide the user with a summary of all the validation errors that have to be resolved. The `NgForm` object assigned to the `form` template reference variable provides access to the individual elements through a property named `controls`. This property returns an object that has properties for each of the individual elements in the form. For example, there is a `name` property that represents the `input` element in the example, which is assigned an object that represents that element and defines the same validation properties that are available for individual elements. In Listing 12-22, I have added a method to the component that receives the object assigned to the form element's template reference variables and uses its `controls` property to generate a list of error messages for the entire form.

Listing 12-22. Generating Form-Wide Validation Messages in the component.ts File in the src/app Folder

```
import { Component } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";
import { NgModel, ValidationErrors, NgForm } from "@angular/forms";

@Component({
  selector: "app",
  templateUrl: "template.html"
})
```

```

export class ProductComponent {
  model: Model = new Model();

  // ...other methods omitted for brevity...

  getFormValidationMessages(form: NgForm): string[] {
    let messages: string[] = [];
    Object.keys(form.controls).forEach(k => {
      this.getMessages(form.controls[k].errors, k)
        .forEach(m => messages.push(m));
    });
    return messages;
  }
}

```

The `getFormValidationMessages` method builds its list of messages by calling the `getMessages` method for each control in the form. The `Object.keys` method creates an array from the properties defined by the object returned by the `controls` property, which is enumerated using the `forEach` method.

In Listing 12-23, I have used this method to include the individual messages at the top of the form, which will be visible once the user clicks the Create button.

Listing 12-23. Displaying Form-Wide Validation Messages in the template.html File in the src/app Folder

```

<div class="p-2">
  <form #form="ngForm" (ngSubmit)="submitForm(form)">

    <div class="bg-danger text-white p-2 mb-2"
         *ngIf="formSubmitted && form.invalid">
      There are problems with the form
      <ul>
        <li *ngFor="let error of getFormValidationMessages(form)">
          {{error}}
        </li>
      </ul>
    </div>

    <div class="form-group">
      <label>Name</label>
      <input class="form-control"
            name="name"
            [(ngModel)]="newProduct.name"
            #name="ngModel"
            required
            minlength="5"
            pattern="^[A-Za-z ]+$" />
      <ul class="text-danger list-unstyled mt-1"
           *ngIf="(formSubmitted || name.dirty) && name.invalid">
        <li *ngFor="let error of getValidationMessages(name)">
          {{error}}
        </li>
      </ul>
    </div>

```

```
<button class="btn btn-primary mt-2" type="submit">  
    Create  
</button>  
</form>  
</div>
```

The result is that validation messages are displayed alongside the input element and collected at the top of the form once it has been submitted, as shown in Figure 12-12.

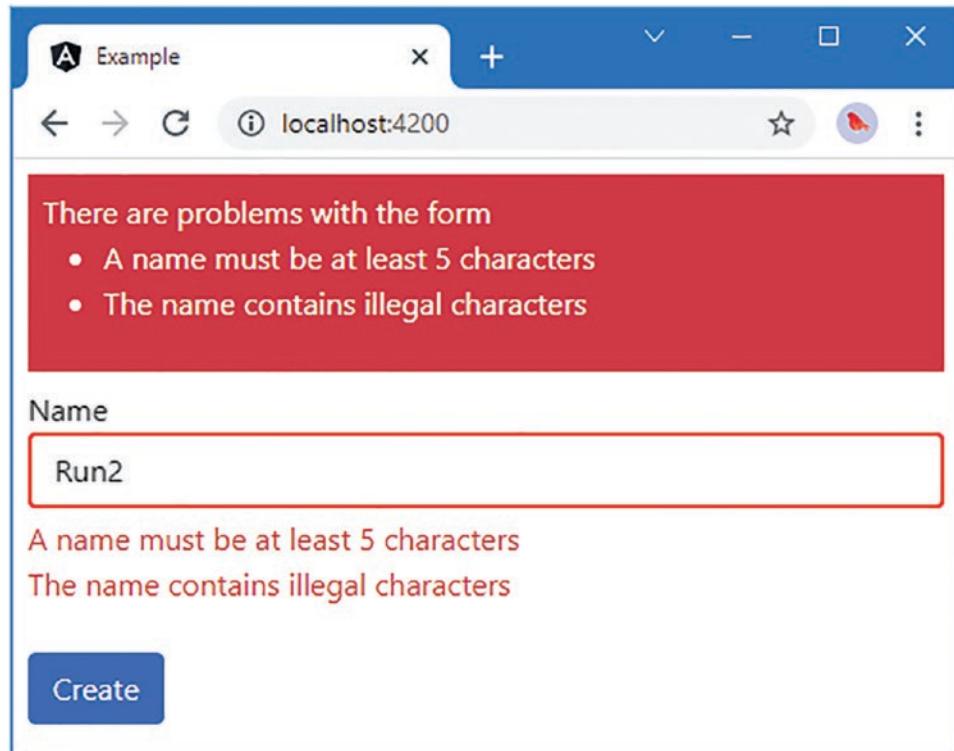


Figure 12-12. Displaying an overall validation summary

Disabling the Submit Button

The next step is to disable the button once the user has submitted the form, preventing the user from clicking it again until all the validation errors have been resolved. This is a commonly used technique even though it has little bearing on the example application, which won't accept the data from the form while it contains invalid values but provides useful reinforcement to the user that they cannot proceed until the validation problems have been resolved. In Listing 12-24, I have used the property binding on the button element.

Listing 12-24. Disabling the Button in the template.html File in the src/app Folder

```
<div class="p-2">
  <form #form="ngForm" (ngSubmit)="submitForm(form)">

    <div class="bg-danger text-white p-2 mb-2"
        *ngIf="formSubmitted && form.invalid">
      There are problems with the form
      <ul>
        <li *ngFor="let error of getFormValidationMessages(form)">
          {{error}}
        </li>
      </ul>
    </div>

    <div class="form-group">
      <label>Name</label>
      <input class="form-control"
            name="name"
            [(ngModel)]="newProduct.name"
            #name="ngModel"
            required
            minlength="5"
            pattern="^[A-Za-z ]+$" />
      <ul class="text-danger list-unstyled mt-1"
          *ngIf="(formSubmitted || name.dirty) && name.invalid">
        <li *ngFor="let error of getValidationMessages(name)">
          {{error}}
        </li>
      </ul>
    </div>
    <button class="btn btn-primary mt-2" type="submit"
           [disabled]="formSubmitted && form.invalid"
           [class.btn-secondary]="formSubmitted && form.invalid">
      Create
    </button>
  </form>
</div>
```

For extra emphasis, I used the class binding to add the button element to the `btn-secondary` class when the form has been submitted and has invalid data. This class applies a Bootstrap CSS style, as shown in Figure 12-13.

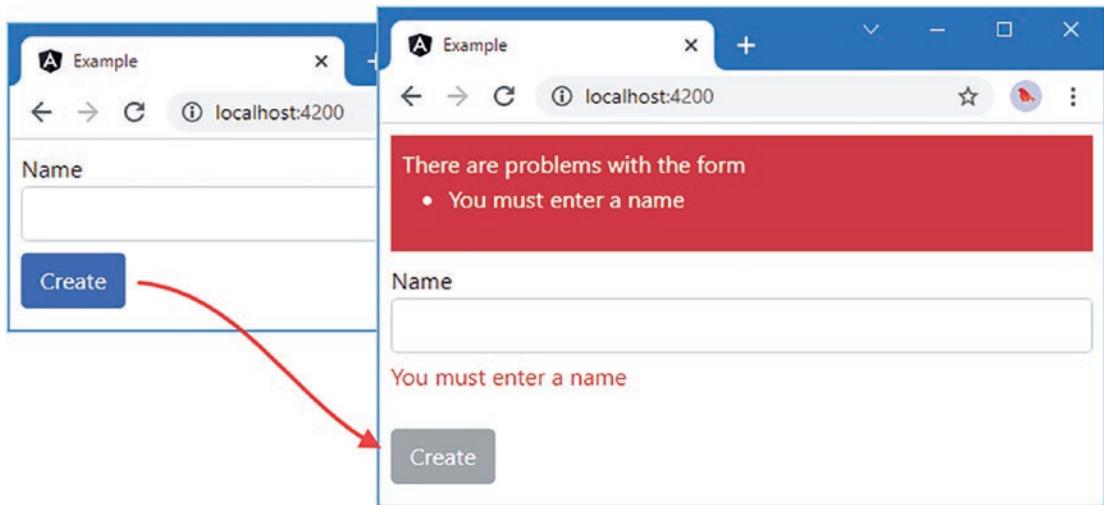


Figure 12-13. Disabling the submit button

Completing the Form

Now that the validation features are done, I can complete the form. Listing 12-25 restores the input elements for the category and price fields, which I removed earlier in the chapter. I also removed the validation messages for the name element so that only the form-wide error messages are displayed.

Listing 12-25. Adding Form Elements in the template.html File in the src/app Folder

```
<div class="p-2">
    <form #form="ngForm" (ngSubmit)="submitForm(form)">

        <div class="bg-danger text-white p-2 mb-2"
            *ngIf="formSubmitted && form.invalid">
            There are problems with the form
            <ul>
                <li *ngFor="let error of getFormValidationMessages(form)">
                    {{error}}
                </li>
            </ul>
        </div>

        <div class="form-group">
            <label>Name</label>
            <input class="form-control"
                name="name"
                [(ngModel)]="newProduct.name"
                #name="ngModel"
                required
                minlength="5"
                pattern="^[\w\W]{5,}$" />
        </div>
    </form>
</div>
```

```

<div class="form-group">
  <label>Category</label>
  <input class="form-control" name="category"
    [(ngModel)]="newProduct.category" required />
</div>

<div class="form-group">
  <label>Price</label>
  <input class="form-control" name="price"
    [(ngModel)]="newProduct.price" required type="number"/>
</div>

<button class="btn btn-primary mt-2" type="submit"
  [disabled]="formSubmitted && form.invalid"
  [class.btn-secondary]="formSubmitted && form.invalid">
  Create
</button>
</form>
</div>

```

The final change is to adjust the selectors for the CSS styles that indicate valid and invalid input elements, as shown in Listing 12-26.

Listing 12-26. Adjusting the CSS Selectors in the styles.css File in the src Folder

```

html, body { height: 100%; }
body { margin: 0; font-family: Roboto, "Helvetica Neue", sans-serif; }

form.ng-submitted input.ng-invalid { border: 2px solid #ff0000; }
form.ng-submitted input.ng-valid { border: 2px solid #6bc502; }

```

In addition to the classes described in Table 12-5, Angular adds form elements to the ng-submitted class when they have been submitted. This allows me to select elements that are invalid once the form has been submitted, regardless of whether the user has edited the elements.

Save the changes and click the Create button; you will see the validation messages and CSS styles shown in Figure 12-14. As you address each validation error, the input elements will turn green, and you will be able to submit the form when there are no validation errors remaining.

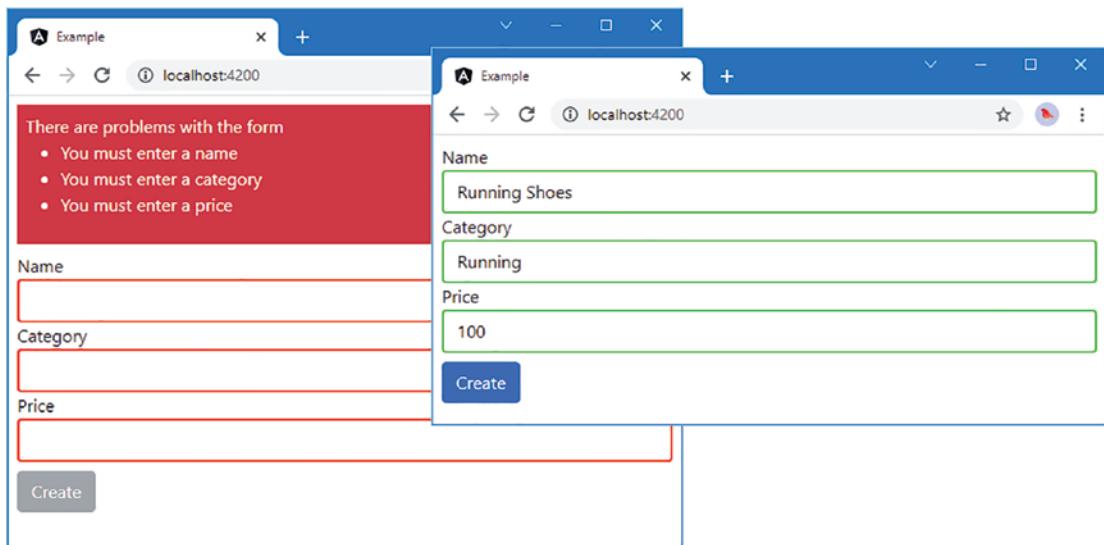


Figure 12-14. Finishing the form

Summary

In this chapter, I introduced the way that Angular supports user interaction using events and forms. I explained how to create event bindings, how to create two-way bindings, and how they can be simplified using the `ngModel` directive. I also described the support that Angular provides for managing and validating HTML forms. In the next chapter, I explain how to create custom directives.

CHAPTER 13



Creating Attribute Directives

In this chapter, I describe how custom directives can be used to supplement the functionality provided by the built-in ones of Angular. The focus of this chapter is *attribute directives*, which are the simplest type that can be created and that change the appearance or behavior of a single element. In Chapter 14, I explain how to create *structural directives*, which are used to change the layout of the HTML document. Components are also a type of directive, and I explain how they work in Chapter 15.

Throughout these chapters, I describe how custom directives work by re-creating the features provided by some of the built-in directives. This isn't something you would typically do in a real project, but it provides a useful baseline against which the process can be explained. Table 13-1 puts attribute directives into context.

Table 13-1. Putting Attribute Directives in Context

Question	Answer
What are they?	Attribute directives are classes that can modify the behavior or appearance of the element they are applied to. The style and class bindings described in Chapter 10 are examples of attribute directives.
Why are they useful?	The built-in directives cover the most common tasks required in web application development but don't deal with every situation. Custom directives allow application-specific features to be defined.
How are they used?	Attribute directives are classes to which the <code>@Directive</code> decorator has been applied. They are enabled in the <code>directives</code> property of the component responsible for a template and applied using a CSS selector.
Are there any pitfalls or limitations?	The main pitfall when creating a custom directive is the temptation to write code to perform tasks that can be better handled using directive features such as input and output properties and host element bindings.
Are there any alternatives?	Angular supports two other types of directive—structural directives and components—that may be more suitable for a given task. You can sometimes combine the built-in directives to create a specific effect if you prefer to avoid writing custom code, although the result can be brittle and lead to complex HTML that is hard to read and maintain.

Table 13-2 summarizes the chapter.

Table 13-2. Chapter Summary

Problem	Solution	Listing
Creating an attribute directive	Apply @Directive to a class	1-5
Accessing host element attribute values	Apply the @Attribute decorator to a constructor parameter	6-9
Creating a data-bound input property	Apply the @Input decorator to a class property	10-11
Receiving a notification when a data-bound input property value changes	Implement the ngOnChanges method	12
Defining an event	Apply the @Output decorator	13, 14
Creating a property binding or event binding on the host element	Apply the @HostBinding or @HostListener decorator	15-19
Exporting a directive's functionality for use in the template	Use the exportAs property of the @Directive decorator	20, 21

Preparing the Example Project

As I have been doing throughout this part of the book, I will continue using the example project from the previous chapter. To prepare for this chapter, I have redefined the form so that it updates the component's newProduct property rather than the model-based form used in Chapter 12, as shown in Listing 13-1.

Tip You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/Apress/pro-angular-5ed>. See Chapter 1 for how to get help if you have problems running the examples.

Listing 13-1. Replacing the Contents of the template.html File in the src/app Folder

```
<div class="row p-2">
  <div class="col-6">
    <form class="m-2" (ngSubmit)="submitForm()">
      <div class="form-group">
        <label>Name</label>
        <input class="form-control" name="name" [(ngModel)]="newProduct.name" />
      </div>
      <div class="form-group">
        <label>Category</label>
        <input class="form-control" name="category"
          [(ngModel)]="newProduct.category" />
      </div>
```

```

<div class="form-group">
  <label>Price</label>
  <input class="form-control" name="price" [(ngModel)]="newProduct.price" />
</div>
<button class="btn btn-primary" type="submit">Create</button>
</form>
</div>

<div class="col">
  <table class="table table-sm table-bordered table-striped">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th></tr>
    <tr *ngFor="let item of getProducts(); let i = index">
      <td>{{i + 1}}</td>
      <td>{{item.name}}</td>
      <td>{{item.category}}</td>
      <td>{{item.price}}</td>
    </tr>
  </table>
</div>
</div>

```

This listing uses the Bootstrap grid layout to position the form and the table side by side. Listing 13-2 simplifies the component and updates the component's addProduct method so that it adds a new object to the data model.

Listing 13-2. Replacing the Contents of the component.ts File in the src/app Folder

```

import { Component } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";

@Component({
  selector: "app",
  templateUrl: "template.html"
})
export class ProductComponent {
  model: Model = new Model();

  getProduct(key: number): Product | undefined {
    return this.model.getProduct(key);
  }

  getProducts(): Product[] {
    return this.model.getProducts();
  }

  newProduct: Product = new Product();

  addProduct(p: Product) {
    this.model.saveProduct(p);
  }
}

```

```

    submitForm() {
        this.addProduct(this.newProduct);
    }
}

```

To start the application, navigate to the example project folder and run the following command:

```
ng serve
```

Open a new browser window and navigate to <http://localhost:4200> to see the form in Figure 13-1. A new item will be added to the data model and displayed in the table when you submit the form. When the form is submitted, the CSS validation styles will be displayed because Angular adds form elements to the validation classes, even when no validation is performed.

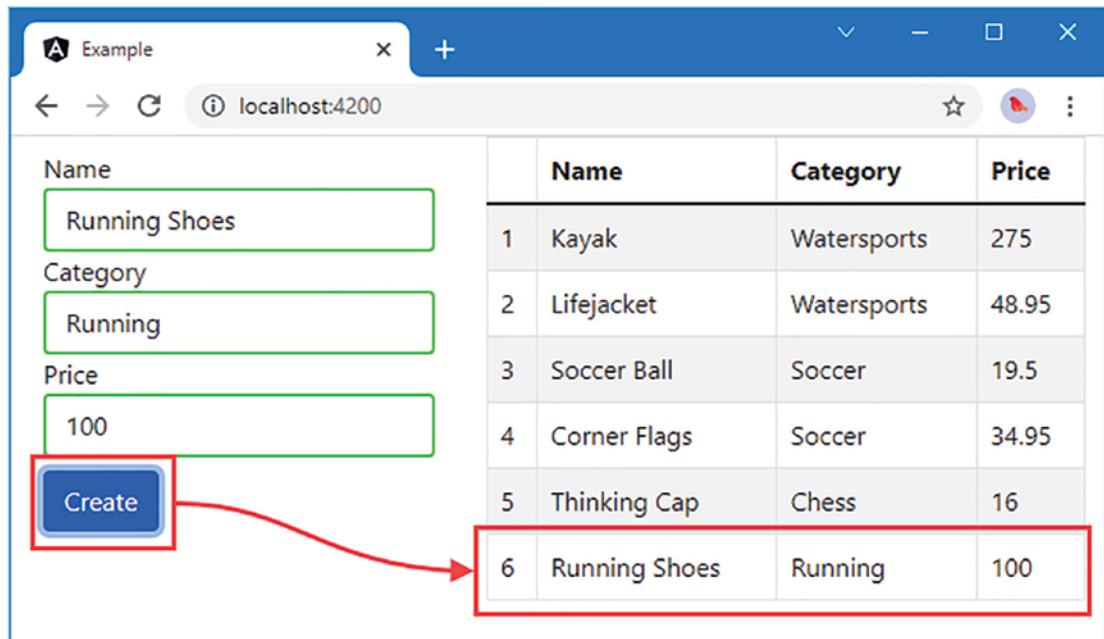


Figure 13-1. Running the example application

Creating a Simple Attribute Directive

The best place to start is to jump in and create a directive to see how they work. I added a file called `attr.directive.ts` to the `src/app` folder with the code shown in Listing 13-3. The name of the file indicates that it contains a directive. I set the first part of the filename to `attr` to indicate that this is an example of an attribute directive.

Listing 13-3. The Contents of the attr.directive.ts File in the src/app Folder

```
import { Directive, ElementRef } from "@angular/core";

@Directive({
  selector: "[pa-attr]",
})
export class PaAttrDirective {

  constructor(element: ElementRef) {
    element.nativeElement.classList.add("table-success", "fw-bold");
  }
}
```

Directives are classes to which the `@Directive` decorator has been applied. The decorator requires the `selector` property, which is used to specify how the directive is applied to elements, expressed using a standard CSS style selector. The selector I used is `[pa-attr]`, which will match any element that has an attribute called `pa-attr`, regardless of the element type or the value assigned to the attribute.

Custom directives are given a distinctive prefix so they can be easily recognized. The prefix can be anything meaningful to your application. I have chosen the prefix `Pa` for my directive, reflecting the title of this book, and this prefix is used in the attribute specified by the `selector` decorator property and the name of the attribute class. The case of the prefix is changed to reflect its use so that an initial lowercase character is used for the selector attribute name (`pa-attr`) and an initial uppercase character is used in the name of the directive class (`PaAttrDirective`).

Note The prefix `Ng/ng` is reserved for use for built-in Angular features and should not be used.

The directive constructor defines a single `ElementRef` parameter, which Angular provides when it creates a new instance of the directive and which represents the host element. The `ElementRef` class defines a single property, `nativeElement`, which returns the object used by the browser to represent the element in the Domain Object Model. This object provides access to the methods and properties that manipulate the element and its contents, including the `classList` property, which can be used to manage the class membership of the element, like this:

```
...
element.nativeElement.classList.add("table-success", "fw-bold");
...
```

To summarize, the `PaAttrDirective` class is a directive that is applied to elements that have a `pa-attr` attribute and adds those elements to the `table-success` and `fw-bold` classes, which the Bootstrap CSS library uses to assign background color and font weight to elements.

Applying a Custom Directive

There are two steps to apply a custom directive. The first is to update the template so that there are one or more elements that match the `selector` that the directive uses. In the case of the example directive, this means adding the `pa-attr` attribute to an element, as shown in Listing 13-4.

Listing 13-4. Adding a Directive Attribute in the template.html File in the src/app Folder

```
...
<div class="col">
  <table class="table table-bordered table-striped">
    <thead>
      <tr><th></th><th>Name</th><th>Category</th><th>Price</th></tr>
    </thead>
    <tbody>
      <tr *ngFor="let item of getProducts(); let i = index" pa-attr>
        <td>{{i + 1}}</td>
        <td>{{item.name}}</td>
        <td>{{item.category}}</td>
        <td>{{item.price}}</td>
      </tr>
    </tbody>
  </table>
</div>
...
```

The directive's selector matches any element that has the pa-attr attribute, regardless of whether a value has been assigned to it or what that value is. The second step to applying a directive is to change the configuration of the Angular module, as shown in Listing 13-5.

Listing 13-5. Configuring the Component in the app.module.ts File in the src/app Folder

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

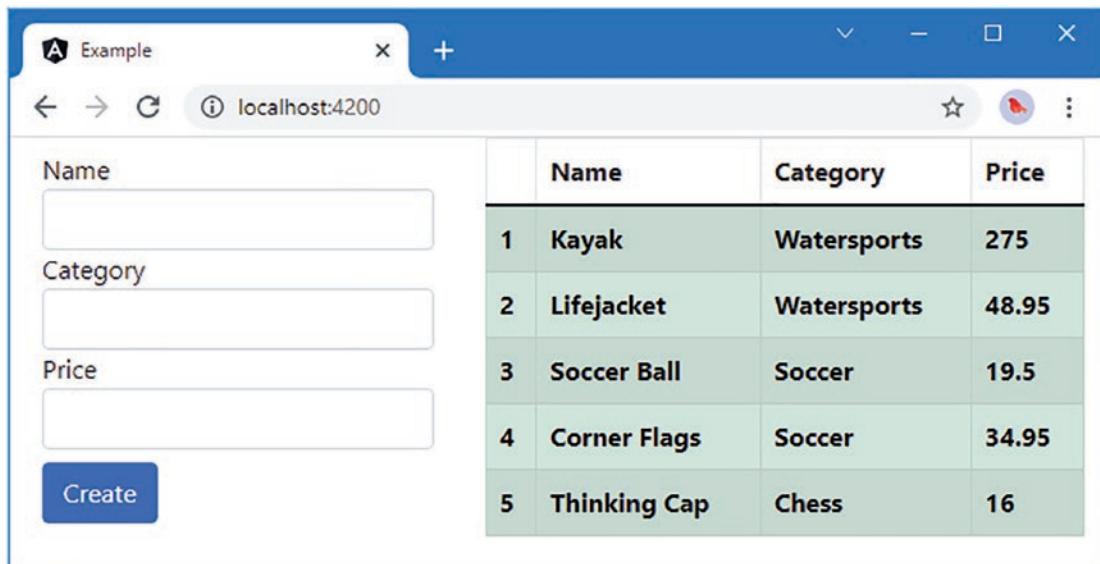
import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

import { ProductComponent } from './component';
import { FormsModule, ReactiveFormsModule } from '@angular/forms';
import { PaAttrDirective } from './attr.directive';

@NgModule({
  declarations: [ProductComponent, PaAttrDirective],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule, ReactiveFormsModule
  ],
  providers: [],
  bootstrap: [ProductComponent]
})
export class AppModule { }
```

The declarations property of the NgModule decorator declares the directives and components that the application will use. Don't worry if the relationship and differences between directives and components seem muddled at the moment; this will all become clear in Chapter 15.

Once both steps have been completed, the effect is that the `pa-attr` attribute applied to the `tr` element in the template will trigger the custom directive, which uses the DOM API to add the element to the `bg-success` and `text-white` classes. Since the `tr` element is part of the micro-template used by the `ngFor` directive, all the rows in the table are affected, as shown in Figure 13-2. (You may have to restart the Angular development tools to see the change.)



The screenshot shows a web browser window titled "Example" at "localhost:4200". On the left, there is a form with three input fields: "Name", "Category", and "Price", each with a corresponding empty text input field below it. Below the inputs is a blue "Create" button. To the right of the form is a table with five rows of data. The table has four columns: "Name", "Category", "Price", and a numeric index column (1, 2, 3, 4, 5). The rows are styled with alternating background colors. The first row (index 1) has the class `bg-success` and text color `white`. The other four rows (indices 2-5) have the class `bg-light` and text color `black`.

	Name	Category	Price
1	Kayak	Watersports	275
2	Lifejacket	Watersports	48.95
3	Soccer Ball	Soccer	19.5
4	Corner Flags	Soccer	34.95
5	Thinking Cap	Chess	16

Figure 13-2. Applying a custom directive

Accessing Application Data in a Directive

The example in the previous section shows the basic structure of a directive, but it doesn't do anything that couldn't be performed just by using a `class` property binding on the `tr` element. Directives become useful when they can interact with the host element and with the rest of the application.

Reading Host Element Attributes

The simplest way to make a directive more useful is to configure it using attributes applied to the host element, which allows each instance of the directive to be provided with its own configuration information and to adapt its behavior accordingly.

As an example, Listing 13-6 applies the directive to some of the `td` elements in the template table and adds an attribute that specifies the class that the host element should be added to. The directive's selector means that it will match any element that has the `pa-attr` attribute, regardless of the tag type, and will work as well on `td` elements as it does on `tr` elements. This listing also removes the `pa-attr` attribute from the `tr` element.

Listing 13-6. Adding Attributes in the template.html File in the src/app Folder

```
...
<tbody>
  <tr *ngFor="let item of getProducts(); let i = index">
    <td>{{i + 1}}</td>
    <td>{{item.name}}</td>
    <td pa-attr pa-attr-class="table-warning">{{item.category}}</td>
    <td pa-attr pa-attr-class="table-info">{{item.price}}</td>
  </tr>
</tbody>
...

```

The `pa-attr` attribute has been applied to two of the `td` elements, along with a new attribute called `pa-attr-class`, which has been used to specify the class to which the directive should add the host element. Listing 13-7 shows the changes required to the directive to get the value of the `pa-attr-class` attribute and use it to change the element.

Listing 13-7. Reading an Attribute in the attr.directive.ts File in the src/app Folder

```
import { Directive, ElementRef, Attribute } from "@angular/core";

@Directive({
  selector: "[pa-attr]",
})
export class PaAttrDirective {

  constructor(element: ElementRef, @Attribute("pa-attr-class") bgClass: string) {
    element.nativeElement.classList.add(bgClass || "table-success", "fw-bold");
  }
}
```

To receive the value of the `pa-attr-class` attribute, I added a new constructor parameter called `bgClass` to which the `@Attribute` decorator has been applied. This decorator is defined in the `@angular/core` module, and it specifies the name of the attribute that should be used to provide a value for the constructor parameter when a new instance of the directive class is created. Angular creates a new instance of the decorator for each element that matches the selector and uses that element's attributes to provide the values for the directive constructor arguments that have been decorated with `@Attribute`.

Within the constructor, the value of the attribute is passed to the `classList.add` method, with a default value that allows the directive to be applied to elements that have the `pa-attr` attribute but not the `pa-attr-class` attribute. Notice that I used the null coalescing operator (`||`) and not the nullish operator (`??`) in Listing 13-7. I want the fallback value to be used if an element defines the `pa-attr-class` attribute but does not assign it a value, in which case the `bgClass` parameter will be set to the empty string, which the `||` operator evaluates as false.

The result is that the class to which elements are added can now be specified using an attribute, producing the result shown in Figure 13-3.

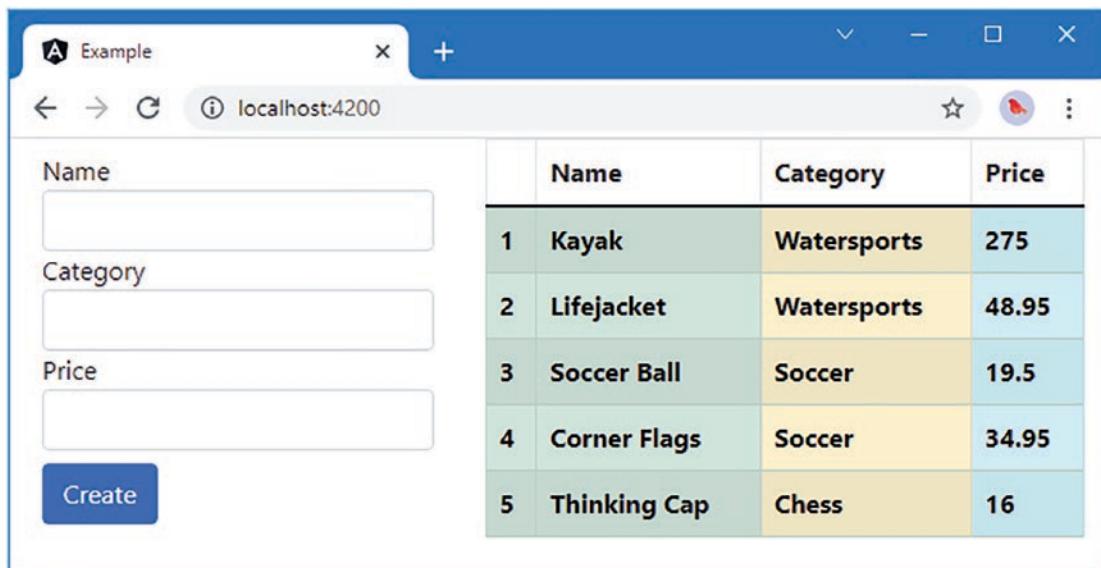


Figure 13-3. Configuring a directive using a host element attribute

Using a Single Host Element Attribute

Using one attribute to apply a directive and another to configure it is redundant, and it makes more sense to make a single attribute do double duty, as shown in Listing 13-8.

Listing 13-8. Reusing an Attribute in the attr.directive.ts File in the src/app Folder

```
import { Directive, ElementRef, Attribute } from "@angular/core";

@Directive({
  selector: "[pa-attr]",
})
export class PaAttrDirective {

  constructor(element: ElementRef, @Attribute("pa-attr") bgClass: string) {
    element.nativeElement.classList.add(bgClass || "table-success", "fw-bold");
  }
}
```

The `@Attribute` decorator now specifies the `pa-attr` attribute as the source of the `bgClass` parameter value. In Listing 13-9, I have updated the template to reflect the dual-purpose attribute.

Listing 13-9. Applying a Directive in the template.html File in the src/app Folder

```
...
<tbody>
  <tr *ngFor="let item of getProducts(); let i = index">
    <td>{{i + 1}}</td>
    <td>{{item.name}}</td>
```

```

<td pa-attr pa-attr-class="table-warning">{{item.category}}</td>
<td pa-attr pa-attr-class="table-info">{{item.price}}</td>
</tr>
</tbody>
...

```

There is no visual change in the result produced by this example, but it has simplified the way that the directive is applied in the HTML template.

Creating Data-Bound Input Properties

The main limitation of reading attributes with `@Attribute` is that values are static. The real power in Angular directives comes through support for expressions that are updated to reflect changes in the application state and that can respond by changing the host element.

Directives receive expressions using *data-bound input properties*, also known as *input properties* or, simply, *inputs*. Listing 13-10 changes the application's template so that the `pa-attr` attributes applied to the `tr` and `td` elements contain expressions, rather than just static class names.

Listing 13-10. Using Expressions in the template.html File in the src/app Folder

```

...
<tbody>
<tr *ngFor="let item of getProducts(); let i = index"
    [pa-attr]="getProducts().length < 6 ? 'table-success' : 'table-warning'"
    <td>{{i + 1}}</td>
    <td>{{item.name}}</td>
    <td [pa-attr]="item.category == 'Soccer' ? 'table-info' : null">
        {{item.category}}
    </td>
    <td [pa-attr]="'table-info'">{{item.price}}</td>
</tr>
</tbody>
...

```

There are three expressions in the listing. The first, which is applied to the `tr` element, uses the number of objects returned by the component's `getProducts` method to select a class.

```

...
<tr *ngFor="let item of getProducts(); let i = index"
    [pa-attr]="getProducts().length < 6 ? 'table-success' : 'table-warning'">
...

```

The second expression, which is applied to the `td` element for the Category column, specifies the `table-info` class for Product objects whose `Category` property returns Soccer and null for all other values.

```

...
<td [pa-attr]="item.category == 'Soccer' ? 'table-info' : null">
...

```

The third and final expression returns a fixed string value, which I have enclosed in single quotes, since this is an expression and not a static attribute value.

```
...
<td [pa-attr]="'table-info'">{{item.price}}</td>
...
```

Notice that the attribute name is enclosed in square brackets. That's because the way to receive an expression in a directive is to create a data binding, just like the built-in directives that are described in Chapters 11 and 12.

Tip Forgetting to use the square brackets is a common mistake. Without them, Angular will just pass the raw text of the expression to the directive without evaluating it. This is the first thing to check if you encounter an error when applying a custom directive.

Implementing the other side of the data binding means creating an input property in the directive class and telling Angular how to manage its value, as shown in Listing 13-11.

Listing 13-11. Defining an Input Property in the attr.directive.ts File in the src/app Folder

```
import { Directive, ElementRef, Input } from "@angular/core";

@Directive({
  selector: "[pa-attr]"
})
export class PaAttrDirective {

  constructor(private element: ElementRef) {}

  @Input("pa-attr")
  bgClass: string | null = "";

  ngOnInit() {
    this.element.nativeElement.classList.add(this.bgClass || "table-success",
      "fw-bold");
  }
}
```

Input properties are defined by applying the @Input decorator to a property and using it to specify the name of the attribute that contains the expression. This listing defines a single input property, which tells Angular to set the value of the directive's `bgClass` property to the value of the expression contained in the `pa-attr` attribute.

Tip You don't need to provide an argument to the @Input decorator if the name of the property corresponds to the name of the attribute on the host element. So, if you apply @Input() to a property called `myVal`, then Angular will look for a `myVal` attribute on the host element.

The role of the constructor has changed in this example. When Angular creates a new instance of a directive class, the constructor is invoked to create a new directive object, and only then is the value of the input property set. This means that the constructor cannot access the input property value because its value will not be set by Angular until after the constructor has completed and the new directive object has been produced. To address this, directives can implement *lifecycle hook methods*, which Angular uses to provide directives with useful information after they have been created and while the application is running, as described in Table 13-3.

Table 13-3. The Directive Lifecycle Hook Methods

Name	Description
ngOnInit	This method is called after Angular has set the initial value for all the input properties that the directive has declared.
ngOnChanges	This method is called when the value of an input property has changed and also just before the ngOnInit method is called.
ngDoCheck	This method is called when Angular runs its change detection process so that directives have an opportunity to update any state that isn't directly associated with an input property.
ngAfterContentInit	This method is called when the directive's content has been initialized. See the “Receiving Query Change Notifications” section in Chapter 14 for an example that uses this method.
ngAfterContentChecked	This method is called after the directive's content has been inspected as part of the change detection process.
ngOnDestroy	This method is called immediately before Angular destroys a directive.

To set the class on the host element, the directive in Listing 13-11 implements the ngOnInit method, which is called after Angular has set the value of the bgClass property. The constructor is still needed to receive the ElementRef object that provides access to the host element, which is assigned to a property called element.

The result is that Angular will create a directive object for each tr element, evaluate the expressions specified in the pa-attr attribute, use the results to set the value of the input properties, and then call the ngOnInit methods, which allows the directives to respond to the new input property values.

To see the effect, use the form to add a new product to the example application. Since there are initially five items in the model, the expression for the tr element will select the bg-success class. When you add a new item, Angular will create another instance of the directive class and evaluate the expression to set the value of the input property; since there are now six items in the model, the expression will select the bg-warning class, which provides the new row with a different background color, as shown in Figure 13-4.

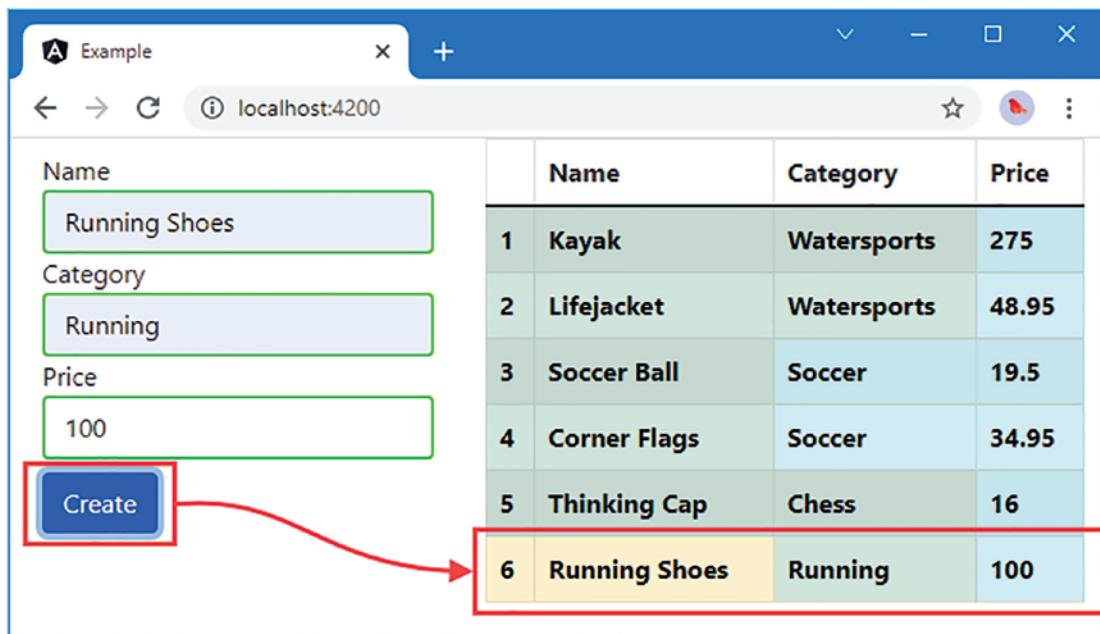


Figure 13-4. Using an input property in a custom directive

Responding to Input Property Changes

Something odd happened in the previous example: adding a new item affected the appearance of the new elements but not the existing elements. Behind the scenes, Angular has updated the value of the `bgClass` property for each of the directives that it created—one for each `td` element in the table column—but the directives didn't notice because changing a property value doesn't automatically cause directives to respond.

To handle changes, a directive must implement the `ngOnChanges` method to receive notifications when the value of an input property changes, as shown in Listing 13-12.

Listing 13-12. Receiving Change Notifications in the attr.directive.ts File in the src/app Folder

```
import { Directive, ElementRef, Input, SimpleChanges } from "@angular/core";

@Directive({
  selector: "[pa-attr]"
})
export class PaAttrDirective {

  constructor(private element: ElementRef) {}

  @Input("pa-attr")
  bgClass: string | null = "";

  // ngOnInit() {
  //   this.element.nativeElement.classList.add(this.bgClass || "table-success",
  // }
```

```

//      "fw-bold");
// }

ngOnChanges(changes: SimpleChanges) {
  let change = changes["bgClass"];
  let classList = this.nativeElement.classList;
  if (!change.isFirstChange() && classList.contains(change.previousValue)) {
    classList.remove(change.previousValue);
  }
  if (!classList.contains(change.currentValue)) {
    classList.add(change.currentValue);
  }
}
}
}

```

The `ngOnChanges` method is called once before the `ngOnInit` method and then called again each time there are changes to any of a directive's input properties. The `ngOnChanges` parameter is a `SimpleChanges` object, which is a map whose keys refer to each changed input property and whose values are `SimpleChange` objects, which are defined in the `@angular/core` module. The `SimpleChange` class defines the members shown in Table 13-4.

Table 13-4. The Properties and Method of the `SimpleChange` Class

Name	Description
<code>previousValue</code>	This property returns the previous value of the input property.
<code>currentValue</code>	This property returns the current value of the input property.
<code>isFirstChange()</code>	This method returns true if this is the call to the <code>ngOnChanges</code> method that occurs before the <code>ngOnInit</code> method.

When responding to changes to the input property value, a directive has to make sure to account for the effect of previous updates. In the case of the example directive, this means removing the element from the `previousValue` class and adding it to the `currentValue` class instead.

It is important to use the `isFirstChange` method so that you don't undo a value that hasn't actually been applied since the `ngOnChanges` method is called the first time a value is assigned to the input property.

The result of handling these change notifications is that the directive responds when Angular reevaluates the expressions and updates the input properties. Now when you add a new product to the application, the background colors for all the `tr` elements are updated, as shown in Figure 13-5.

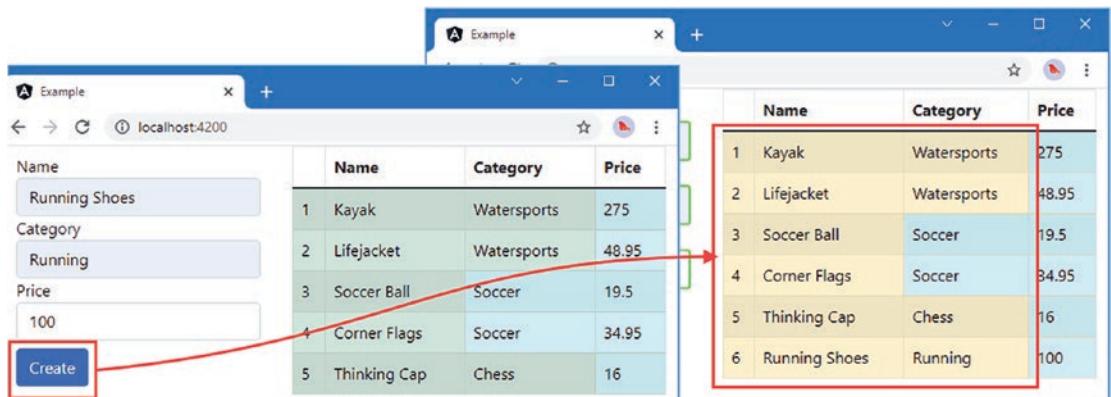


Figure 13-5. Responding to input property changes

Creating Custom Events

Output properties are the Angular feature that allows directives to add custom events to their host elements, through which details of important changes can be sent to the rest of the application. Output properties are defined using the `@Output` decorator, which is defined in the `@angular/core` module, as shown in Listing 13-13.

Listing 13-13. Defining an Output Property in the `attr.directive.ts` File in the `src/app` Folder

```
import { Directive, ElementRef, Input, SimpleChanges, Output, EventEmitter }  
from "@angular/core";  
import { Product } from "./product.model";  
  
@Directive({  
    selector: "[pa-attr]"  
})  
export class PaAttrDirective {  
  
    constructor(private element: ElementRef) {  
        this.element.nativeElement.addEventListener("click", () => {  
            if (this.product != null) {  
                this.click.emit(this.product.category);  
            }  
        });  
    }  
  
    @Input("pa-attr")  
    bgClass: string | null = "";  
  
    @Input("pa-product")  
    product: Product = new Product();  
  
    @Output("pa-category")  
    click = new EventEmitter<string>();
```

```

ngOnChanges(changes: SimpleChanges) {
  let change = changes["bgClass"];
  let classList = this.element.nativeElement.classList;
  if (!change.isFirstChange() && classList.contains(change.previousValue)) {
    classList.remove(change.previousValue);
  }
  if (!classList.contains(change.currentValue)) {
    classList.add(change.currentValue);
  }
}
}

```

The `EventEmitter<T>` interface provides the event mechanism for Angular directives. The listing creates an `EventEmitter<string>` object and assigns it to a variable called `click`, like this:

```

...
@Output("pa-category")
click = new EventEmitter<string>();
...

```

The `string` type parameter indicates that listeners to the event will receive a `string` when the event is triggered. Directives can provide any type of object to their event listeners, but common choices are `string` and `number` values, data model objects, and JavaScript Event objects.

The custom event in the listing is triggered when the mouse button is clicked on the host element, and the event provides its listeners with the `category` of the `Product` object that was used to create the table row using the `ngFor` directive. The effect is that the directive is responding to a DOM event on the host element and generating its own custom event in response. The listener for the DOM event is set up in the directive class constructor using the browser's standard `addEventListener` method, like this:

```

...
constructor(private element: ElementRef) {
  this.element.nativeElement.addEventListener("click", () => {
    if (this.product != null) {
      this.click.emit(this.product.category);
    }
  });
}
...

```

The directive defines an input property to receive the `Product` object whose `category` will be sent in the event. (The directive can refer to the value of the input property value in the constructor because Angular will have set the property value before the function assigned to handle the DOM event is invoked.)

The most important statement in the listing is the one that uses the `EventEmitter<string>` object to send the event, which is done using the `EventEmitter.emit` method, which is described in Table 13-5 for quick reference. The argument to the `emit` method is the value that you want the event listeners to receive, which is the value of the `category` property for this example.

Table 13-5. The EventEmitter Method

Name	Description
emit(value)	This method triggers the custom event associated with the EventEmitter, providing the listeners with the object or value received as the method argument.

Tying everything together is the @Output decorator, which creates a mapping between the directive's EventEmitter<string> property and the name that will be used to bind to the event in the template, like this:

```
...
@Output("pa-category")
click = new EventEmitter<string>();
...
...
```

The argument to the decorator specifies the attribute name that will be used in event bindings applied to the host element. You can omit the argument if the TypeScript property name is also the name you want for the custom event. I have specified pa-category in the listing, which allows me to refer to the event as click within the directive class but requires a more meaningful name externally.

Binding to a Custom Event

Angular makes it easy to bind to custom events in templates by using the same binding syntax that is used for built-in events, which was described in Chapter 12. Listing 13-14 adds the pa-product attribute to the tr element in the template to provide the directive with its Product object and adds a binding for the pa-category event.

Listing 13-14. Binding to a Custom Event in the template.html File in the src/app Folder

```
...
<tbody>
  <tr *ngFor="let item of getProducts(); let i = index"
      [pa-attr]="getProducts().length < 6 ? 'table-success' : 'table-warning'"
      [pa-product]="item" (pa-category)="newProduct.category = $event">
    <td>{{i + 1}}</td>
    <td>{{item.name}}</td>
    <td [pa-attr]="item.category == 'Soccer' ? 'table-info' : null">
      {{item.category}}
    </td>
    <td [pa-attr]="'table-info'">{{item.price}}</td>
  </tr>
</tbody>
...
...
```

The term \$event is used to access the value the directive passed to the EventEmitter<string>.emit method. That means \$event will be a string value containing the product category in this example. The value received from the event is used to set the value of the category input element, meaning that clicking a row in the table displays the product's category in the form, as shown in Figure 13-6.

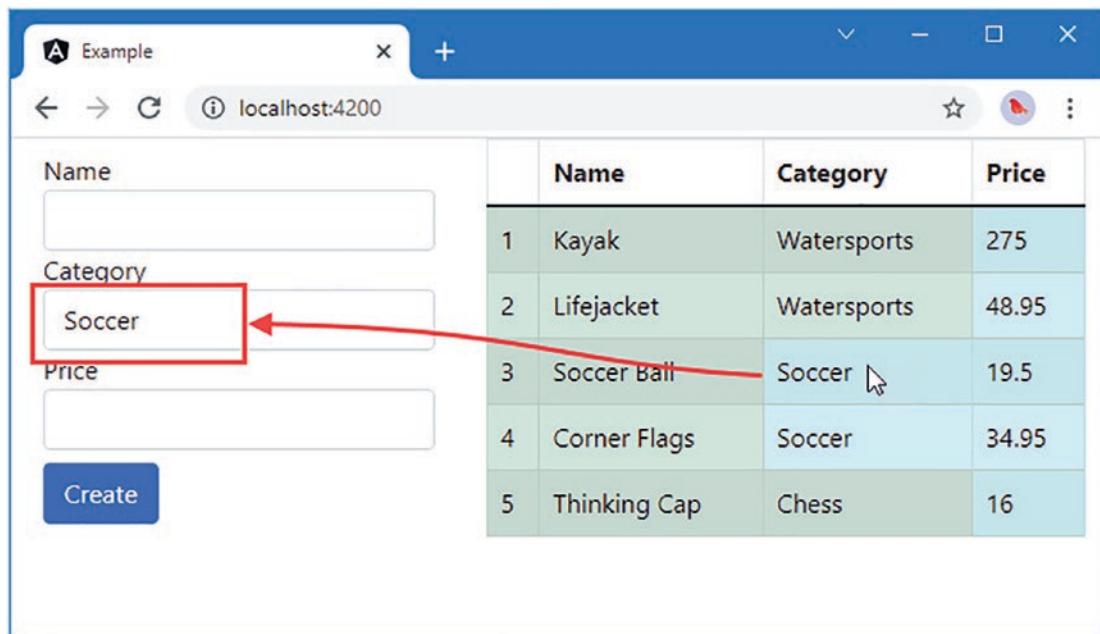


Figure 13-6. Defining and receiving a custom event using an output property

Note Behind the scenes, Angular uses the Reactive Extensions package to distribute events. The `EventEmitter<T>` interface extends the RxJS `Subject<T>` interface, which, in turn, extends the `Observable<T>` interface.

Creating Host Element Bindings

The example directive relies on the browser’s DOM API to manipulate its host element, both to add and remove class memberships and to receive the `click` event. Working with the DOM API in an Angular application is a useful technique, but it does mean that your directive can be used only in applications that are run in a web browser. Angular is intended to be run in a range of different execution environments, and not all of them can be assumed to provide the DOM API.

Even if you are sure that a directive will have access to the DOM, the same results can be achieved in a more elegant way using standard Angular directive features: property and event bindings. Rather than use the DOM to add and remove classes, a class binding can be used on the host element. And rather than use the `addEventListener` method, an event binding can be used to deal with the mouse click.

Behind the scenes, Angular implements these features using the DOM API when the directive is used in a web browser—or some equivalent mechanism when the directive is used in a different environment.

Bindings on the host element are defined using two decorators, `@HostBinding` and `@HostListener`, both of which are defined in the `@angular/core` module, as shown in Listing 13-15.

Listing 13-15. Creating Host Bindings in the attr.directive.ts File in the src/app Folder

```
import { Directive, ElementRef, Input, SimpleChanges, Output,
    EventEmitter, HostListener, HostBinding } from "@angular/core";
import { Product } from "./product.model";

@Directive({
    selector: "[pa-attr]"
})
export class PaAttrDirective {

    // constructor(private element: ElementRef) {
    //     this.element.nativeElement.addEventListener("click", () => {
    //         if (this.product != null) {
    //             this.click.emit(this.product.category);
    //         }
    //     });
    // }

    @Input("pa-attr")
    @HostBinding("class")
    bgClass: string | null = "";

    @Input("pa-product")
    product: Product = new Product();

    @Output("pa-category")
    click = new EventEmitter<string>();

    // ngOnChanges(changes: SimpleChanges) {
    //     let change = changes["bgClass"];
    //     let classList = this.element.nativeElement.classList;
    //     if (!change.isFirstChange() && classList.contains(change.previousValue)) {
    //         classList.remove(change.previousValue);
    //     }
    //     if (!classList.contains(change.currentValue)) {
    //         classList.add(change.currentValue);
    //     }
    // }

    @HostListener("click")
    triggerCustomEvent() {
        if (this.product != null) {
            this.click.emit(this.product.category);
        }
    }
}
```

The `@HostBinding` decorator is used to set up a property binding on the host element and is applied to a directive property. The listing sets up a binding between the `class` property on the host element and the decorator's `bgClass` property.

Tip If you want to manage the contents of an element, you can use the `@HostBinding` decorator to bind to the `textContent` property. See Chapter 17 for an example.

The `@HostListener` decorator is used to set up an event binding on the host element and is applied to a method. The listing creates an event binding for the `click` event that invokes the `triggerCustomEvent` method when the mouse button is pressed and released. The `triggerCustomEvent` method uses the `EventEmitter.emit` method to dispatch the custom event through the `output` property.

Using the host element bindings means that the directive constructor can be removed since there is no longer any need to access the HTML element via the `ElementRef` object. Instead, Angular takes care of setting up the event listener and setting the element's class membership through the property binding.

Although the directive code is much simpler, the effect of the directive is the same: clicking a table row sets the value of one of the `input` elements, and adding a new item using the form triggers a change in the background color of the table cells for products that are not part of the Soccer category.

Creating a Two-Way Binding on the Host Element

Directives can support two-way bindings, which means they can be used with the banana-in-a-box bracket style that `ngModel` uses and can bind to a model property in both directions.

The two-way binding feature relies on a naming convention. To demonstrate how it works, Listing 13-16 adds some new elements and bindings to the `template.html` file.

Listing 13-16. Applying a Directive in the `template.html` File in the `src/app` Folder

```
...
<div class="col">
  <div class="form-group bg-info text-white p-2">
    <label>Name:</label>
    <input class="bg-primary text-white form-control"
      [paModel]="newProduct.name"
      (paModelChange)="newProduct.name = $event" />
  </div>

  <table class="table table-bordered table-striped">
    <thead>
      <tr><th></th><th>Name</th><th>Category</th><th>Price</th></tr>
    </thead>
    <tbody>
      <tr *ngFor="let item of getProducts(); let i = index"
        [pa-attr]="getProducts().length < 6
          ? 'table-success' : 'table-warning'"
        [pa-product]="item" (pa-category)="newProduct.category = $event">
        <td>{{i + 1}}</td>
        <td>{{item.name}}</td>
        <td [pa-attr]="item.category == 'Soccer' ? 'table-info' : null">
          {{item.category}}
        </td>
        <td [pa-attr]="'table-info'">{{item.price}}</td>
      </tr>
    </tbody>
  </table>
</div>
```

```

        </tbody>
    </table>
</div>
...

```

The binding whose target is `paModel` will be updated when the value of the `newProduct.name` property changes, which provides a flow of data from the application to the directive and will be used to update the contents of the `input` element. The custom event, `paModelChange`, will be triggered when the user changes the contents of the name `input` element and will provide a flow of data from the directive to the rest of the application.

To implement the directive, I added a file called `twoWay.directive.ts` to the `src/app` folder and used it to define the directive shown in Listing 13-17.

Listing 13-17. The Contents of the `twoWay.directive.ts` File in the `src/app` Folder

```

import {
    Input, Output, EventEmitter, Directive,
    HostBinding, HostListener, SimpleChange
} from "@angular/core";

@Directive({
    selector: "input[paModel]"
})
export class PaModel {

    @Input("paModel")
    modelProperty: string | undefined = "";

    @HostBinding("value")
    fieldValue: string = "";

    ngOnChanges(changes: { [property: string]: SimpleChange }) {
        let change = changes["modelProperty"];
        if (change.currentValue != this.fieldValue) {
            this.fieldValue = changes["modelProperty"].currentValue || "";
        }
    }

    @Output("paModelChange")
    update = new EventEmitter<string>();

    @HostListener("input", ["$event.target.value"])
    updateValue(newValue: string) {
        this.fieldValue = newValue;
        this.update.emit(newValue);
    }
}

```

This directive uses features that have been described previously. The `selector` property for this directive specifies that it will match `input` elements that have a `paModel` attribute. The built-in `ngModel` two-way directive has support for a range of form elements and knows which events and properties each of them

uses, but I want to keep this example simple, so I am going to support just `input` elements, which define a `value` property that gets and sets the element content.

The `paModel` binding is implemented using an `input` property and the `ngOnChanges` method, which responds to changes in the expression value by updating the contents of the `input` element through a host binding on the `input` element's `value` property.

The `paModelChange` event is implemented using a host listener on the `input` event, which then sends an update through an `output` property. Notice that the method invoked by the event can receive the event object by specifying an additional argument to the `@HostListener` decorator, like this:

```
...
@HostListener("input", ["$event.target.value"])
updateValue(newValue: string) {
...
}
```

The first argument to the `@HostListener` decorator specifies the name of the event that will be handled by the listener. The second argument is an array that will be used to provide the decorated methods with arguments. In this example, the `input` event will be handled by the listener, and when the `updateValue` method is invoked, its `newValue` argument will be set to the `target.value` property of the `Event` object, which is referred to using `$event`.

To enable the directive, I added it to the Angular module, as shown in Listing 13-18.

Listing 13-18. Registering the Directive in the `app.module.ts` File in the `src/app` Folder

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

import { ProductComponent } from './component';
import { FormsModule, ReactiveFormsModule } from "@angular/forms";
import { PaAttrDirective } from "./attr.directive";
import { PaModel } from "./twoway.directive";

@NgModule({
  declarations: [ProductComponent, PaAttrDirective, PaModel],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule, ReactiveFormsModule
  ],
  providers: [],
  bootstrap: [ProductComponent]
})
export class AppModule { }
```

When you save the changes and the browser has reloaded, you will see a new `input` element that responds to changes to a `model` property and updates the `model` property if its host element's content is changed. The expressions in the bindings specify the same `model` property used by the `Name` field in the form on the left side of the HTML document, which provides a convenient way to test the relationship between them, as shown in Figure 13-7.

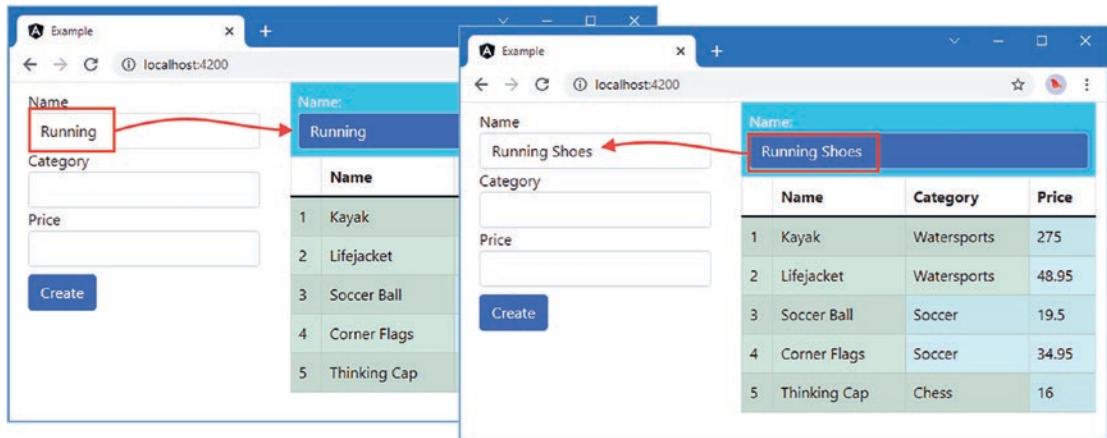


Figure 13-7. Testing the two-way flow of data

Tip You may need to stop the Angular development tools, restart them, and reload the browser for this example. The Angular development tools don't always process the changes correctly.

The final step is to simplify the bindings and apply the banana-in-a-box style of brackets, as shown in Listing 13-19.

Listing 13-19. Simplifying the Bindings in the template.html File in the src/app Folder

```
...
<div class="col">

    <div class="form-group bg-info text-white p-2">
        <label>Name:</label>
        <input class="bg-primary text-white form-control"
            [(paModel)]="newProduct.name" />
    </div>

    <table class="table table-bordered table-striped">
        <thead>
            <tr><th></th><th>Name</th><th>Category</th><th>Price</th></tr>
        </thead>
        <tbody>
            <tr *ngFor="let item of getProducts(); let i = index"
                [pa-attr]="getProducts().length < 6
                    ? 'table-success' : 'table-warning'"
                [pa-product]="item" (pa-category)="newProduct.category = $event">
                <td>{{i + 1}}</td>
                <td>{{item.name}}</td>
                <td [pa-attr]="item.category == 'Soccer' ? 'table-info' : null">
                    {{item.category}}
                </td>
            </tr>
        </tbody>
    </table>
</div>
```

```

        <td [pa-attr]="'table-info'">{{item.price}}</td>
    </tr>
</tbody>
</table>
</div>
...

```

When Angular encounters the [()] brackets, it expands the binding to match the format used in Listing 13-16, targeting the `paModel` input property and setting up the `paModelChange` event. As long as a directive exposes these to Angular, it can be targeted using the banana-in-a-box brackets, producing a simpler template syntax.

Exporting a Directive for Use in a Template Variable

In earlier chapters, I used template variables to access functionality provided by built-in directives, such as `ngForm`. As an example, here is an element from Chapter 12:

```

...
<form #form="ngForm" (ngSubmit)="submitForm(form)">
...

```

The `form` template variable is assigned `ngForm`, which is then used to access validation information for the HTML form. This is an example of how a directive can provide access to its properties and methods so they can be used in data bindings and expressions.

Listing 13-20 modifies the directive from the previous section so that it provides details of whether it has expanded the text in its host element.

Listing 13-20. Exporting a Directive in the `twoway.directive.ts` File in the `src/app` Folder

```

import {
  Input, Output, EventEmitter, Directive,
  HostBinding, HostListener, SimpleChange
} from "@angular/core";

@Directive({
  selector: "input[paModel]",
  exportAs: "paModel"
})
export class PaModel {

  direction: string = "None";

  @Input("paModel")
  modelProperty: string | undefined = "";

  @HostBinding("value")
  fieldValue: string = "";

  ngOnChanges(changes: { [property: string]: SimpleChange }) {
    let change = changes["modelProperty"];
    if (change.currentValue != this.fieldValue) {

```

```

        this.fieldValue = changes["ModelProperty"].currentValue || "";
this.direction = "Model";
    }
}

@Output("paModelChange")
update = new EventEmitter<string>();

@HostListener("input", ["$event.target.value"])
updateValue(newValue: string) {
    this.fieldValue = newValue;
    this.update.emit(newValue);
this.direction = "Element";
}
}

```

The `exportAs` property of the `@Directive` decorator specifies a name that will be used to refer to the directive in template variables. This example uses `paModel` as the value—also known as the *identifier*—for the `exportAs` property, and you should try to use names that make it clear which directive is providing the functionality.

The listing adds a property called `direction` to the directive, which used to indicate when data is flowing from the model to the element or from the element to the model.

When you use the `exportAs` decorator, you are providing access to all the methods and properties defined by the directive to be used in template expressions and data bindings. Some developers prefix the names of the methods and properties that are not for use outside of the directive with an underscore (the `_` character) or apply the `private` keyword. This is an indication to other developers that some methods and properties should not be used but isn't enforced by Angular. Listing 13-21 creates a template variable for the directive's exported functionality and uses it in a style binding.

Listing 13-21. Using Exported Directive Functionality in the template.html File in the src/app Folder

```

...
<div class="form-group bg-info text-white p-2">
    <label>Name:</label>
    <input class="bg-primary text-white form-control"
        [(paModel)]="newProduct.name" #paModel="paModel" />
    <div class="bg-info text-white p-1">Direction: {{paModel.direction}}</div>
</div>
...

```

The template variable is called `paModel`, and its value is the name used in the directive's `exportAs` property.

```

...
#paModel="paModel"
...
```

Tip You don't have to use the same names for the variable and the directive, but it does help to make the source of the functionality clear.

Once the template variable has been defined, it can be used in interpolation bindings or as part of a binding expression. I opted for a string interpolation binding whose expression uses the value of the directive's `direction` property.

```
...  
<div class="bg-info text-white p-1">Direction: {{paModel.direction}}</div>  
...
```

The result is that you can see the effect of typing text into the two input elements that are bound to the `newProduct.name` model property. When you type into the one that uses the `ngModel` directive, then the string interpolation binding will display `Model`. When you type into the element that uses the `paModel` directive, the string interpolation binding will display `Element`, as shown in Figure 13-8.

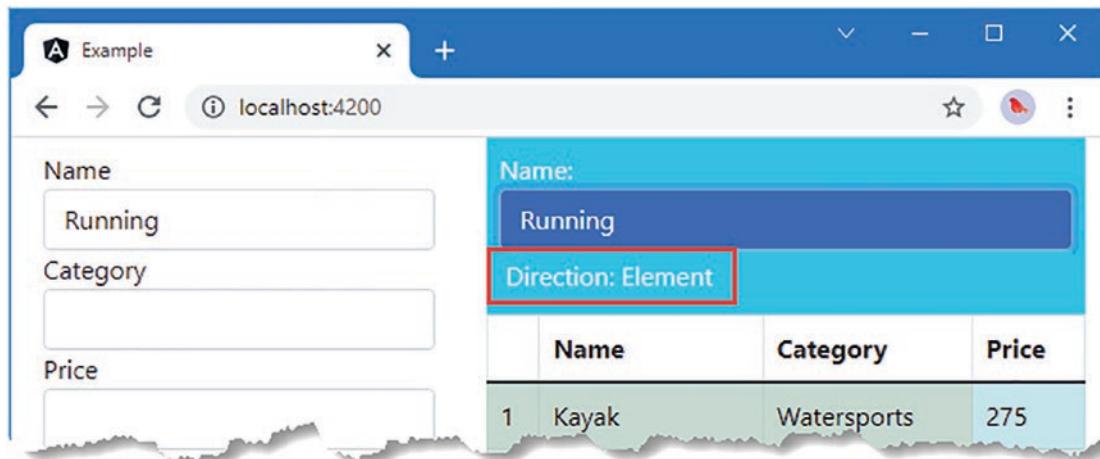


Figure 13-8. Exporting functionality from a directive

Summary

In this chapter, I described how to define and use attribute directives, including the use of input and output properties and host bindings. In the next chapter, I explain how structural directives work and how they can be used to change the layout or structure of the HTML document.

CHAPTER 14



Creating Structural Directives

Structural directives change the layout of the HTML document by adding and removing elements. They build on the core features available for attribute directives, described in Chapter 13, with additional support for micro-templates, which are small fragments of contents defined within the templates used by components. You can recognize when a structural directive is being used because its name will be prefixed with an asterisk, such as `*ngIf` or `*ngFor`. In this chapter, I explain how structural directives are defined and applied, how they work, and how they respond to changes in the data model. Table 14-1 puts structural directives in context.

Table 14-1. Putting Structural Directives in Context

Question	Answer
What are they?	Structural directives use micro-templates to add content to the HTML document.
Why are they useful?	Structural directives allow content to be added conditionally based on the result of an expression or for the same content to be repeated for each object in a data source, such as an array.
How are they used?	Structural directives are applied to an <code>ng-template</code> element, which contains the content and bindings that comprise its micro-template. The template class uses objects provided by Angular to control the inclusion of the content or to repeat the content.
Are there any pitfalls or limitations?	Unless care is taken, structural directives can make a lot of unnecessary changes to the HTML document, which can ruin the performance of a web application. It is important to make changes only when they are required, as explained in the “Dealing with Collection-Level Data Changes” section later in the chapter.
Are there any alternatives?	You can use the built-in directives for common tasks, but writing custom structural directives provides the ability to tailor behavior to your application.

Table 14-2 summarizes the chapter.

Table 14-2. Chapter Summary

Problem	Solution	Listing
Creating a structural directive	Apply the <code>@Directive</code> decorator to a class that receives view container and template constructor parameters	1-6
Creating an iterating structural directive	Define a <code>ForOf</code> input property in a structural directive class and iterate over its value	7-12
Handling data changes in a structural directive	Use a differ to detect changes in the <code>ngDoCheck</code> method	13-19
Querying the content of the host element to which a structural directive has been applied	Use the <code>@ContentChild</code> or <code>@ContentChildren</code> decorator	20-26

Preparing the Example Project

In this chapter, I continue working with the example project that I created in Chapter 9 and have been using since. To prepare for this chapter, I simplified the template to remove the form, leaving only the table, as shown in Listing 14-1. (I'll add the form back in later in the chapter.)

Tip You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/Apress/pro-angular-5ed>. See Chapter 1 for how to get help if you have problems running the examples.

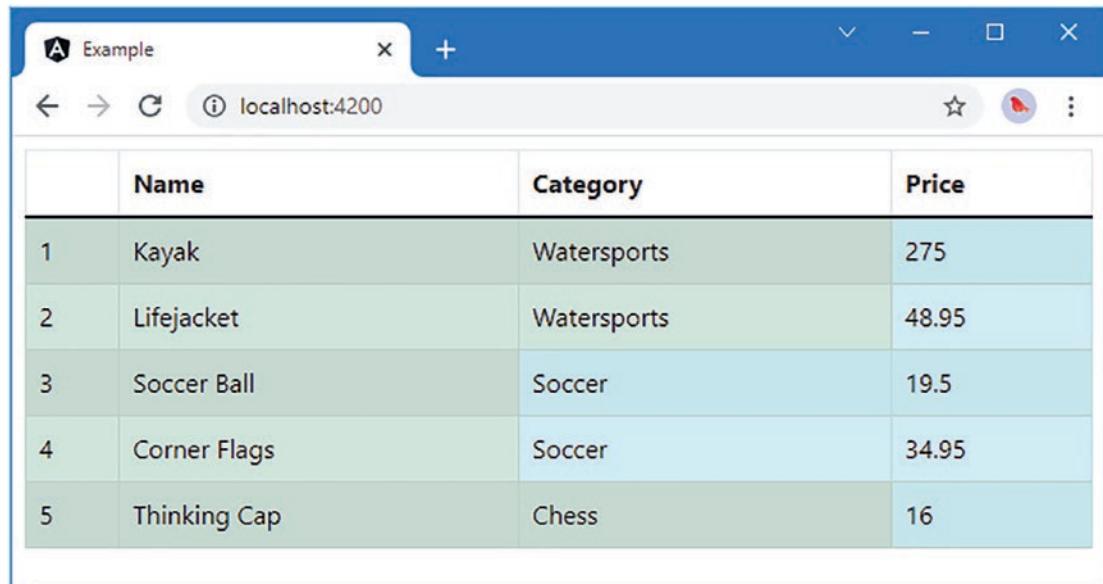
Listing 14-1. Simplifying the Template in the template.html File in the src/app Folder

```
<div class="p-2">
  <table class="table table-bordered table-striped">
    <thead>
      <tr><th></th><th>Name</th><th>Category</th><th>Price</th></tr>
    </thead>
    <tbody>
      <tr *ngFor="let item of getProducts(); let i = index"
          [pa-attr]="getProducts().length < 6 ? 'table-success' : 'table-warning'"
          [pa-product]="item" (pa-category)="newProduct.category = $event">
        <td>{{i + 1}}</td>
        <td>{{item.name}}</td>
        <td [pa-attr]="item.category == 'Soccer' ? 'table-info' : null">
          {{item.category}}
        </td>
        <td [pa-attr]="'table-info'">{{item.price}}</td>
      </tr>
    </tbody>
  </table>
</div>
```

Run the following command in the example folder to start the development tools:

```
ng serve
```

Open a new browser window and navigate to `http://localhost:4200` to see the content shown in Figure 14-1.



The screenshot shows a Microsoft Edge browser window titled "Example". The address bar displays "localhost:4200". The main content is a table with five rows and four columns. The columns are labeled "Name", "Category", and "Price". The data is as follows:

	Name	Category	Price
1	Kayak	Watersports	275
2	Lifejacket	Watersports	48.95
3	Soccer Ball	Soccer	19.5
4	Corner Flags	Soccer	34.95
5	Thinking Cap	Chess	16

Figure 14-1. Running the example application

Creating a Simple Structural Directive

A good place to start with structural directives is to re-create the functionality provided by the `ngIf` directive, which is relatively simple, is easy to understand, and provides a good foundation for explaining how structural directives work. I start by making changes to the template and working backward to write the code that supports it. Listing 14-2 shows the template changes.

Listing 14-2. Applying a Structural Directive in the template.html File in the src/app Folder

```
<div class="p-2">

<div class="form-check m-2">
  <input type="checkbox" class="form-check-input" [(ngModel)]="showTable" />
  <label class="form-check-label">Show Table</label>
</div>

<ng-template [paIf]="showTable">
  <table class="table table-bordered table-striped">
    <thead>
```

```

<tr><th></th><th>Name</th><th>Category</th><th>Price</th></tr>
</thead>
<tbody>
  <tr *ngFor="let item of getProducts(); let i = index"
    [pa-attr]="getProducts().length < 6
      ? 'table-success' : 'table-warning'"
    [pa-product]="item" (pa-category)="newProduct.category = $event">
    <td>{{i + 1}}</td>
    <td>{{item.name}}</td>
    <td [pa-attr]="item.category == 'Soccer' ? 'table-info' : null">
      {{item.category}}
    </td>
    <td [pa-attr]="'table-info'">{{item.price}}</td>
  </tr>
</tbody>
</table>
</ng-template>
</div>

```

This listing uses the full template syntax, in which the directive is applied to an `ng-template` element, which contains the content that will be used by the directive. In this case, the `ng-template` element contains the `table` element and all its contents, including bindings, directives, and expressions. (There is also a concise syntax, which I use later in the chapter.)

The `ng-template` element has a standard one-way data binding, which targets a directive called `paIf`, like this:

```

...
<ng-template [paIf]="showTable">
...

```

The expression for this binding uses the value of a property called `showTable`. This is the same property that is used in the other new binding in the template, which has been applied to a checkbox, as follows:

```

...
<input type="checkbox" class="form-check-input" [(ngModel)]="showTable" />
...

```

The objectives in this section are to create a structural directive that will add the contents of the `ng-template` element to the HTML document when the `showTable` property is true, which will happen when the checkbox is checked, and to remove the contents of the `ng-template` element when the `showTable` property is false, which will happen when the checkbox is unchecked. Listing 14-3 adds the `showTable` property to the component.

Listing 14-3. Adding a Property in the component.ts File in the src/app Folder

```

import { Component } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";

@Component({
  selector: "app",
  templateUrl: "template.html"
})

```

```

})
export class ProductComponent {
  model: Model = new Model();
  showTable: boolean = false;

  getProduct(key: number): Product | undefined {
    return this.model.getProduct(key);
  }

  getProducts(): Product[] {
    return this.model.getProducts();
  }

  newProduct: Product = new Product();

  addProduct(p: Product) {
    this.model.saveProduct(p);
  }

  submitForm() {
    this.addProduct(this.newProduct);
  }
}

```

Implementing the Structural Directive Class

You know from the template what the directive should do. To implement the directive, I added a file called `structure.directive.ts` in the `src/app` folder and added the code shown in Listing 14-4.

Listing 14-4. The Contents of the `structure.directive.ts` File in the `src/app` Folder

```

import {
  Directive, SimpleChanges, ViewContainerRef, TemplateRef, Input
} from "@angular/core";

@Directive({
  selector: "[paIf]"
})
export class PaStructureDirective {

  constructor(private container: ViewContainerRef,
    private template: TemplateRef<Object>) { }

  @Input("paIf")
  expressionResult: boolean | undefined;

  ngOnChanges(changes: SimpleChanges) {
    let change = changes["expressionResult"];
    if (!change.isFirstChange() && !change.currentValue) {
      this.container.clear();
    }
  }
}

```

```

        } else if (change.currentValue) {
            this.container.createEmbeddedView(this.template);
        }
    }
}

```

The selector property of the `@Directive` decorator is used to match host elements that have the `paIf` attribute; this corresponds to the template additions that I made in Listing 14-1.

There is an input property called `expressionResult`, which the directive uses to receive the results of the expression from the template. The directive implements the `ngOnChanges` method to receive change notifications so it can respond to changes in the data model.

The first indication that this is a structural directive comes from the constructor, which asks Angular to provide parameters using some new types.

```

...
constructor(private container: ViewContainerRef,
    private template: TemplateRef<Object>) {}
...

```

The `ViewContainerRef` object is used to manage the contents of the *view container*, which is the part of the HTML document where the `ng-template` element appears and for which the directive is responsible.

As its name suggests, the view container is responsible for managing a collection of *views*. A view is a region of HTML elements that contains directives, bindings, and expressions, and they are created and managed using the methods and properties provided by the `ViewContainerRef` class, the most useful of which are described in Table 14-3.

Table 14-3. Useful `ViewContainerRef` Methods and Properties

Name	Description
<code>element</code>	This property returns an <code>ElementRef</code> object that represents the container element.
<code>createEmbeddedView(template)</code>	This method uses a template to create a new view. See the text after the table for details. This method also accepts optional arguments for context data (as described in the “Creating Iterating Structural Directives” section) and an index position that specifies where the view should be inserted. The result is a <code>ViewRef</code> object that can be used with the other methods in this table.
<code>clear()</code>	This method removes all the views from the container.
<code>length</code>	This property returns the number of views in the container.
<code>get(index)</code>	This method returns the <code>ViewRef</code> object representing the view at the specified index.
<code>indexOf(view)</code>	This method returns the index of the specified <code>ViewRef</code> object.
<code>insert(view, index)</code>	This method inserts a view at the specified index.
<code>remove(index)</code>	This method removes and destroys the view at the specified index.
<code>detach(index)</code>	This method detaches the view from the specified index without destroying it so that it can be repositioned with the <code>insert</code> method.

Two of the methods from Table 14-3 are required to re-create the `ngIf` directive's functionality: `createEmbeddedView` to show the `ng-template` element's content to the user and `clear` to remove it again.

The `createEmbeddedView` method adds a view to the view container. This method's argument is a `TemplateRef` object, which represents the content of the `ng-template` element.

The directive receives the `TemplateRef` object as one of its constructor arguments, for which Angular will provide a value automatically when creating a new instance of the directive class.

Putting everything together, when Angular processes the `template.html` file, it discovers the `ng-template` element, examines its binding, and determines that it needs to create a new instance of the `PaStructureDirective` class. Angular examines the `PaStructureDirective` constructor and can see that it needs to provide it with `ViewContainerRef` and `TemplateRef` objects.

```
...
constructor(private container: ViewContainerRef,
    private template: TemplateRef<Object>) {}
...
...
```

The `ViewContainerRef` object represents the place in the HTML document occupied by the `ng-template` element, and the `TemplateRef` object represents the `ng-template` element's contents. Angular passes these objects to the constructor and creates a new instance of the directive class.

Angular then starts processing the expressions and data bindings. As described in Chapter 13, Angular invokes the `ngOnChanges` method during initialization (just before the `ngOnInit` method is invoked) and again whenever the value of the directive's expression changes.

The `PaStructureDirective` class's implementation of the `ngOnChanges` method uses the `SimpleChanges` object that it receives to show or hide the contents of the `ng-template` element based on the current value of the expression. When the expression is true, the directive displays the `ng-template` element's content by adding them to the container view.

```
...
this.container.createEmbeddedView(this.template);
...
...
```

When the result of the expression is `false`, the directive clears the view container, which removes the elements from the HTML document.

```
...
this.container.clear();
...
...
```

The directive doesn't have any insight into the contents of the `ng-template` element and is responsible only for managing its visibility.

Enabling the Structural Directive

The directive must be enabled in the Angular module before it can be used, as shown in Listing 14-5.

Listing 14-5. Enabling the Directive in the `app.module.ts` File in the `src/app` Folder

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';
```

```

import { BrowserModule } from '@angular/platform-browser/animations';
import { ProductComponent } from './component';
import { FormsModule, ReactiveFormsModule } from "@angular/forms";
import { PaAttrDirective } from "./attr.directive";
import { PaModel } from "./twoway.directive";
import { PaStructureDirective } from "./structure.directive";

@NgModule({
  declarations: [ProductComponent, PaAttrDirective, PaModel, PaStructureDirective],
  imports: [
    BrowserModule,
    BrowserModule,
    FormsModule, ReactiveFormsModule
  ],
  providers: [],
  bootstrap: [ProductComponent]
})
export class AppModule { }

```

Structural directives are enabled in the same way as attribute directives and are specified in the module's declarations array.

Once you save the changes, the browser will reload the HTML document, and you can see the effect of the new directive: the `table` element, which is the content of the `ng-template` element, will be shown only when the checkbox is checked, as shown in Figure 14-2. (If you don't see the changes or the table isn't shown when you check the box, restart the Angular development tools and then reload the browser window.)

Note The contents of the `ng-template` element are being destroyed and re-created, not simply hidden and revealed. If you want to show or hide content without removing it from the HTML document, then you can use a style binding to set the `display` or `visibility` property.

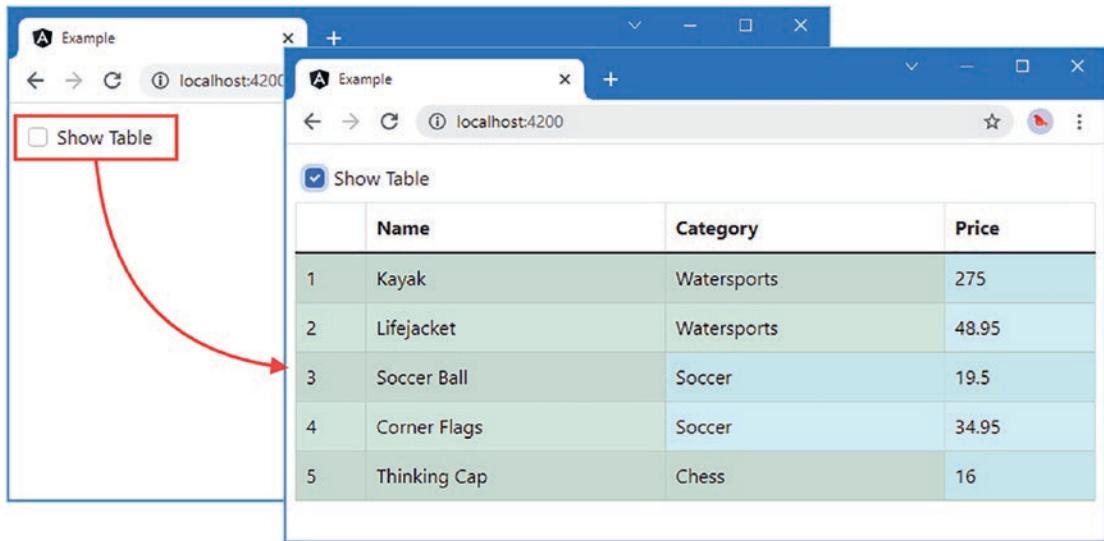


Figure 14-2. Creating a structural directive

Using the Concise Structural Directive Syntax

The use of the `ng-template` element helps illustrate the role of the view container in structural directives. The concise syntax does away with the `ng-template` element and applies the directive and its expression to the outermost element that it would contain, as shown in Listing 14-6.

Tip The concise structural directive syntax is intended to be easier to use and read, but it is just a matter of preference as to which syntax you use.

Listing 14-6. Using the Concise Structural Directive Syntax in the template.html File in the src/app Folder

```
<div class="p-2">

<div class="form-check m-2">
  <input type="checkbox" class="form-check-input" [(ngModel)]="showTable" />
  <label class="form-check-label">Show Table</label>
</div>

<table *paIf="showTable" class="table table-sm table-bordered table-striped">
  <thead>
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th></tr>
  </thead>
  <tbody>
    <tr *ngFor="let item of getProducts(); let i = index"
       [pa-attr]="getProducts().length < 6
       ? 'table-success' : 'table-warning'"
       [pa-product]="item" (pa-category)="newProduct.category = $event">
```

```

<td>{{i + 1}}</td>
<td>{{item.name}}</td>
<td [pa-attr]="item.category == 'Soccer' ? 'table-info' : null">
    {{item.category}}
</td>
<td [pa-attr]="'table-info'">{{item.price}}</td>
</tr>
</tbody>
</table>
</div>

```

The `ng-template` element has been removed, and the directive has been applied to the `table` element, like this:

```

...
<table *paIf="showTable" class="table table-sm table-bordered table-striped">
...

```

The directive's name is prefixed with an asterisk (the `*` character) to tell Angular that this is a structural directive that uses the concise syntax. When Angular parses the `template.html` file, it discovers the directive and the asterisk and handles the elements as though there were an `ng-template` element in the document. No changes are required to the directive class to support the concise syntax.

Creating Iterating Structural Directives

Angular provides special support for directives that need to iterate over a data source. The best way to demonstrate this is to re-create another of the built-in directives: `ngFor`.

To prepare for the new directive, I have removed the `ngFor` directive from the `template.html` file, inserted an `ng-template` element, and applied a new directive attribute and expression, as shown in Listing 14-7.

Listing 14-7. Preparing for a New Structural Directive in the `template.html` File in the `src/app` Folder

```

<div class="m-2">
  <div class="checkbox">
    <label>
      <input type="checkbox" [(ngModel)]="showTable" />
      Show Table
    </label>
  </div>

  <table *paIf="showTable" class="table table-sm table-bordered table-striped">
    <thead>
      <tr><th></th><th>Name</th><th>Category</th><th>Price</th></tr>
    </thead>
    <tbody>
      <ng-template [paForOf]="getProducts()" let-item>
        <tr><td colspan="4">{{item.name}}</td></tr>
      </ng-template>
    </tbody>
  </table>
</div>

```

The full syntax for iterating structural directives is a little odd. In the listing, the `ng-template` element has two attributes that are used to apply the directive. The first is a standard binding whose expression obtains the data required by the directive, bound to an attribute called `paForOf`.

```
...
<ng-template [paForOf]="getProducts()" let-item>
...
```

The name of this attribute is important. When using an `ng-template` element, the name of the data source attribute must end with `Of` to support the concise syntax, which I will introduce shortly.

The second attribute is used to define the *implicit value*, which allows the currently processed object to be referred to within the `ng-template` element as the directive iterates through the data source. Unlike other template variables, the implicit variable isn't assigned a value, and its purpose is only to define the variable name.

```
...
<ng-template [paForOf]="getProducts()" let-item>
...
```

In this example, I have used `let-item` to tell Angular that I want the implicit value to be assigned to a variable called `item`, which is then used within a string interpolation binding to display the `name` property of the current data item.

```
...
<td colspan="4">&{{item.name}}</td>
...
```

Looking at the `ng-template` element, you can see that the purpose of the new directive is to iterate through the component's `getProducts` method and generate a table row for each of them that displays the `name` property. To implement this functionality, I created a file called `iterator.directive.ts` in the `src/app` folder and defined the directive shown in Listing 14-8.

Listing 14-8. The Contents of the iterator.directive.ts File in the src/app Folder

```
import { Directive, ViewContainerRef, TemplateRef, Input }
    from "@angular/core";

@Directive({
    selector: "[paForOf]"
})
export class PaIteratorDirective {

    constructor(private container: ViewContainerRef,
        private template: TemplateRef<Object>) { }

    @Input("paForOf")
    dataSource: any;

    ngOnInit() {
        this.container.clear();
        for (let i = 0; i < this.dataSource.length; i++) {
            this.container.createEmbeddedView(this.template,
                new PaIteratorContext(this.dataSource[i]));
        }
    }
}
```

```

        }
    }

class PaIteratorContext {
    constructor(public $implicit: any) { }
}

```

The selector property in the `@Directive` decorator matches elements with the `paForOf` attribute, which is also the source of the data for the `dataSource` input property and which provides the source of objects that will be iterated.

The `ngOnInit` method will be called once the value of the `input` property has been set, and the directive empties the view container using the `clear` method and adds a new view for each object using the `createEmbeddedView` method.

When calling the `createEmbeddedView` method, the directive provides two arguments: the `TemplateRef` object received through the constructor and a context object. The `TemplateRef` object provides the content to insert into the container, and the context object provides the data for the `$implicit` value, which is specified using a property called `$implicit`. It is this object, with its `$implicit` property, that is assigned to the `item` template variable and that is referred to in the string interpolation binding. To provide templates with the context object in a type-safe way, I defined a class called `PaIteratorContext`, whose only property is called `$implicit`.

The `ngOnInit` method reveals some important aspects of working with view containers. First, a view container can be populated with multiple views—in this case, one view per object in the data source. The `ViewContainerRef` class provides the functionality required to manage these views once they have been created, as you will see in the sections that follow.

Second, a template can be reused to create multiple views. In this example, the contents of the `ng-template` element will be used to create identical `tr` and `td` elements for each object in the data source. The `td` element contains a data binding, which is processed by Angular when each view is created and is used to tailor the content to its data object.

Third, the directive has no special knowledge about the data it is working with and no knowledge of the content that is being generated. Angular takes care of providing the directive with the context it needs from the rest of the application, providing the data source through the `input` property and providing the content for each view through the `TemplateRef` object.

Enabling the directive requires an addition to the Angular module, as shown in Listing 14-9.

Listing 14-9. Adding a Custom Directive in the `app.module.ts` File in the `src/app` Folder

```

import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

import { ProductComponent } from './component';
import { FormsModule, ReactiveFormsModule } from "@angular/forms";
import { PaAttrDirective } from "./attr.directive";
import { PaModel } from "./twoWay.directive";
import { PaStructureDirective } from "./structure.directive";
import { PaIteratorDirective } from "./iterator.directive";

```

```

@NgModule({
  declarations: [ProductComponent, PaAttrDirective, PaModel, PaStructureDirective,
    PaIteratorDirective],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule, ReactiveFormsModule
  ],
  providers: [],
  bootstrap: [ProductComponent]
})
export class AppModule { }

```

The result is that the directive iterates through the objects in its data source and uses the `ng-template` element's content to create a view for each of them, providing rows for the table, as shown in Figure 14-3. You will need to check the box to show the table. (If you don't see the changes, then start the Angular development tools and reload the browser window.)

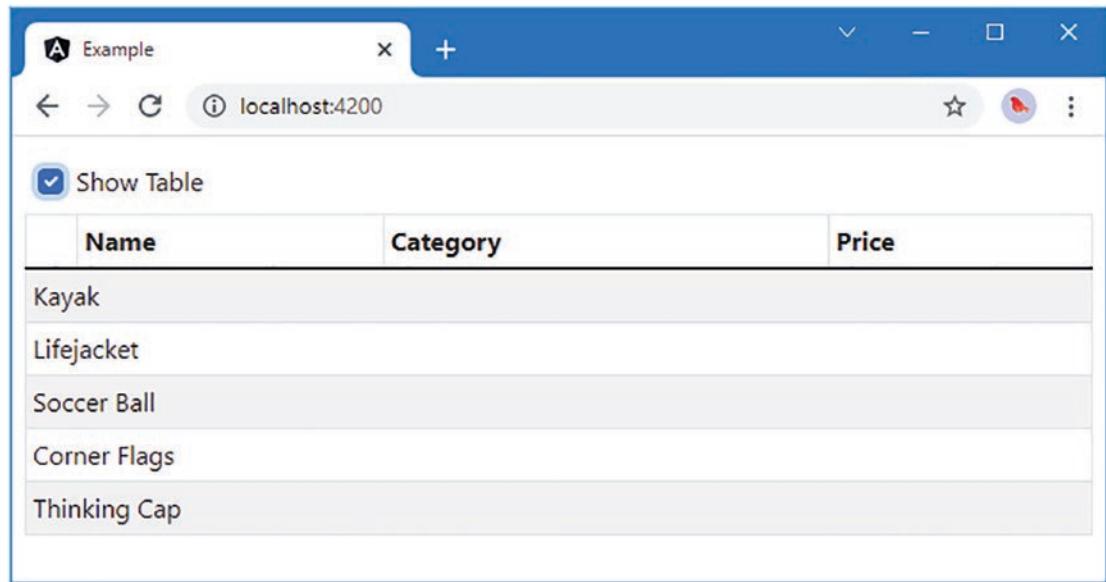


Figure 14-3. Creating an iterating structural directive

Providing Additional Context Data

Structural directives can provide templates with additional values to be assigned to template variables and used in bindings. For example, the `ngFor` directive provides `odd`, `even`, `first`, and `last` values. Context values are provided through the same object that defines the `$implicit` property, and in Listing 14-10, I have re-created the same set of values that `ngFor` provides.

Listing 14-10. Providing Context Data in the iterator.directive.ts File in the src/app Folder

```
import { Directive, ViewContainerRef, TemplateRef, Input}
    from "@angular/core";

@Directive({
    selector: "[paForOf]"
})
export class PaIteratorDirective {

    constructor(private container: ViewContainerRef,
        private template: TemplateRef<Object>) { }

    @Input("paForOf")
    dataSource: any;

    ngOnInit() {
        this.container.clear();
        for (let i = 0; i < this.dataSource.length; i++) {
            this.container.createEmbeddedView(this.template,
                new PaIteratorContext(this.dataSource[i],
                    i, this.dataSource.length));
        }
    }
}

class PaIteratorContext {
    odd: boolean; even: boolean;
    first: boolean; last: boolean;

    constructor(public $implicit: any,
        public index: number, total: number ) {

        this.odd = index % 2 == 1;
        this.even = !this.odd;
        this.first = index == 0;
        this.last = index == total - 1;
    }
}
```

This listing defines additional properties in the `PaIteratorContext` class and expands its constructor so that it receives additional parameters, which are used to set the property values.

The effect of these additions is that context object properties can be used to create template variables, which can then be referred to in binding expressions, as shown in Listing 14-11.

Listing 14-11. Using Structural Directive Context Data in the template.html File in the src/app Folder

```
<div class="p-2">
    <div class="form-check m-2">
        <input type="checkbox" class="form-check-input" [(ngModel)]="showTable" />
        <label class="form-check-label">Show Table</label>
```

```

</div>

<table *paIf="showTable" class="table table-sm table-bordered table-striped">
  <thead>
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th></tr>
  </thead>
  <tbody>
    <ng-template [paForOf]="getProducts()" let-item let-i="index"
      let-odd="odd" let-even="even">
      <tr [class.table-info]="odd" [class.table-warning]="even">
        <td>{{i + 1}}</td>
        <td>{{item.name}}</td>
        <td>{{item.category}}</td>
        <td>{{item.price}}</td>
      </tr>
    </ng-template>
  </tbody>
</table>
</div>

```

Template variables are created using the `let-<name>` attribute syntax and assigned one of the context data values. In this listing, I used the odd and even context values to create template variables of the same name, which are then incorporated into class bindings on the `tr` element, resulting in striped table rows, as shown in Figure 14-4. The listing also adds table cells to display all the Product properties.

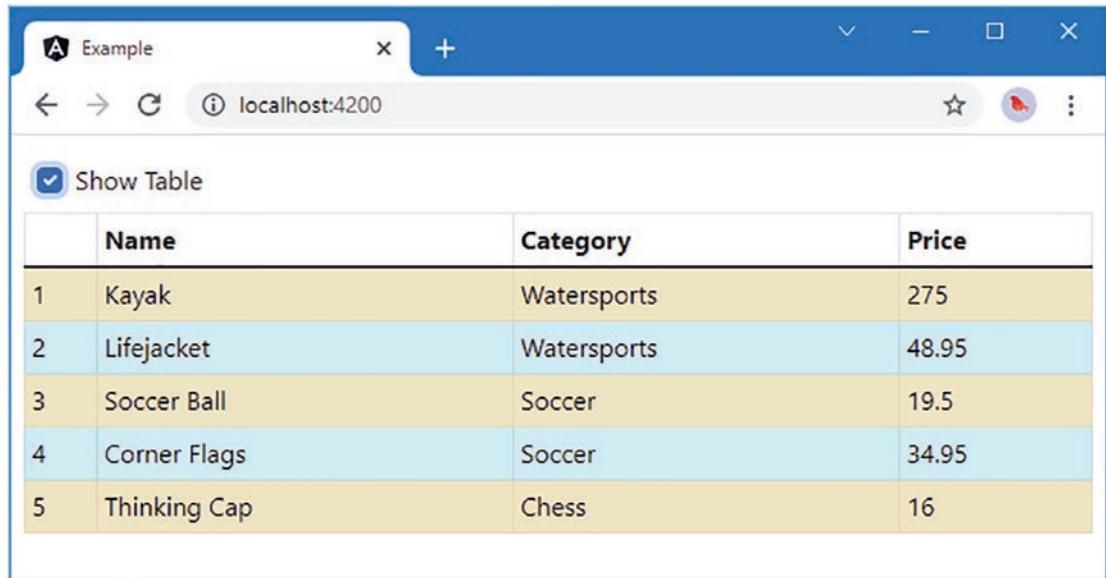


Figure 14-4. Using directive context data

Using the Concise Structure Syntax

Iterating structural directives support the concise syntax and omit the `ng-template` element, as shown in Listing 14-12.

Listing 14-12. Using the Concise Syntax in the template.html File in the src/app Folder

```
<div class="p-2">
  <div class="form-check m-2">
    <input type="checkbox" class="form-check-input" [(ngModel)]="showTable" />
    <label class="form-check-label">Show Table</label>
  </div>

  <table *paIf="showTable" class="table table-sm table-bordered table-striped">
    <thead>
      <tr><th></th><th>Name</th><th>Category</th><th>Price</th></tr>
    </thead>
    <tbody>
      <tr *paFor="let item of getProducts(); let i = index; let odd = odd;
           let even = even" [class.table-info]="odd"
              [class.table-warning]="even">
        <td>{{i + 1}}</td>
        <td>{{item.name}}</td>
        <td>{{item.category}}</td>
        <td>{{item.price}}</td>
      </tr>
    </tbody>
  </table>
</div>
```

This is a more substantial change than the one required for attribute directives. The biggest change is in the attribute used to apply the directive. When using the full syntax, the directive was applied to the `ng-template` element using the attribute specified by its selector, like this:

```
...
<ng-template [paForOf]="getProducts()" let-item let-i="index" let-odd="odd"
             let-even="even">
...

```

When using the concise syntax, the `Of` part of the attribute is omitted, the name is prefixed with an asterisk, and the brackets are omitted.

```
...
<tr *paFor="let item of getProducts(); let i = index; let odd = odd;
     let even = even" [class.table-info]="odd" [class.table-warning]="even">
...

```

The other change is to incorporate all the context values into the directive's expression, replacing the individual `let-` attributes. The main data value becomes part of the initial expression, with additional context values separated by semicolons.

No changes are required to the directive to support the concise syntax, whose selector and input property still specify an attribute called `paForOf`. Angular takes care of expanding the concise syntax, and the directive doesn't know or care whether an `ng-template` element has been used.

Dealing with Property-Level Data Changes

There are two kinds of changes that can occur in the data sources used by iterating structural directives. The first kind happens when the properties of an individual object change. This has a knock-on effect on the data bindings contained within the `ng-template` element, either directly through a change in the implicit value or indirectly through the additional context values provided by the directive. Angular takes care of these changes automatically, reflecting any changes in the context data in the bindings that depend on them.

To demonstrate, in Listing 14-13 I have added a call to the standard JavaScript `setInterval` function in the constructor of the context class. The function passed to `setInterval` alters the `odd` and `even` properties and changes the value of the `price` property of the `Product` object that is used as the implicit value.

Listing 14-13. Modifying Individual Objects in the iterator.directive.ts File in the src/app Folder

```
...
class PaIteratorContext {
    odd: boolean; even: boolean;
    first: boolean; last: boolean;

    constructor(public $implicit: any,
               public index: number, total: number) {

        this.odd = index % 2 == 1;
        this.even = !this.odd;
        this.first = index == 0;
        this.last = index == total - 1;

        setInterval(() => {
            this.odd = !this.odd; this.even = !this.even;
            this.$implicit.price++;
        }, 2000);
    }
}
...
```

Once every two seconds, the values of the `odd` and `even` properties are inverted, and the `price` value is incremented. When you save the changes, you will see that the colors of the table rows change and the prices slowly increase, as illustrated in Figure 14-5.

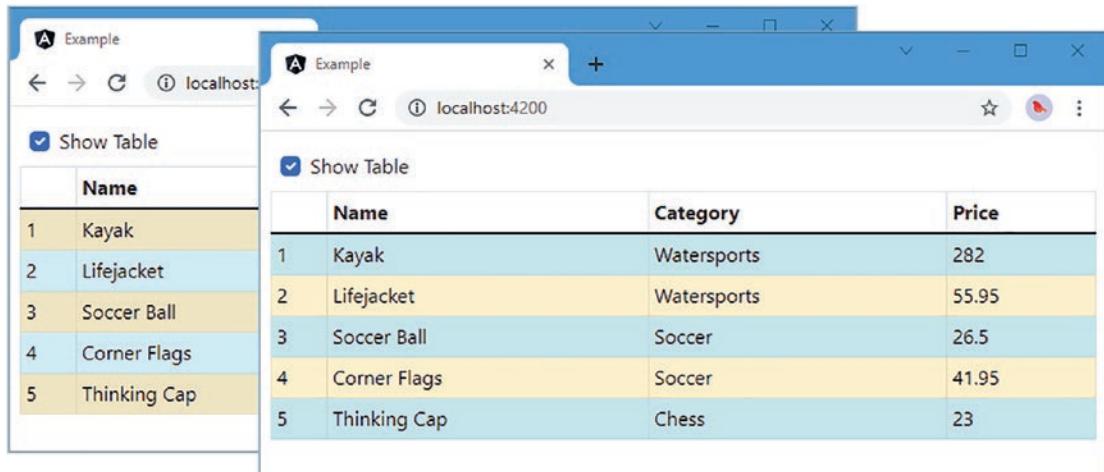


Figure 14-5. Automatic change detection for individual data source objects

Dealing with Collection-Level Data Changes

The second type of change occurs when the objects within the collection are added, removed, or replaced. Angular doesn't detect this kind of change automatically, which means the iterating directive's `ngOnChanges` method won't be invoked.

Receiving notifications about collection-level changes is done by implementing the `ngDoCheck` method, which is called whenever a data change is detected in the application, regardless of where that change occurs or what kind of change it is. The `ngDoCheck` method allows a directive to respond to changes even when they are not automatically detected by Angular. Implementing the `ngDoCheck` method requires caution, however, because it represents a pitfall that can destroy the performance of a web application. To demonstrate the problem, Listing 14-14 implements the `ngDoCheck` method so that the directive updates the content it displays when there is a change.

Listing 14-14. Implementing the `ngDoCheck` Methods in the iterator.directive.ts File in the src/app Folder

```
import { Directive, ViewContainerRef, TemplateRef, Input }
    from "@angular/core";

@Directive({
  selector: "[paForOf]"
})
export class PaIteratorDirective {

  constructor(private container: ViewContainerRef,
    private template: TemplateRef<Object>) { }

  @Input("paForOf")
  dataSource: any;
```

```

ngOnInit() {
  this.updateContent();
}

ngDoCheck() {
  console.log("ngDoCheck Called");
  this.updateContent();
}

private updateContent() {
  this.container.clear();
  for (let i = 0; i < this.dataSource.length; i++) {
    this.container.createEmbeddedView(this.template,
      new PaIteratorContext(this.dataSource[i],
        i, this.dataSource.length));
  }
}

class PaIteratorContext {
  odd: boolean; even: boolean;
  first: boolean; last: boolean;

  constructor(public $implicit: any,
    public index: number, total: number ) {

    this.odd = index % 2 == 1;
    this.even = !this.odd;
    this.first = index == 0;
    this.last = index == total - 1;

    // setInterval(() => {
    //   this.odd = !this.odd; this.even = !this.even;
    //   this.$implicit.price++;
    // }, 2000);
  }
}

```

The ngOnInit and ngDoCheck methods both call a new updateContent method that clears the contents of the view container and generates new template content for each object in the data source. I have also commented out the call to the setInterval function in the PaIteratorContext class.

To understand the problem with collection-level changes and the ngDoCheck method, I need to restore the form to the component's template, as shown in Listing 14-15. I also removed the checkbox and removed the directive from the table so that it is always displayed.

Listing 14-15. Restoring the HTML Form in the template.html File in the src/app Folder

```

<div class="container-fluid">
  <div class="row p-2">
    <div class="col-4">
      <form class="m-2" (ngSubmit)="submitForm()">
        <div class="form-group">

```

```

<label>Name</label>
<input class="form-control" name="name"
   [(ngModel)]="newProduct.name" />
</div>
<div class="form-group">
  <label>Category</label>
  <input class="form-control" name="category"
     [(ngModel)]="newProduct.category" />
</div>
<div class="form-group">
  <label>Price</label>
  <input class="form-control" name="price"
     [(ngModel)]="newProduct.price" />
</div>
<button class="btn btn-primary mt-2" type="submit">Create</button>
</form>
</div>

<div class="col">
  <table class="table table-sm table-bordered table-striped">
    <thead>
      <tr><th></th><th>Name</th><th>Category</th><th>Price</th></tr>
    </thead>
    <tbody>
      <tr *paFor="let item of getProducts(); let i = index;
           let odd = odd; let even = even" [class.table-info]="odd"
           [class.table-warning]="even">
        <td>{{i + 1}}</td>
        <td>{{item.name}}</td>
        <td>{{item.category}}</td>
        <td>{{item.price}}</td>
      </tr>
    </tbody>
  </table>
</div>
</div>

```

When you save the changes to the template, the HTML form will be displayed alongside the table of products, as shown in Figure 14-6.

	Name	Category	Price
1	Kayak	Watersports	275
2	Lifejacket	Watersports	48.95
3	Soccer Ball	Soccer	19.5
4	Corner Flags	Soccer	34.95
5	Thinking Cap	Chess	16

Figure 14-6. Restoring the table in the template

The problem with the `ngDoCheck` method is that it is invoked every time Angular detects a change anywhere in the application—and those changes happen more often than you might expect.

To demonstrate how often changes occur, I added a call to the `console.log` method within the directive's `ngDoCheck` method in Listing 14-14 so that a message will be displayed in the browser's JavaScript console each time the `ngDoCheck` method is called. Use the HTML form to create a new product and see how many messages are written out to the browser's JavaScript console, each of which represents a change detected by Angular and which results in a call to the `ngDoCheck` method.

A new message is displayed each time an input element gets the focus, each time a key event is triggered, each time a validation check is performed, and so on. A quick test adding a Running Shoes product in the Running category with a price of 100 generates 27 messages on my system, although the exact number will vary based on how you navigate between elements, whether you need to correct typos, and so on.

For each of those 27 times, the structural directive destroys and re-creates its content, which means producing new `tr` and `td` elements with new directive and binding objects.

There are only a few rows of data in the example application, but these are expensive operations, and a real application can grind to a halt as the content is repeatedly destroyed and re-created. The worst part of this problem is that all the changes except one were unnecessary because the content in the table didn't need to be updated until the new `Product` object was added to the data model. For all the other changes, the directive destroyed its content and created an identical replacement.

Fortunately, Angular provides some tools for managing updates more efficiently and updating content only when it is required. The `ngDoCheck` method will still be called for all changes in the application, but the directive can inspect its data to see whether any changes that require new content have occurred, as shown in Listing 14-16.

Listing 14-16. Minimizing Content Changes in the iterator.directive.ts File in the src/app Folder

```
import { Directive, ViewContainerRef, TemplateRef, Input,
  IterableDiffer, IterableDiffers, IterableChangeRecord }
from "@angular/core";
```

```

@Directive({
  selector: "[paForOf]"
})
export class PaIteratorDirective {
  private differ: IterableDiffer<any> | undefined;

  constructor(private container: ViewContainerRef,
    private template: TemplateRef<Object>,
    private differs: IterableDiffers) { }

  @Input("paForOf")
  dataSource: any;

  ngOnInit() {
    this.differ =
      <IterableDiffer<any>> this.differs.find(this.dataSource).create();
  }

  ngDoCheck() {
    let changes = this.differ?.diff(this.dataSource);
    if (changes != null) {
      console.log("ngDoCheck called, changes detected");
      let arr: IterableChangeRecord<any>[] = [];
      changes.forEachAddedItem(addition => arr.push(addition));
      arr.forEach(addition => {
        if (addition.currentIndex != null) {
          this.container.createEmbeddedView(this.template,
            new PaIteratorContext(addition.item, addition.currentIndex,
              arr.length));
        }
      });
    }
  }

  // private updateContent() {
  //   this.container.clear();
  //   for (let i = 0; i < this.dataSource.length; i++) {
  //     this.container.createEmbeddedView(this.template,
  //       new PaIteratorContext(this.dataSource[i],
  //         i, this.dataSource.length));
  //   }
  // }
}

class PaIteratorContext {
  odd: boolean; even: boolean;
  first: boolean; last: boolean;
}

```

```

constructor(public $implicit: any,
            public index: number, total: number ) {

    this.odd = index % 2 == 1;
    this.even = !this.odd;
    this.first = index == 0;
    this.last = index == total - 1;
}
}

```

The idea is to work out whether there have been objects added, removed, or moved from the collection. This means the directive has to do some work every time the `ngDoCheck` method is called to avoid unnecessary and expensive DOM operations when there are no collection changes to process.

The process starts in the constructor, which receives two new arguments whose values will be provided by Angular when a new instance of the directive class is created. The `IterableDiffers` object is used to set up change detection on the data source collection in the `ngOnInit` method, like this:

```

...
ngOnInit() {
    this.differ = <IterableDiffer<any>> this.differs.find(this.dataSource).create();
}
...

```

Angular includes built-in classes, known as *differs*, that can detect changes in different types of objects. The `IterableDiffers.find` method accepts an object and returns an `IterableDifferFactory` object that is capable of creating a differ class for that object. The `IterableDifferFactory` class defines a `create` method that returns a `IterableDiffer` object that will perform the change detection.

The important part of this incantation is the `IterableDiffer` object, which was assigned to a property called `differ` so that it can be used when the `ngDoCheck` method is called.

```

...
ngDoCheck() {
    let changes = this.differ?.diff(this.dataSource);
    if (changes != null) {
        console.log("ngDoCheck called, changes detected");
        let arr: IterableChangeRecord<any>[] = [];
        changes.forEachAddedItem(addition => arr.push(addition));
        arr.forEach(addition => {
            if (addition.currentIndex != null) {
                this.container.createEmbeddedView(this.template,
                    new PaIteratorContext(addition.item, addition.currentIndex,
                        arr.length));
            }
        });
    }
}
...
```

The `IterableDiffer.diff` method accepts an object for comparison and returns an `IterableChanges` object, which contains a list of the changes, or `null` if there have been no changes. Checking for the `null` result allows the directive to avoid unnecessary work when the `ngDoCheck` method is called for changes elsewhere in the application. The `IterableChanges` object returned by the `diff` method provides methods described in Table 14-4 for processing changes.

Table 14-4. The `IterableChanges` Methods and Properties

Name	Description
<code>forEachItem(func)</code>	This method invokes the specified function for each object in the collection.
<code>forEachPreviousItem(func)</code>	This method invokes the specified function for each object in the previous version of the collection.
<code>forEachAddedItem(func)</code>	This method invokes the specified function for each new object in the collection.
<code>forEachMovedItem(func)</code>	This method invokes the specified function for each object whose position has changed.
<code>forEachRemovedItem(func)</code>	This method invokes the specified function for each object that was removed from the collection.
<code>forEachIdentityChange(func)</code>	This method invokes the specified function for each object whose identity has changed.

The functions that are passed to the methods described in Table 14-4 will receive an `IterableChangeRecord` object that describes an item and how it has changed, using the properties shown in Table 14-5.

Table 14-5. The `IterableChangeRecord` Properties

Name	Description
<code>item</code>	This property returns the data item.
<code>trackById</code>	This property returns the identity value if a <code>trackBy</code> function is used.
<code>currentIndex</code>	This property returns the current index of the item in the collection.
<code>previousIndex</code>	This property returns the previous index of the item in the collection.

The code in Listing 14-16 only needs to deal with new objects in the data source since that is the only change that the rest of the application can perform. If the result of the `diff` method isn't `null`, then I use the `forEachAddedItem` method to invoke a fat arrow function for each new object that has been detected. The function is called once for each new object and uses the properties in Table 14-5 to create new views in the view container.

The changes in Listing 14-16 included a new console message that is written to the browser's JavaScript console only when there has been a data change detected by the directive. If you repeat the process of adding a new product, you will see that the message is displayed only when the application first starts and when the Create button is clicked. The `ngDoCheck` method is still being called, and the directive has to check for data changes every time, so there is still unnecessary work going on. But these operations are much less expensive and time-consuming than destroying and then re-creating HTML elements.

Keeping Track of Views

Handling change detection is simple when you are handling the creation of new data items. Other operations—such as dealing with deletions or modifications—are more complex and require the directive to keep track of which view is associated with which data object.

To demonstrate, I am going to add support for deleting a Product object from the data model. First, Listing 14-17 adds a method to the component to delete a product using its key. This isn't a requirement because the template could access the repository through the component's model property, but it can help make applications easier to understand when all of the data is accessed and used in the same way.

Listing 14-17. Adding a Delete Method in the component.ts File in the src/app Folder

```
import { Component } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";

@Component({
  selector: "app",
  templateUrl: "template.html"
})
export class ProductComponent {
  model: Model = new Model();
  showTable: boolean = false;

  getProduct(key: number): Product | undefined {
    return this.model.getProduct(key);
  }

  getProducts(): Product[] {
    return this.model.getProducts();
  }

  newProduct: Product = new Product();

  addProduct(p: Product) {
    this.model.saveProduct(p);
  }

  deleteProduct(key: number) {
    this.model.deleteProduct(key);
  }

  submitForm() {
    this.addProduct(this.newProduct);
  }
}
```

Listing 14-18 updates the template so that the content generated by the structural directive contains a column of button elements that will delete the data object associated with the row that contains it.

Listing 14-18. Adding a Delete Button in the template.html File in the src/app Folder

```
...
<table class="table table-sm table-bordered table-striped">
  <thead>
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th><th></th></tr>
  </thead>
  <tbody>
    <tr *paFor="let item of getProducts(); let i = index; let odd = odd;
          let even = even" [class.table-info]="odd"
             [class.table-warning]="even" class="align-middle">
      <td>{{i + 1}}</td>
      <td>{{item.name}}</td>
      <td>{{item.category}}</td>
      <td>{{item.price}}</td>
      <td class="text-center">
        <button class="btn btn-danger btn-sm"
               (click)="deleteProduct(item.id)">
          Delete
        </button>
      </td>
    </tr>
  </tbody>
</table>
...

```

The button elements have click event bindings that call the component's deleteProduct method. I also assigned the tr element to the align-middle class so the text in the table cells is aligned with the button text. The final step is to process the data changes in the structural directive so that it responds when an object is removed from the data source, as shown in Listing 14-19.

Listing 14-19. Responding to a Removed Item in the iterator.directive.ts File in the src/app Folder

```
import { Directive, ViewContainerRef, TemplateRef, Input,
         IterableDiffer, IterableDiffers, ChangeDetectorRef, IterableChangeRecord,
         ViewRef } from "@angular/core";

@Directive({
  selector: "[paForOf]"
})
export class PaIteratorDirective {
  private differ: IterableDiffer<any> | undefined;
  private views: Map<any, PaIteratorContext> = new Map<any, PaIteratorContext>();

  constructor(private container: ViewContainerRef,
             private template: TemplateRef<Object>,
             private differs: IterableDiffers,
             private changeDetector: ChangeDetectorRef) { }

  @Input("paForOf")
  dataSource: any;
}
```

```

ngOnInit() {
    this.differ =
        <IterableDiffer<any>> this.differs.find(this.dataSource).create();
}

ngDoCheck() {
    let changes = this.differ?.diff(this.dataSource);
    if (changes != null) {
        let arr: IterableChangeRecord<any>[] = [];
        changes.forEachAddedItem(addition => arr.push(addition));
        arr.forEach(addition => {
            if (addition.currentIndex != null) {
                let context = new PaIteratorContext(addition.item,
                    addition.currentIndex, arr.length);
                context.view = this.container.createEmbeddedView(this.template,
                    context);
                this.views.set(addition.trackById, context);
            }
        });
    let removals = false;
    changes.forEachRemovedItem(removal => {
        removals = true;
        let context = this.views.get(removal.trackById);
        if (context != null && context.view != null) {
            this.container.remove(this.container.indexOf(context.view));
            this.views.delete(removal.trackById);
        }
    });
    if (removals) {
        let index = 0;
        this.views.forEach(context =>
            context.setData(index++, this.views.size));
    }
}
}

class PaIteratorContext {
    index: number = 0;
    odd: boolean = false; even: boolean = false;
    first: boolean = false; last: boolean = false;
    view: ViewRef | undefined;

    constructor(public $implicit: any,
        public position: number, total: number ) {
        this.setData(position, total);
    }

    setData(index: number, total: number) {
        this.index = index;
        this.odd = index % 2 == 1;
    }
}

```

```

    this.even = !this.odd;
    this.first = index == 0;
    this.last = index == total - 1;
}
}
}

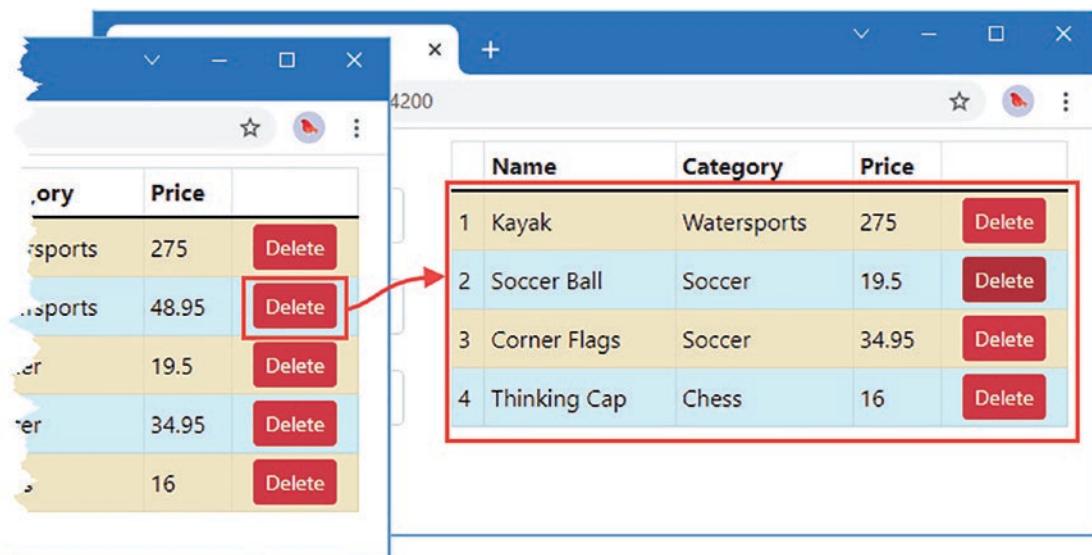
```

Two tasks are required to handle removed objects. The first task is updating the set of views by removing the ones that correspond to the items provided by the `forEachRemovedItem` method. This means keeping track of the mapping between the data objects and the views that represent them, which I have done by adding a `ViewRef` property to the `PaIteratorContext` class and using a `Map` to collect them, indexed by the value of the `IterableChangeRecord.trackById` property.

When processing the collection changes, the directive handles each removed object by retrieving the corresponding `PaIteratorContext` object from the `Map`, getting its `ViewRef` object, and passing it to the `ViewContainerRef.remove` element to remove the content associated with the object from the view container.

The second task is to update the context data for those objects that remain so that the bindings that rely on a view's position in the view container are updated correctly. The directive calls the `PaIteratorContext.setData` method for each context object left in the `Map` to update the view's position in the container and to update the total number of views that are in use. Without these changes, the properties provided by the context object wouldn't accurately reflect the data model, which means the background colors for the rows wouldn't be striped and the Delete buttons wouldn't target the right objects.

The effect of these changes is that each table row contains a Delete button that removes the corresponding object from the data model, which in turn triggers an update of the table, as shown in Figure 14-7.



The screenshot shows a web interface with two main components: a sidebar on the left and a main content area on the right.

Sidebar: It contains a table with columns "Category" and "Price". The data rows are: Watersports (275), Soccer (48.95), Soccer (19.5), Soccer (34.95), and Chess (16). Each row has a "Delete" button at the end.

Main Content Area: It contains a table with columns "Name", "Category", and "Price". The data rows are: 1. Kayak (Watersports, 275), 2. Soccer Ball (Soccer, 19.5), 3. Corner Flags (Soccer, 34.95), and 4. Thinking Cap (Chess, 16). Each row has a "Delete" button at the end. A red box highlights the "Delete" button for the second row (Soccer Ball).

Figure 14-7. Removing objects from the data model

Querying the Host Element Content

Directives can query the contents of their host element to access the directives it contains, known as the *content children*, which allows directives to coordinate themselves to work together.

Tip Directives can also work together by sharing services, which I describe in Chapter 17.

To demonstrate how content can be queried, I added a file called `cellColor.directive.ts` to the `src/app` folder and used it to define the directive shown in Listing 14-20.

Listing 14-20. The Contents of the `cellColor.directive.ts` File in the `src/app` Folder

```
import { Directive, HostBinding } from "@angular/core";

@Directive({
  selector: "td"
})
export class PaCellColor {

  @HostBinding("class")
  bgClass: string = "";

  setColor(dark: Boolean) {
    this.bgClass = dark ? "table-dark" : "";
  }
}
```

The `PaCellColor` class defines a simple attribute directive that operates on `td` elements and that binds to the `class` property of the host element. The `setColor` method accepts a Boolean parameter that, when the value is `true`, sets the `class` property to `table-dark`, which is the Bootstrap class for a dark background.

The `PaCellColor` class will be the directive that is embedded in the host element's content in this example. The goal is to write another directive that will query its host element to locate the embedded directive and invoke its `setColor` method. To that end, I added a file called `cellColorSwitcher.directive.ts` to the `src/app` folder and used it to define the directive shown in Listing 14-21.

Listing 14-21. The Contents of the `cellColorSwitcher.directive.ts` File in the `src/app` Folder

```
import { Directive, Input, SimpleChanges, ContentChild } from "@angular/core";
import { PaCellColor } from "./cellColor.directive";

@Directive({
  selector: "table"
})
export class PaCellColorSwitcher {

  @Input("paCellDarkColor")
  modelProperty: Boolean | undefined;

  @ContentChild(PaCellColor)
  contentChild: PaCellColor | undefined;
}
```

```

ngOnChanges(changes: SimpleChanges ) {
    if (this.contentChild != null) {
        this.contentChild.setColor(changes["modelProperty"].currentValue);
    }
}
}

```

The `PaCellColorSwitcher` class defines a directive that operates on `table` elements and that defines an input property called `paCellDarkColor`. The important part of this directive is the `contentChild` property.

```

...
@ContentChild(PaCellColor)
contentChild: PaCellColor | undefined;
...

```

The `@ContentChild` decorator tells Angular that the directive needs to query the host element's content and assign the first result of the query to the property. The argument to the `@ContentChild` director is one or more directive classes. In this case, the argument to the `@ContentChild` decorator is `PaCellColor`, which tells Angular to locate the first `PaCellColor` object contained within the host element's content and assign it to the decorated property.

Tip You can also query using template variable names, such that `@ContentChild("myVariable")` will find the first directive that has been assigned to `myVariable`.

The query result provides the `PaCellColorSwitcher` directive with access to the child component and allows it to call the `setColor` method in response to changes to the input property.

Tip If you want to include the descendants of children in the results, then you can configure the query, like this: `@ContentChild(PaCellColor, { descendants: true})`.

In Listing 14-22, I altered the checkbox in the template so it uses the `ngModel` directive to set a variable that is bound to the `PaCellColorSwitcher` directive's input property.

Listing 14-22. Applying the Directives in the template.html File in the src/app Folder

```

...
<div class="col">

    <div class="form-check">
        <label class="form-check-label">Dark Cell Color</label>
        <input type="checkbox" class="form-check-input" [(ngModel)]="darkColor" />
    </div>

    <table class="table table-sm table-bordered table-striped"
        [paCellDarkColor]="darkColor">
        <thead>
            <tr><th></th><th>Name</th><th>Category</th><th>Price</th><th></th></tr>
        </thead>

```

```

<tbody>
  <tr *paFor="let item of getProducts(); let i = index; let odd = odd;
       let even = even" [class.table-info]="odd"
          [class.table-warning]="even" class="align-middle">
    <td>{{i + 1}}</td>
    <td>{{item.name}}</td>
    <td>{{item.category}}</td>
    <td>{{item.price}}</td>
    <td class="text-center">
      <button class="btn btn-danger btn-sm"
             (click)="deleteProduct(item.id)">
        Delete
      </button>
    </td>
  </tr>
</tbody>
</table>
</div>
...

```

Listing 14-23 adds the `darkColor` property to the component.

Listing 14-23. Defining a Property in the component.ts File in the src/app Folder

```

import { Component } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";

@Component({
  selector: "app",
  templateUrl: "template.html"
})
export class ProductComponent {
  model: Model = new Model();
  showTable: boolean = false;
  darkColor: boolean = false;

  getProduct(key: number): Product | undefined {
    return this.model.getProduct(key);
  }

  getProducts(): Product[] {
    return this.model.getProducts();
  }

  newProduct: Product = new Product();

  addProduct(p: Product) {
    this.model.saveProduct(p);
  }
}

```

```

    deleteProduct(key: number) {
        this.model.deleteProduct(key);
    }

    submitForm() {
        this.addProduct(this.newProduct);
    }
}

```

The final step is to register the new directives with the Angular module's declarations property, as shown in Listing 14-24.

Listing 14-24. Registering New Directives in the app.module.ts File in the src/app Folder

```

import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';
import { BrowserAnimationsModule } from '@angular/platform-browser/animations';

import { ProductComponent } from './component';
import { FormsModule, ReactiveFormsModule } from "@angular/forms";
import { PaAttrDirective } from "./attr.directive";
import { PaModel } from "./twoway.directive";
import { PaStructureDirective } from "./structure.directive";
import { PaIteratorDirective } from "./iterator.directive";
import { PaCellColor } from "./cellColor.directive";
import { PaCellColorSwitcher } from "./cellColorSwitcher.directive";

@NgModule({
    declarations: [ProductComponent, PaAttrDirective, PaModel,
        PaStructureDirective, PaIteratorDirective,
        PaCellColor, PaCellColorSwitcher],
    imports: [
        BrowserModule,
        BrowserAnimationsModule,
        FormsModule, ReactiveFormsModule
    ],
    providers: [],
    bootstrap: [ProductComponent]
})
export class AppModule { }

```

When you save the changes, you will see a new checkbox above the table. When you check the box, the `ngModel` directive will cause the `PaCellColorSwitcher` directive's `input` property to be updated, which will call the `setColor` method of the `PaCellColor` directive object that was found using the `@ContentChild` decorator. The visual effect is small because only the first `PaCellColor` directive is affected, which is the cell that displays the number 1, at the top-left corner of the table, as shown in Figure 14-8. (If you don't see the color change, then restart the Angular development tools and reload the browser.)

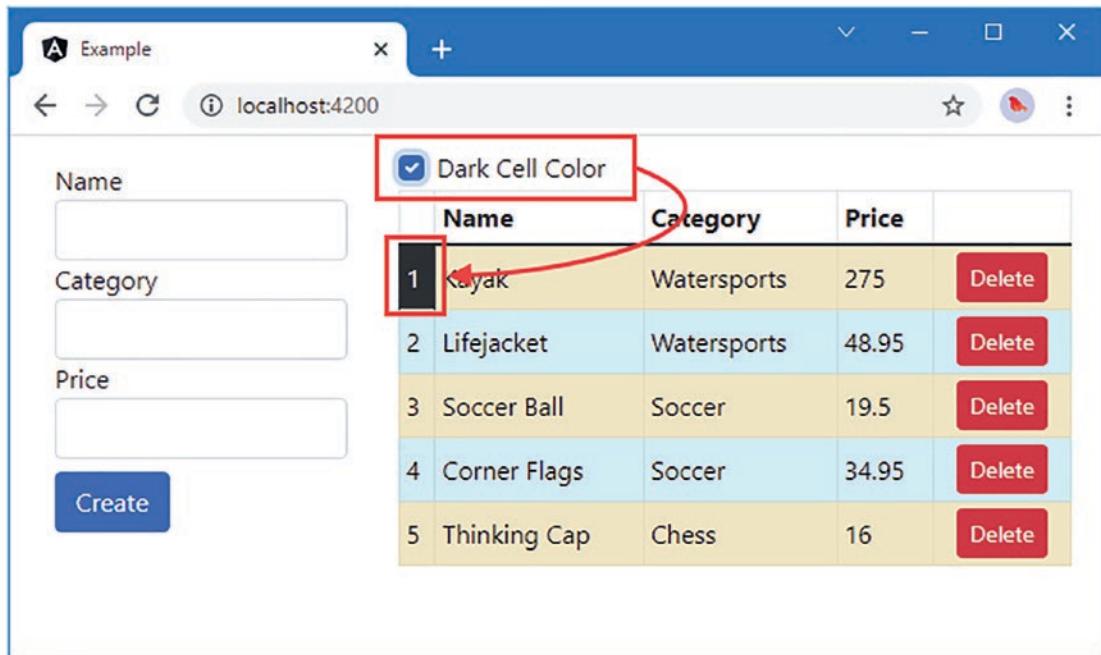


Figure 14-8. Operating on a content child

Querying Multiple Content Children

The `@ContentChild` decorator finds the first directive object that matches the argument and assigns it to the decorated property. If you want to receive all the directive objects that match the argument, then you can use the `@ContentChildren` decorator instead, as shown in Listing 14-25.

Listing 14-25. Querying Multiple Children in the `cellColorSwitcher.directive.ts` File in the `src/app` Folder

```
import { Directive, Input, SimpleChanges, ContentChildren, QueryList }
  from "@angular/core";
import { PaCellColor } from "./cellColor.directive";

@Directive({
  selector: "table"
})
export class PaCellColorSwitcher {

  @Input("paCellDarkColor")
  modelProperty: Boolean | undefined;

  @ContentChildren(PaCellColor, {descendants: true})
  contentChildren: QueryList<PaCellColor> | undefined;
```

```

ngOnChanges(changes: SimpleChanges) {
  this.updateContentChildren(changes["modelProperty"].currentValue);
}

private updateContentChildren(dark: Boolean) {
  if (this.contentChildren != null && dark != undefined) {
    this.contentChildren.forEach((child, index) => {
      child.setColor(index % 2 ? dark : !dark);
    });
  }
}
}

```

When you use the `@ContentChildren` decorator, the results of the query are provided through a `QueryList`, which provides access to the directive objects using the methods and properties described in Table 14-6. The `descendants` configuration property is used to select descendant elements, and without this value, only direct children are selected.

Table 14-6. The `QueryList` Members

Name	Description
<code>length</code>	This property returns the number of matched directive objects.
<code>first</code>	This property returns the first matched directive object.
<code>last</code>	This property returns the last matched directive object.
<code>map(function)</code>	This method calls a function for each matched directive object to create a new array, equivalent to the <code>Array.map</code> method.
<code>filter(function)</code>	This method calls a function for each matched directive object to create an array containing the objects for which the function returns true, equivalent to the <code>Array.filter</code> method.
<code>reduce(function)</code>	This method calls a function for each matched directive object to create a single value, equivalent to the <code>Array.reduce</code> method.
<code>forEach(function)</code>	This method calls a function for each matched directive object, equivalent to the <code>Array.forEach</code> method.
<code>some(function)</code>	This method calls a function for each matched directive object and returns <code>true</code> if the function returns <code>true</code> at least once, equivalent to the <code>Array.some</code> method.
<code>changes</code>	This property is used to monitor the results for changes, as described in the upcoming “Receiving Query Change Notifications” section.

In the listing, the directive responds to changes in the input property value by calling the `updateContentChildren` method, which in turn uses the `forEach` method on the `QueryList` and invokes the `setColor` method on every second directive that has matched the query. Figure 14-9 shows the effect when the checkbox is selected.

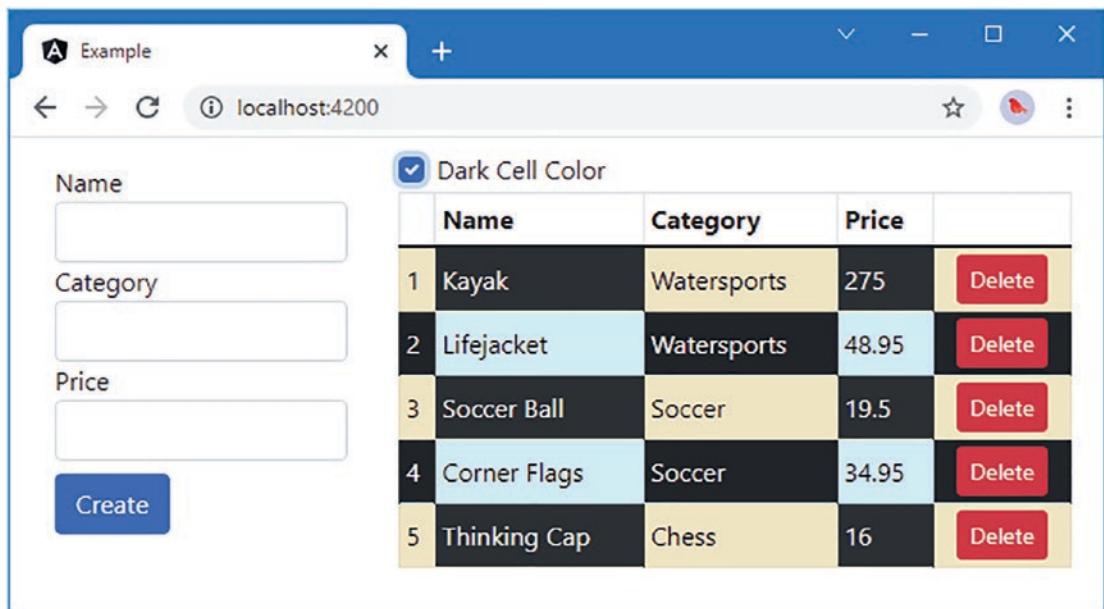


Figure 14-9. Operating on multiple content children

Receiving Query Change Notifications

The results of content queries are live, meaning that they are automatically updated to reflect additions, changes, or deletions in the host element's content. Receiving a notification when there is a change in the query results requires using the Observable interface, which is provided by the Reactive Extensions package, as described in Chapter 4.

In Listing 14-26, I have updated the PaCellColorSwitcher directive so that it receives notifications when the set of content children in the QueryList changes.

Listing 14-26. Receiving Notifications in the cellColorSwitcher.directive.ts File in the src/app Folder

```
import { Directive, Input, SimpleChanges, ContentChildren, QueryList } from "@angular/core";
import { PaCellColor } from "./cellColor.directive";

@Directive({
  selector: "table"
})
export class PaCellColorSwitcher {

  @Input("paCellDarkColor")
  modelProperty: Boolean | undefined;

  @ContentChildren(PaCellColor, {descendants: true})
  contentChildren: QueryList<PaCellColor> | undefined;

  ngOnChanges(changes: SimpleChanges) {
    this.updateContentChildren(changes["modelProperty"].currentValue);
  }
}
```

```

ngAfterContentInit() {
  if (this.modelProperty != undefined) {
    this.contentChildren?.changes.subscribe(() => {
      this.updateContentChildren(this.modelProperty as Boolean);
    });
  }
}

private updateContentChildren(dark: Boolean) {
  if (this.contentChildren != null && dark != undefined) {
    this.contentChildren.forEach((child, index) => {
      child.setColor(index % 2 ? dark : !dark);
    });
  }
}
}

```

The value of a content child query property isn't set until the `ngAfterContentInit` lifecycle method is invoked, so I use this method to set up the change notification. The `QueryList` class defines a `changes` method that returns a Reactive Extensions `Observable` object, which defines a `subscribe` method. This method accepts a function that is called when the contents of the `QueryList` change, meaning that there is some change in the set of directives matched by the argument to the `@ContentChildren` decorator. The function that I passed to the `subscribe` method calls the `updateContentChildren` method to set the colors.

The result of these changes is that the dark coloring is automatically applied to new table cells that are created when the HTML form is used, as shown in Figure 14-10.

The screenshot shows a web browser window titled "Example" at "localhost:4200". On the left, there is a form with three input fields: "Name", "Category", and "Price", each with a corresponding text input box below it. Below the inputs is a blue "Create" button. To the right of the form is a table titled "Dark Cell Color". The table has a header row with columns "Name", "Category", "Price", and an empty column. There are five data rows, each containing a number (1 through 5), a product name, its category, its price, and a red "Delete" button. The rows alternate in color between light orange and light blue. A checkbox labeled "Dark Cell Color" is checked above the table.

	Name	Category	Price	
1	Kayak	Watersports	275	Delete
2	Lifejacket	Watersports	48.95	Delete
3	Soccer Ball	Soccer	19.5	Delete
4	Corner Flags	Soccer	34.95	Delete
5	Thinking Cap	Chess	16	Delete

Figure 14-10. Acting on content query change notifications

Summary

In this chapter, I explained how structural directives work by re-creating the functionality of the built-in `ngIf` and `ngFor` directives. I explained the use of view containers and templates, described the full and concise syntax for applying structural directives, and showed you how to create a directive that iterates over a collection of data objects and how directives can query the content of their host element. In the next chapter, I introduce components and explain how they differ from directives.