

Pro Angular 16

Adam Freeman



Pro Angular 16

SIXTH EDITION

Adam Freeman

For online information and ordering of this and other Manning books, please visit www.manning.com.

The publisher offers discounts on this book when ordered in quantity. For more information, please contact

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

©2024 by Manning Publications Co. All rights Reserved.

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

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

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

The author and publisher have made every effort to ensure that the information in this book was correct at press time. The author and publisher do not assume and hereby disclaim any liability to any party for any loss, damage, or disruption caused by errors or omissions, whether such errors or omissions result from negligence, accident, or any other cause, or from any usage of the information herein.



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

Development editor:
Production editor:
Cover designer:

Ian Hough
Aleksandar Dragosavljević
Marija Tudor

ISBN 9781633436695

Printed in the United States of America

contents

<i>preface</i>	<i>vi</i>
<i>about this book</i>	<i>vii</i>
<i>about the author</i>	<i>ix</i>
<i>about the cover illustration</i>	<i>x</i>

Chapter 1 Getting Ready	1
-------------------------	---

PART 1: GETTING STARTED WITH ANGULAR 9

Chapter 2 Jumping right in	10
Chapter 3 Primer, Part 1	43
Chapter 4 Primer, Part 2	71
Chapter 5 SportsStore: a real application	92
Chapter 6 SportsStore: orders and checkout	121
Chapter 7 SportsStore: administration	152
Chapter 8 SportsStore: deployment	192

PART 2: ANGULAR IN DETAIL 214

Chapter 9 Understanding Angular projects and tools	215
Chapter 10 Angular reactivity and signals	250
Chapter 11 Using Data Bindings	275
Chapter 12 Using the built-in directives	299
Chapter 13 Using events and forms	329
Chapter 14 Creating attribute directives	369

Chapter 15 Creating structural directives	399
Chapter 16 Understanding components	438
Chapter 17 Using and creating pipes	465
Chapter 18 Using services	511
Chapter 19 Using and creating modules	540
PART 3: ADVANCED ANGULAR FEATURES	567
Chapter 20 Creating the example project	568
Chapter 21 Using the forms API, part 1	581
Chapter 22 Using the forms API, part 2	615
Chapter 23 Making HTTP Requests	647
Chapter 24 Routing and navigation: part 1	669
Chapter 25 Routing and navigation: part 2	701
Chapter 26 Routing and navigation: part 3	725
Chapter 27 Optimizing application delivery	748
Chapter 28 Working with component libraries	780
Chapter 29 Angular unit testing	807

index 832

preface

Thank you for purchasing *Pro Angular 16*. This is the 6th edition of this book, and the first to be published by Manning, and I am delighted that it is ready for the publication.

Angular has become one of the most popular web application frameworks by balancing innovation with stability and consistency. After a turbulent transition from the original AngularJS, the Angular of recent years has been focused on providing a robust set of features that have evolved gradually.

Angular 16 introduces *signals*, which alters the way that changes in data are detected. This book explains how signals work and demonstrates their use, setting the foundation for the next generation of Angular functionality that will be in Angular 17.

My goal is that you will become familiar with every important Angular feature and be equipped to choose the ones that best suit your projects. I appreciate feedback and you can raise issues and ask questions using liveBook Discussion forum or using the email address given in the book.

about this book

This book is for experienced web developers who are new to Angular. It doesn't explain the basics of web applications or programming. 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.

2.1 *How this book is organized—a roadmap*

This book is divided into three parts, each of which delves into a set of related topics.

In Part 1, titled "Getting started with Angular," you'll find all the information you need to prepare for the rest of the book. It includes primers and refreshers for critical technologies such as HTML and TypeScript, which is a superset of JavaScript used in Angular development. Additionally, you'll learn how to build your first Angular application and take a step-by-step approach to building a more realistic application called SportsStore.

Part 2 of the book, "Angular in detail," covers the building blocks provided by Angular for creating applications. You'll work through each of these in turn and learn about Angular's built-in functionality, as well as endless customization options.

Finally, in Part 3, "Advanced Angular features," you'll discover how to use advanced features to create more complex and scalable applications. You'll learn how to make asynchronous HTTP requests in an Angular application and explore other advanced features.

2.2 *About the code*

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

In many cases, the original source code has been reformatted; we've added line breaks and reworked indentation to accommodate the available page space in the book. In rare cases, even this was not enough, and listings include line-continuation markers (↵). Additionally, comments in the source code have often been removed from the listings when the code is described in the text. Code annotations accompany many of the listings, highlighting important concepts.

You can get executable snippets of code from the liveBook (online) version of this book at <https://livebook.manning.com/book/pro-angular-16>. The complete code for the examples in the book is available for download from the Manning website at <https://www.manning.com/books/pro-angular-16>, and from GitHub at <https://github.com/manningbooks/pro-angular-16>.

2.3 *liveBook discussion forum*

Purchase of *Pro Angular 16* includes free access to liveBook, Manning's online reading platform. Using liveBook's exclusive discussion features, you can attach comments to the book globally or to specific sections or paragraphs. It's a snap to make notes for yourself, ask and answer technical questions, and receive help from the author and other users. To access the forum, go to <https://livebook.manning.com/book/pro-angular-16/discussion>. You can also learn more about Manning's forums and the rules of conduct at <https://livebook.manning.com/discussion>.

Manning's commitment to our readers is to provide a venue where a meaningful dialogue between individual readers and between readers and the author can take place. It is not a commitment to any specific amount of participation on the part of the author, whose contribution to the forum remains voluntary (and unpaid). We suggest you try asking the author some challenging questions lest his interest stray! The forum and the archives of previous discussions will be accessible from the publisher's website as long as the book is in print.

about the author



Adam Freeman is an experienced IT professional who started his career a programmer. He has held senior positions in a range of companies, most recently serving as Chief Technology Officer and Chief Operating Officer of a global bank. He has written 49 programming books, focusing mostly on web application development. Now retired, he spends his time writing and trying to make furniture.

about the cover illustration

The figure on the cover of *Pro Angular 16*, titled “Jésuite,” or “Jesuit,” is taken from a book by Louis Curmer published in 1841. Each illustration is finely drawn and colored by hand.

In those days, it was easy to identify where people lived and what their trade or station in life was just by their dress. Manning celebrates the inventiveness and initiative of the computer business with book covers based on the rich diversity of regional culture centuries ago, brought back to life by pictures from collections such as this one.

1

Getting ready

This chapter covers

- Understanding the purpose of Angular
- Understanding the contents of this book.
- Reporting errors in this book.
- Contacting the author.

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 will show you everything you need to know to get the benefits of Angular in your projects.

1.1 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*.

1.1.1 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.

1.2 Comparing Angular to React

The main competitor to Angular is React. There are some low-level differences between them but both frameworks are excellent, they work in similar ways, and both 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. React doesn't require TypeScript (although it is supported) and leans 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 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.

1.3 What do you need to know?

Before reading this book, you should be familiar with the basics of web development, understand 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.

1.4 What is the structure of this book?

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

1.4.1 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 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.

1.4.2 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 and provides endless customization options.

1.4.3 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 use component libraries.

1.5 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. 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.

1.6 What software do you need for Angular?

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.

1.7 How do you 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.

1.8 What if you 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 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. Next, check the typos list, also in the GitHub repository, which contains corrections for issues that are unlikely to cause confusion or break the examples.

If you still have problems, then download the project for the chapter you are reading from the book's GitHub repository, <https://github.com/manningbooks/pro-angular-16>, 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. Please remember that I get a lot of emails and that I may not respond immediately.

1.9 What if you 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 and typos lists, which you can find in the book's GitHub repository at <https://github.com/manningbooks/pro-angular-16>, in case it has already been reported.

Errata bounty

Manning 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 Manning ebook, 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 Manning 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. Or, 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.

1.10 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 a later chapter. 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);
  }
}
...
```

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 } from "@angular/core";
import { FormArray, FormControl, FormGroup, NgForm, 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 } from "../sharedState.service";
import { toObservable } from "@angular/core/rxjs-interop";

@Component({
  selector: "paForm",
  templateUrl: "form.component.html",
  styleUrls: ["form.component.css"]
})
export class FormComponent {

  // ...statements omitted for brevity...
```

```

    addKeywordControl() {
        this.keywordGroup.push(this.createKeywordFormControl());
    }

    removeKeywordControl(index: number) {
        this.keywordGroup.removeAt(index);
    }
}

```

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.

1.11 Where can you get the example code?

You can download the example projects for all the chapters in this book from <https://github.com/manningbooks/pro-angular-16>.

1.12 How do you 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.

1.13 What if you 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.

1.14 What if this book has made you angry?

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.

1.15 Summary

In this chapter, I briefly introduced Angular and outlined the content and structure of this book.

- Angular is a JavaScript framework used to create dynamic web applications.
- Angular applications are written in TypeScript, which is a superset of the JavaScript language.
- Angular is comparable to other web application frameworks, such as React.

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.

Part I

2

Jumping right in

This chapter covers

- Installing the tools and packages required for Angular development
- Creating an Angular project
- Using Angular features to dynamically create HTML
- Displaying data in the HTML content
- Responding to events
- Styling the HTML content using the Angular Material package

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.

2.1 *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.

2.1.1 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 18.14.0, 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 18.14.0 release for the examples in this book. A complete set of 18.14.0 releases, with installers for Windows and macOS and binary packages for other platforms, is available at <https://nodejs.org/dist/v18.14.0>.

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.

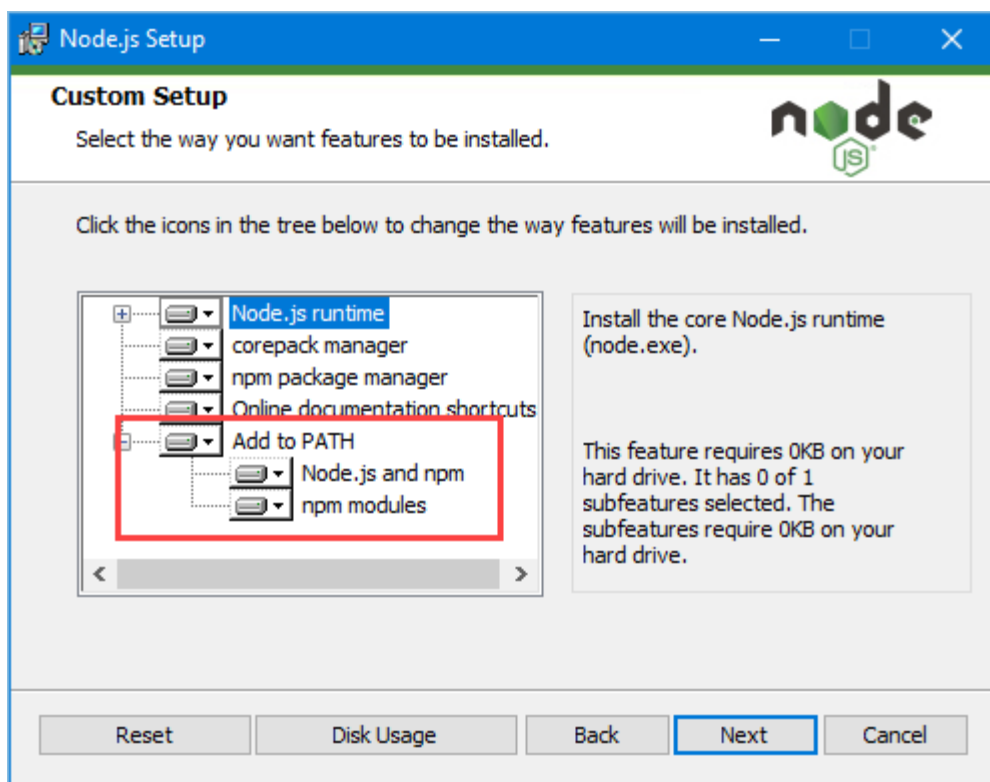


Figure 2.1. Installing Node.js

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:

```
v18.14.0
```

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:

```
8.1.4
```

2.1.2 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>.

2.1.3 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@16.0.0
```

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@16.0.0
```

2.1.4 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.

2.2 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 part 2). 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/manningbooks/pro-angular-16>. See chapter 1 for how to get help if you have problems running the examples.

2.2.1 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 part 2), 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.

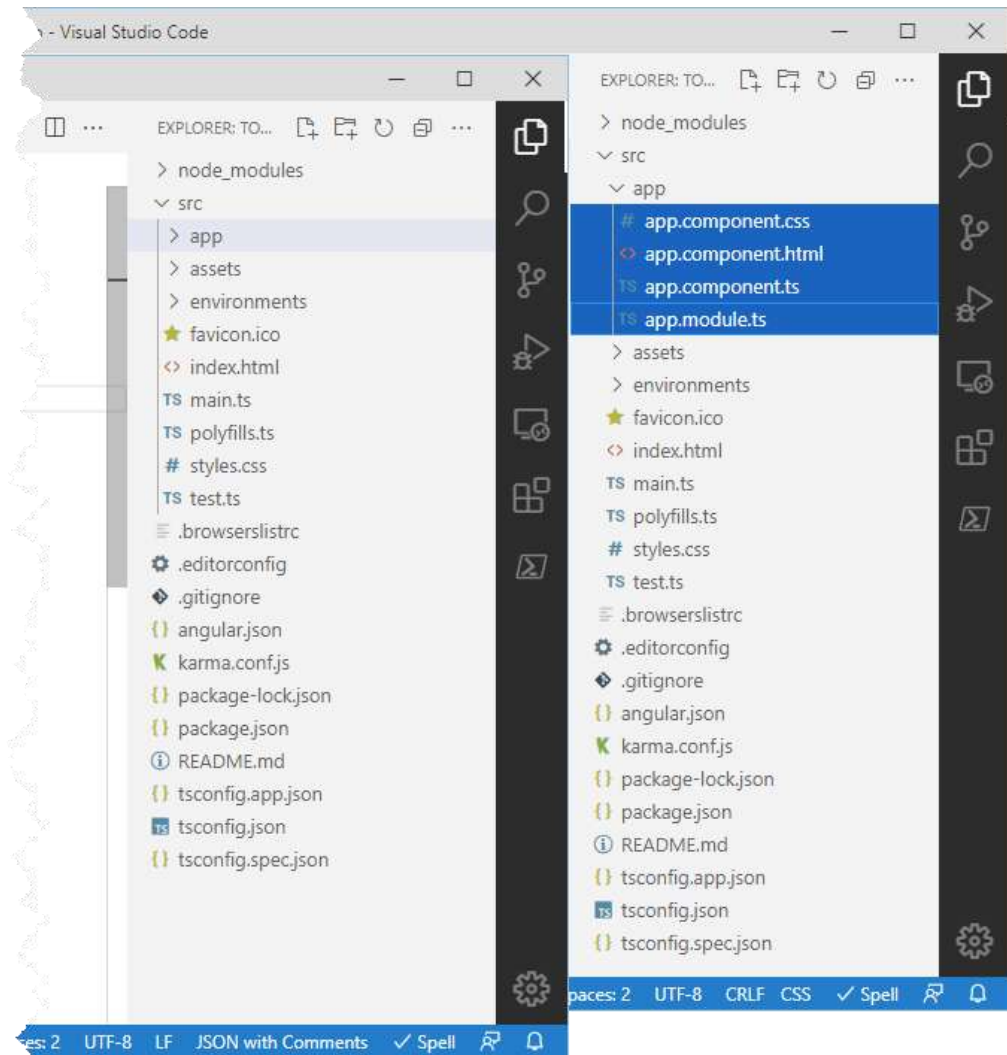


Figure 2.2. The initial contents of an Angular project

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.

2.2.2 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	Raw Size
vendor.js	vendor	1.94 MB
polyfills.js	polyfills	328.81 kB
styles.css, styles.js	styles	226.24 kB
main.js	main	45.98 kB
runtime.js	runtime	6.51 kB
	Initial Total	2.53 MB

```
Build at: 2023-07-20T07:42:19.998Z - Hash: 8b780a792e617175 - Time: 11498ms
```

```
** 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.

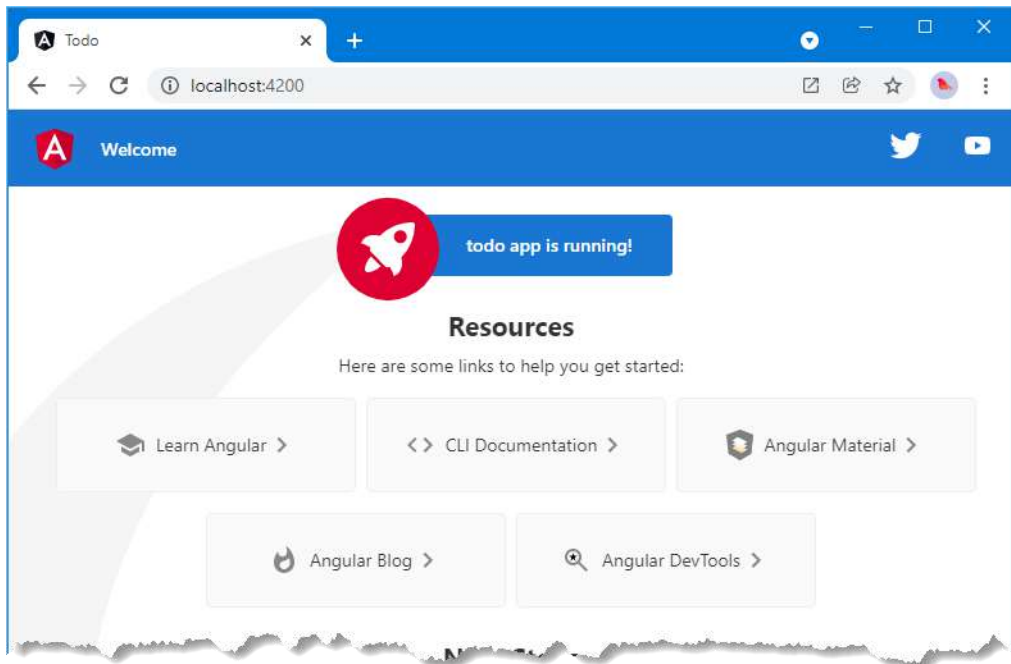


Figure 2.3. The placeholder content in a new Angular project

2.3 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.)

2.3.1 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 part 2.

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.

2.3.2 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 `h6` 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.

2.3.3 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 Angular feature that do 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 part 2 of this book.

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.

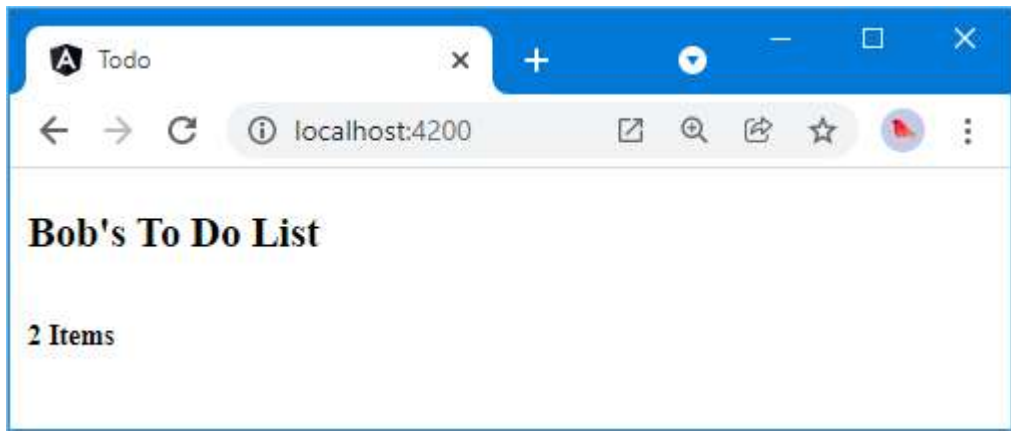


Figure 2.4. Generating content in the example application

2.4 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@16.0.0 --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 part 2. 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: [
    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/manningbooks/pro-angular-16>.

2.4.1 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.

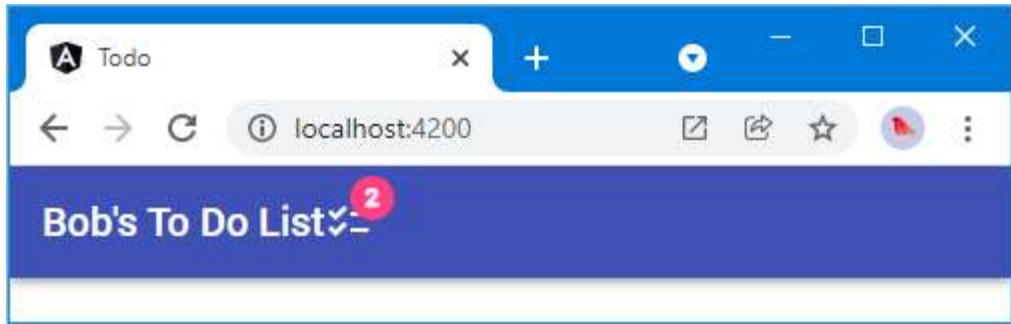


Figure 2.5. Introducing Angular Material components

I will 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.

2.4.2 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>
  <span>{{ username }}'s To Do List</span>
  <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.

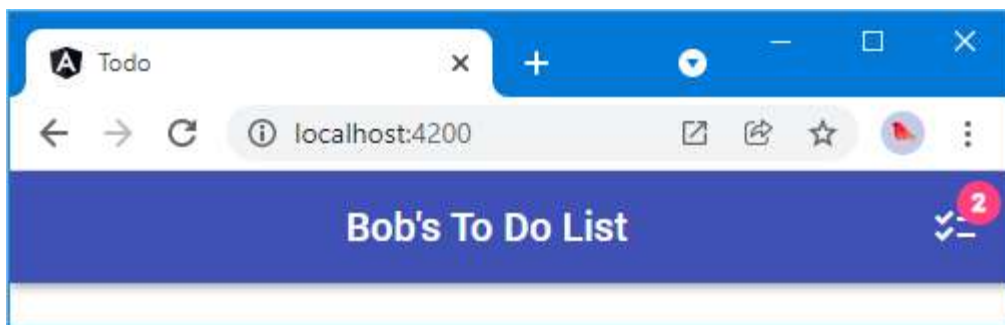


Figure 2.6. Adding spacers to the component layout

2.5 *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 a custom 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 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, 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 of the book, 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.

2.5.1 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.

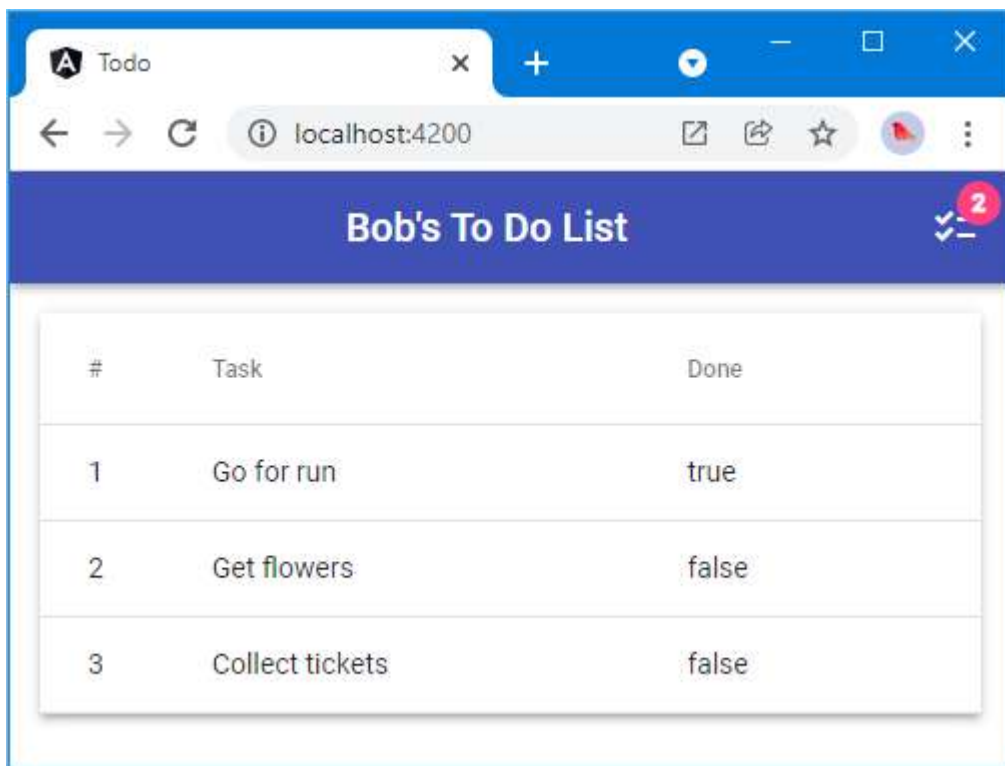


Figure 2.7. Displaying the list of to-do items

2.6 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.

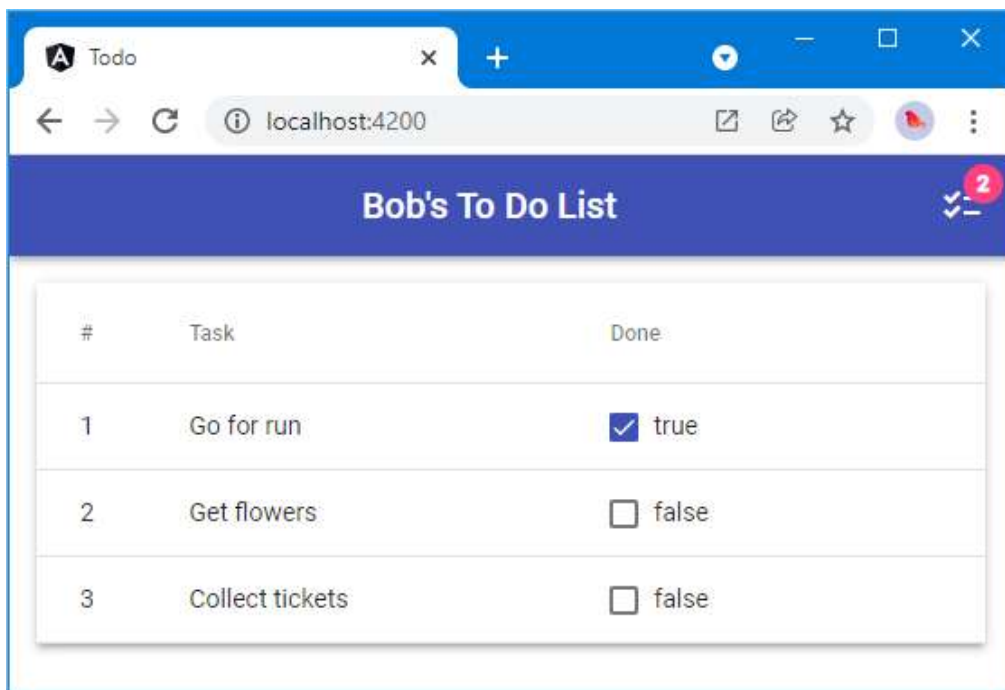


Figure 2.8. Using two-way bindings

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.

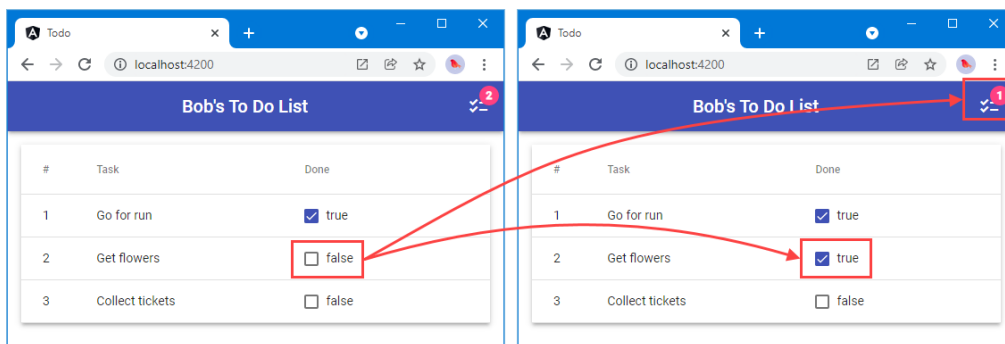


Figure 2.9. Toggling a checkbox

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 part 2, but this is a simple and effective way to understand what an application is doing.

2.7 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.

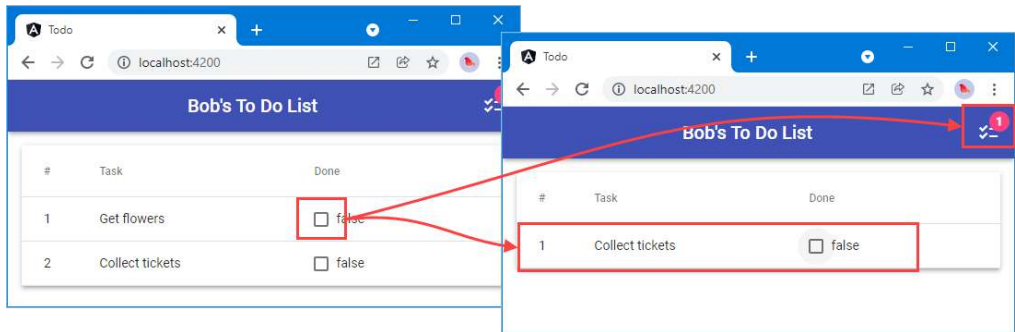


Figure 2.10. Filtering the to-do items

2.8 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.

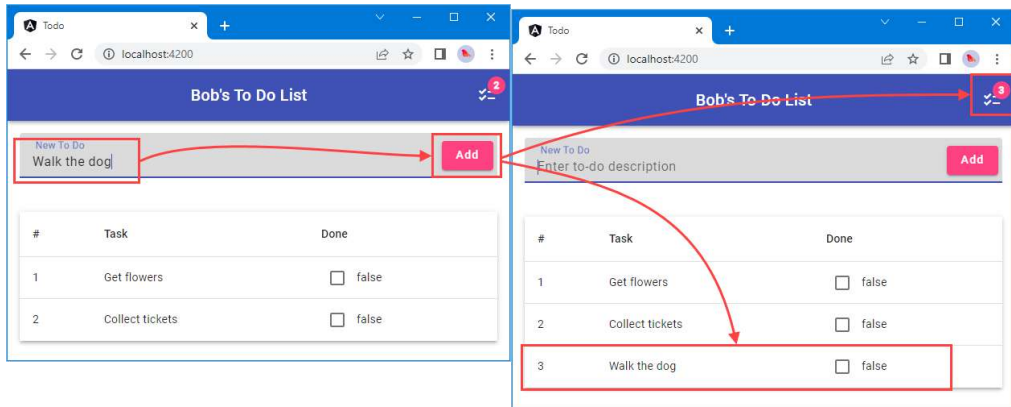


Figure 2.11. Creating a to-do item

2.9 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>

```

```

        </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;
}

```

An additional CSS style is required to lay out the toggle switch, as shown in listing 2.26.

Listing 2.26. Adding a style 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.

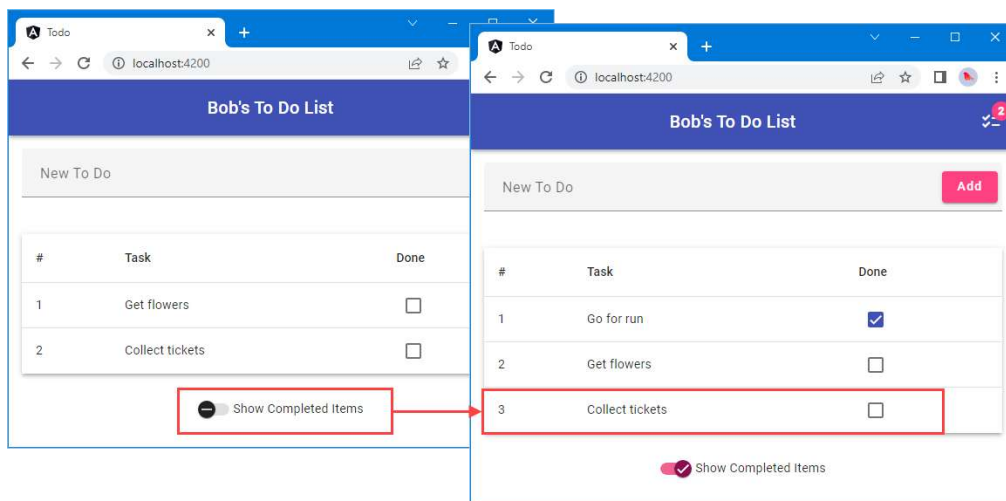


Figure 2.12. Showing completed tasks

2.10 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.

- Angular development uses open-source packages and tools, although there are many paid-for options for the code editor.
- Angular provides a tool package for creating new projects using a standard template.
- Angular generates content using components and templates, which are connected using data bindings and event bindings.
- Angular Material is an add-on package that is used to style content and is written using Angular features.

In the next chapter, I put Angular in context and describe the structure of this book.

3

Primer, part 1

This chapter covers

- Understanding the basic structure of HTML and the role of CSS
- Understanding the relationship between JavaScript and TypeScript
- Using TypeScript to make the JavaScript type system predictable
- Using the basic JavaScript/TypeScript types and operators

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.

3.1 *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/manningbooks/pro-angular-16>. 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@5.2.3
```

If you are using Linux or macOS, 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.Primer.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.

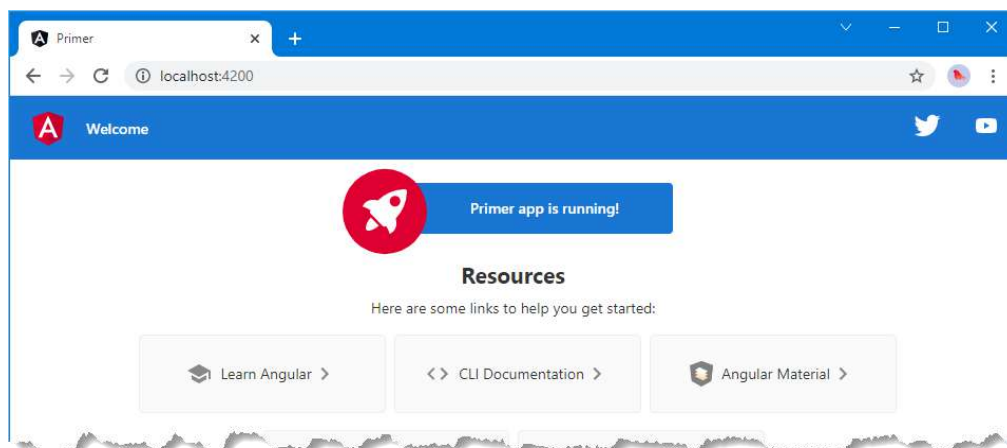


Figure 3.1. Running the example application

3.2 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.

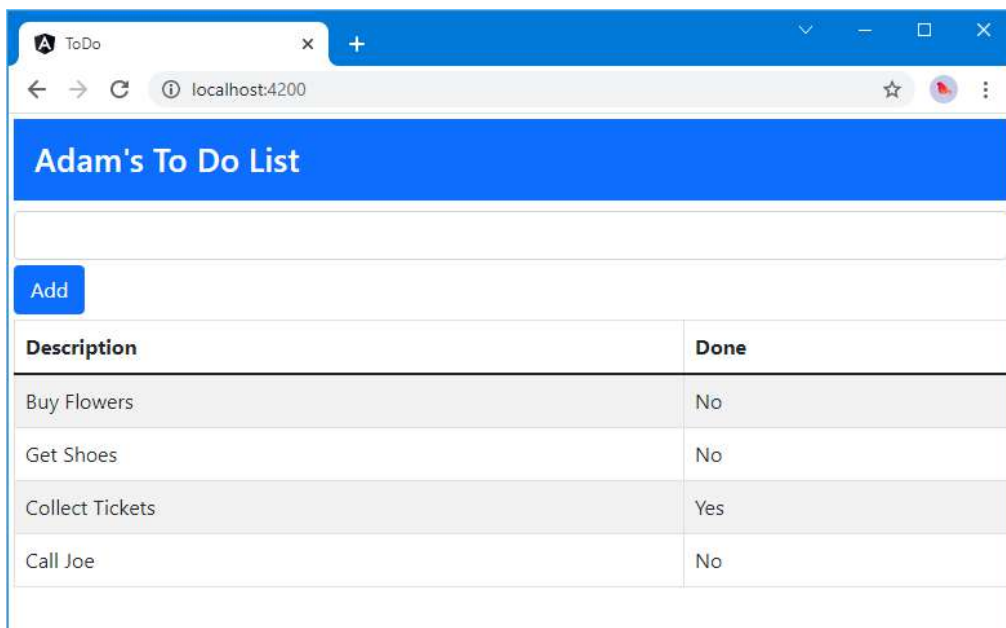


Figure 3.2. Understanding HTML

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.

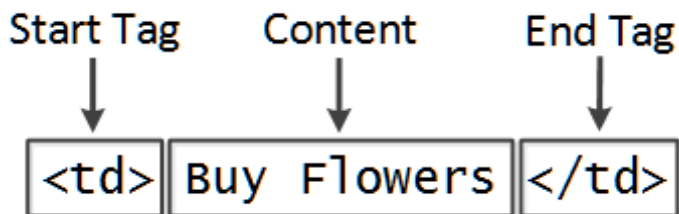


Figure 3.3. The anatomy of a simple HTML element

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.

3.2.1 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.

3.2.2 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.

3.2.3 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.

3.2.4 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.

3.2.5 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.

3.2.6 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

<code>meta</code>	Provides descriptive data about the document, such as the character encoding
<code>table</code>	Denotes a table, used to organize content into rows and columns
<code>tbody</code>	Denotes the body of the table (as opposed to the header or footer)
<code>td</code>	Denotes a content cell in a table row
<code>th</code>	Denotes a header cell in a table row
<code>thead</code>	Denotes the header of a table
<code>title</code>	Denotes the title of the document; used by the browser to set the title of the window or tab
<code>tr</code>	Denotes a row in a table

3.3 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 because they let me style the 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 won'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.

3.4 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.

3.4.1 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.

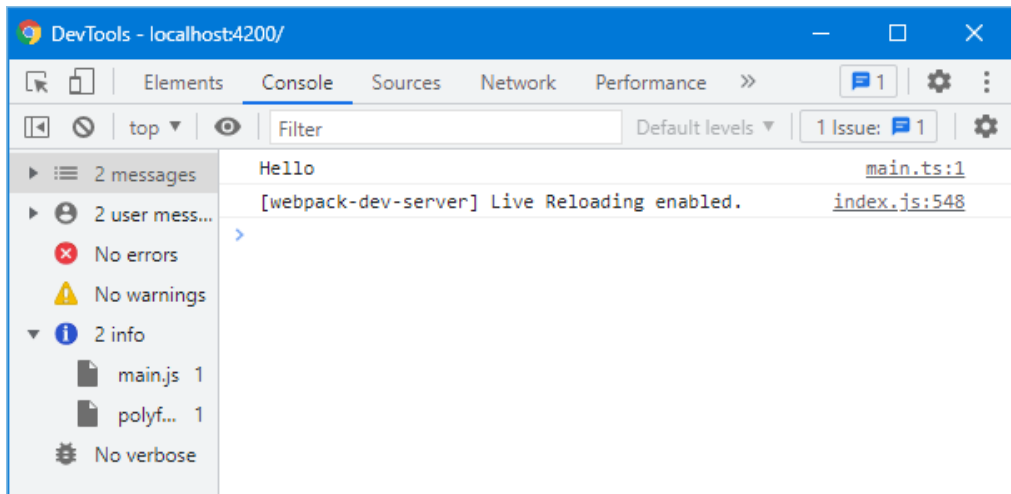


Figure 3.4. A message in the JavaScript console

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
```

3.4.2 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. 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:\example.js:6
    throw ("Expected a number: " + param)
    ^
Expected a number: London
(Use `node --trace-uncaught ...` 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.

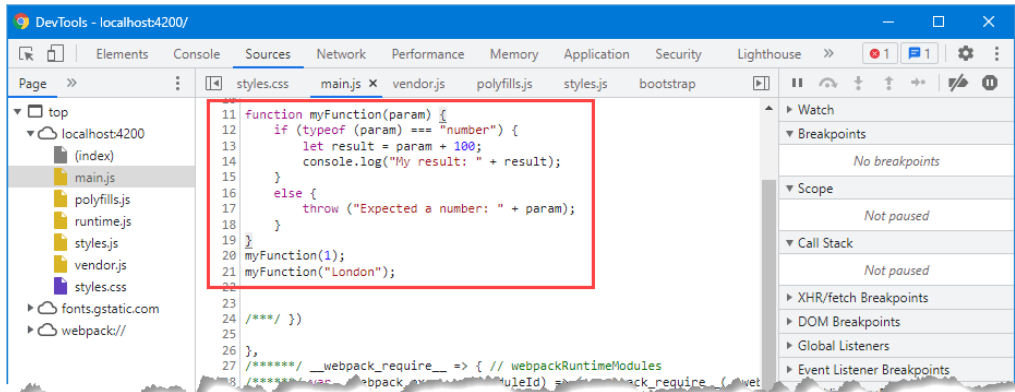


Figure 3.5. Examining the compiled code

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 `bigint` and `symbol` primitive types, but I don't use them in this book. The `bigint` 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 argument types 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:3:20 - 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
```

The `as` keyword should be used with caution because the TypeScript compiler is sophisticated and usually has a good 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:3:25 - 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

```

3.4.3 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.

3.4.4 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;

```

3.4.5 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.

3.4.6 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 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
<code>length</code>	This property returns the number of characters in the string.
<code>charAt(index)</code>	This method returns a string containing the character at the specified index.
<code>concat(string)</code>	This method returns a new string that concatenates the string on which the method is called and the string provided as an argument.
<code>indexOf(term, start)</code>	This method returns the first index at which <code>term</code> appears in the string or <code>-1</code> if there is no match. The optional <code>start</code> argument specifies the start index for the search.

<code>replace(term, newTerm)</code>	This method returns a new string in which all instances of <code>term</code> are replaced with <code>newTerm</code> .
<code>slice(start, end)</code>	This method returns a substring containing the characters between the start and end indices.
<code>split(term)</code>	This method splits up a string into an array of values that were separated by <code>term</code> .
<code>toUpperCase()</code> <code>toLowerCase()</code>	These methods return new strings in which all the characters are uppercase or lowercase.
<code>trim()</code>	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.

3.4.7 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>	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
toString()	This method returns a string that represents a number in base 10.
toString(2) toString(8) toString(16)	This method returns a string that represents a number in binary, octal, or hexadecimal notation.
toFixed(n)	This method returns a string representing a real number with the n digits after the decimal point.
toExponential(n)	This method returns a string that represents a number using exponential notation with one digit before the decimal point and n digits after.
toPrecision(n)	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 explained 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
```

3.5 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.

- Angular applications generate HTML content, which is displayed to the user by a web browser.
- Browsers execute JavaScript, which is a full-featured language with some features that often confuse.
- Angular applications are written in TypeScript, which is a superset of JavaScript, which makes development more consistent with other mainstream programming languages.
- TypeScript code is compiled into JavaScript so that it can be executed by browsers, which means that the compiler has to erase the TypeScript features from the code that is used by the browser.
- TypeScript doesn't alter the JavaScript type system but requires the developer to

indicate what data types are expected in code, which avoids many of the problems encountered by developers who are new to JavaScript.

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.

4

Primer, part 2

This chapter covers

- Understanding and using JavaScript functions
- Using the JavaScript concise function syntax
- Defining and using JavaScript arrays
- Creating objects using the literal syntax
- Using classes to create objects
- Defining and using JavaScript modules

In this chapter, I continue to describe the basic features of TypeScript and JavaScript that are required for Angular development.

4.1 *Preparing for this chapter*

This chapter uses the Primer project created in chapter 3. 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/manningbooks/pro-angular-16>. 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.

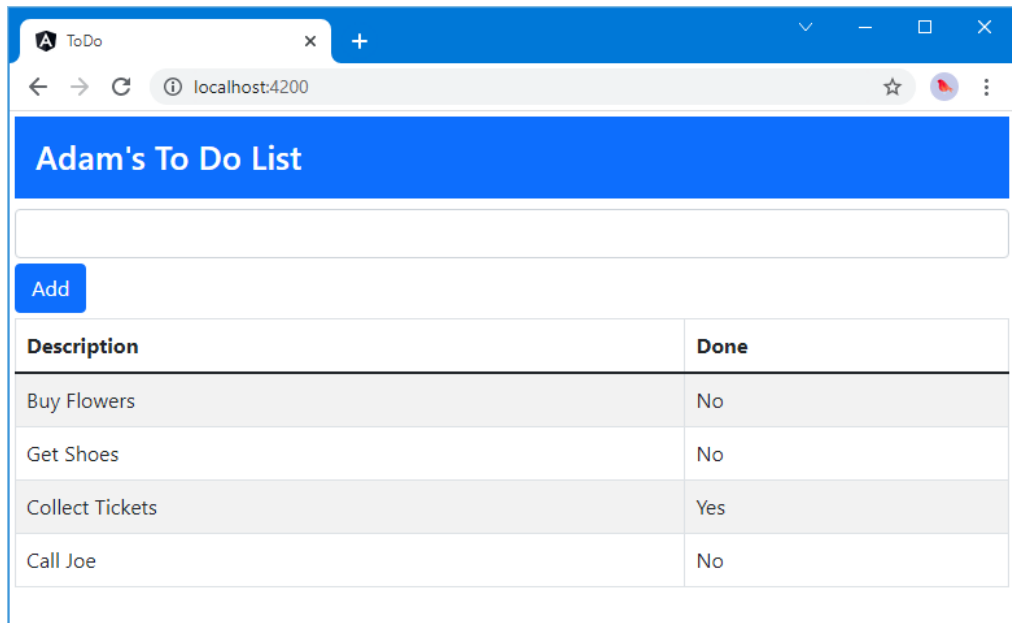


Figure 4.1. Running the example application

This chapter continues to use the browser's JavaScript console. Press F12 to open the browser's developer 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
```

4.2 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
```

4.2.1 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
```

4.2.2 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
```

4.2.3 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:
```

4.2.4 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
```

4.2.5 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
```

4.3 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 and 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.

4.3.1 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
```

4.3.2 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
```

4.3.3 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
```

4.3.4 Using the built-in array methods

JavaScript arrays define many 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 <code>count</code> items from the array, starting at the specified <code>index</code> . 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 <code>test</code> function for each item in the array and returns <code>true</code> if the function returns <code>true</code> for all of them and <code>false</code> otherwise.
<code>some(test)</code>	This method returns <code>true</code> if calling the <code>test</code> function for each item in the array returns <code>true</code> at least once.
<code>filter(test)</code>	This method returns a new array containing the items for which the <code>test</code> function returns <code>true</code> .
<code>find(test)</code>	This method returns the first item in the array for which the <code>test</code> function returns <code>true</code> .
<code>findIndex(test)</code>	This method returns the index of the first item in the array for which the <code>test</code> function returns <code>true</code> .
<code>foreach(callback)</code>	This method invokes the <code>callback</code> function for each item in the array, as described in the previous section.
<code>includes(value)</code>	This method returns <code>true</code> if the array contains the specified value.

<code>map(callback)</code>	This method returns a new array containing the result of invoking the <code>callback</code> function for every item in the array.
<code>reduce(callback)</code>	This method returns the accumulated value produced by invoking the <code>callback</code> 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.

4.4 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
```

4.4.1 Understanding literal object types

When the TypeScript compiler 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
```

4.4.2 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
```

Defining Static Methods

Static methods are accessed via the class and cannot access the instance fields, properties, and methods defined by a class. You can see examples of static methods in part 3 of this book.

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
```

4.4.3 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
```

4.5 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.

4.5.1 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 files 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.

4.6 *Summary*

In this chapter, I continued to describe the key features provided by TypeScript and JavaScript, including functions, arrays, objects, and modules.

- Functions define sets of statements that are not executed until the function is invoked.
- Functions can define optional parameters, default parameter values, rest parameters, and return results, all of which are annotated with types.
- Functions can accept other functions as parameters and there is a concise function syntax that makes this easier to write.
- JavaScript arrays are all variable length and can accept values of any type, unless restricted with a TypeScript type annotation.
- JavaScript objects are collections of properties, which are assigned values or functions.
- Objects can be created using a literal syntax or using classes.
- Multiple code files can be combined into a module, which makes it easier to structure the code in large projects.

In the next chapter, I start building a more realistic project that demonstrates many of the features that are described in depth later in the book.

5

SportsStore: A real application

This chapter covers

- Creating the SportsStore project
- Starting a data model with dummy data
- Using signals to track changes in data
- Displaying a list of products to the user
- Filtering products by category
- Paginating products

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* 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.

5.1 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/manningbooks/pro-angular-16>. See chapter 1 for how to get help if you have problems running the examples.

5.1.1 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@5.2.3
npm install @fortawesome/fontawesome-free@6.2.1
npm install --save-dev json-server@0.17.3
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

If you are using Linux or macOS, 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"
]
```

5.1.2 Preparing the RESTful web service

The `SportsStore` application will use asynchronous HTTP requests to get model data provided by a RESTful web service. REST is an approach to designing web services that use the HTTP method or verb to specify an operation and the URL to select the data objects to which operation applies.

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.

5.1.3 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 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.

5.1.4 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.

5.1.5 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.

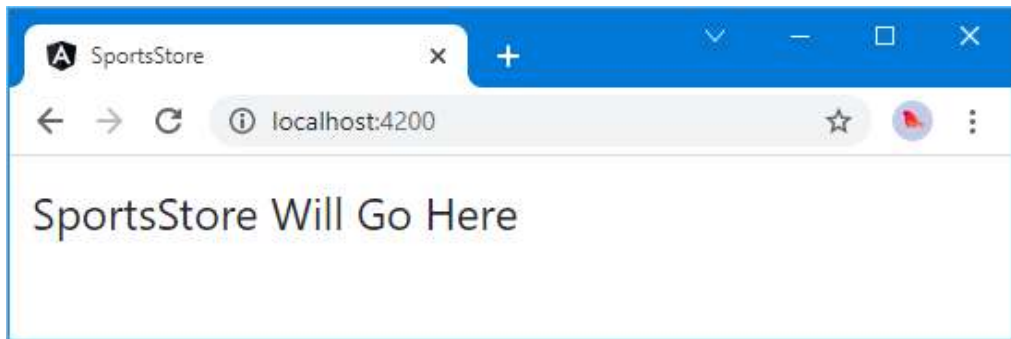


Figure 5.1. Running the example application

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.)

5.1.6 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
}
```

5.2 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.

5.2.1 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 part 2 of this book, 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.

5.2.2 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.

5.2.3 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 { platformBrowserDynamic }
  from '@angular/platform-browser-dynamic';
import { AppModule } from './app/app.module';

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.

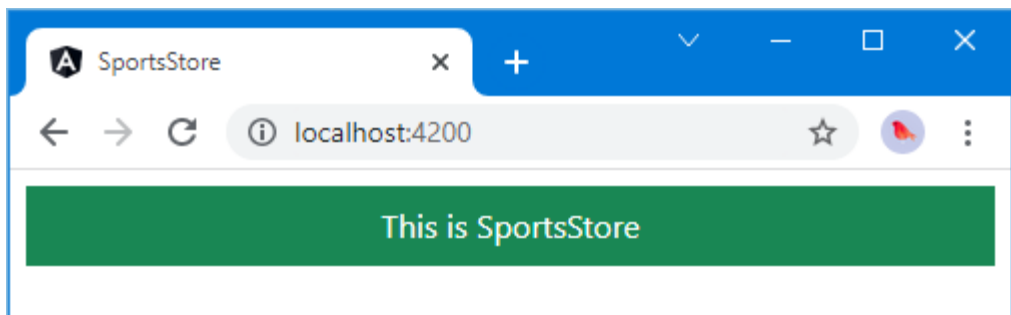


Figure 5.2. Starting the `SportsStore` application

5.3 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.

5.3.1 *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.

5.3.2 *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, Signal, signal } from "@angular/core";
import { Product } from "../product.model";

@Injectable()
export class StaticDataSource {
  private data: 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),
    ];

    products: Signal<Product[]> = signal<Product[]>(this.data)
}

```

The `StaticDataSource` class defines a property named `products`, which returns the dummy data. The type of the value returned by the `products` property is a `Signal<Product[]>`, which is an example of a signal. Signals are the headline addition to Angular 16 and they are used to describe the relationships between the data values used by an application, which helps Angular update the HTML content it presents to the user as efficiently as possible. Signals are included in Angular 16 as a preview, which means the API may change in Angular 17 and that the signal features are not yet fully integrated into the rest of the Angular API.

Signals are created with the `signal<T>` function, where `T` is the type of data contained by the signal. In this case, the type is an array of `Product` objects. As you will see shortly, signals of this type are used to create derived data values that are updated efficiently, as I explain in detail in part 2.

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 part 2. You'll see how services work as the application takes shape. (Strictly speaking, the `@Injectable` decorator can be omitted when a class has no constructor arguments, but it is good practice to apply the decorator anyway).

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.

5.3.3 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, Signal, computed } from "@angular/core";
import { Product } from "../product.model";
import { StaticDataSource } from "../static.datasource";

@Injectable()
export class ProductRepository {
  products: Signal<Product[]>;
  categories: Signal<string[]>;

  constructor(private dataSource: StaticDataSource) {
    this.products = dataSource.products;
    this.categories = computed(() => {
      return this.dataSource.products()
        .map(p => p.category ?? "(None)")
        .filter((c, index, array) =>
          array.indexOf(c) == index).sort();
    })
  }

  getProduct(id: number): Product | undefined {
    return this.dataSource.products().find(p => p.id == id);
  }
}
```

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 uses the signal created by the data source in different ways. The simplest use is to read the data contained in the signal, like this:

```
...
getProduct(id: number): Product | undefined {
  return this.dataSource.products().find(p => p.id == id);
}
...
```

Signals are read by invoking them like a function, which is a little awkward until you get used to the syntax. The expression `this.dataSource.products()` returns the array of

Product objects managed by the signal, which can then be used like any other array, including calling the `find` method.

A more complex use is to create a computed signal, like this:

```
...
constructor(private dataSource: StaticDataSource) {
  this.categories = computed(() => {
    return this.dataSource.Products()
      .map(p => p.category ?? "(None)")
      .filter((c, index, array) =>
        array.indexOf(c) == index).sort();
  })
}
...
```

The `computed` function accepts a function argument that generates a value using one or more other signals. In this case, the argument function reads the value of the `products` signal and uses the `map` and `filter` array methods to generate a string array containing the products categories.

Angular won't recompute the value of the computed signal unless the underlying signals change. In this case, this means that the mapping and filtering of the products will only be done when the products change. As I explain in part 2, this is a change from the way that Angular has traditionally worked and helps avoid recomputing values that have not changed.

Don't worry if signals don't make immediate sense. You will get a better idea of how they work as application features are added and I describe them in more detail in part 2 of this book.

5.3.4 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 part 2. Feature modules—and the `@NgModule` decorator—are also described in part 2.

5.4 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.

5.4.1 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, Signal } 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 {
  products: Signal<Product[]>;
  categories: Signal<string[]>;

  constructor(private repository: ProductRepository) {
    this.products = repository.products
    this.categories = repository.categories;
  }
}
```

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 part 2. 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.

5.4.2 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.

5.4.3 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 into.

If you see zero instead of the numbers of categories and products shown in the figure, then the likely cause is that you have not defined the template as shown in listing 5.16, and have forgotten the `()` characters in the string interpolation bindings.

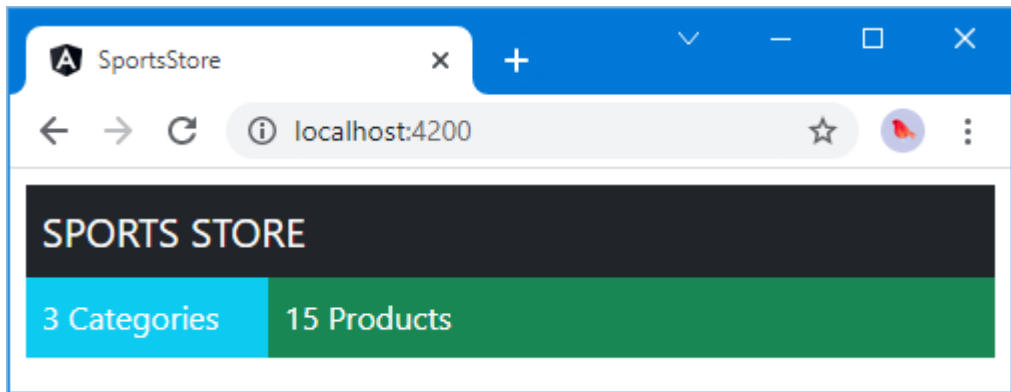


Figure 5.4. Basic features in the SportsStore application

5.5 Adding store features

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.

5.5.1 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>
</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 part 2.

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.

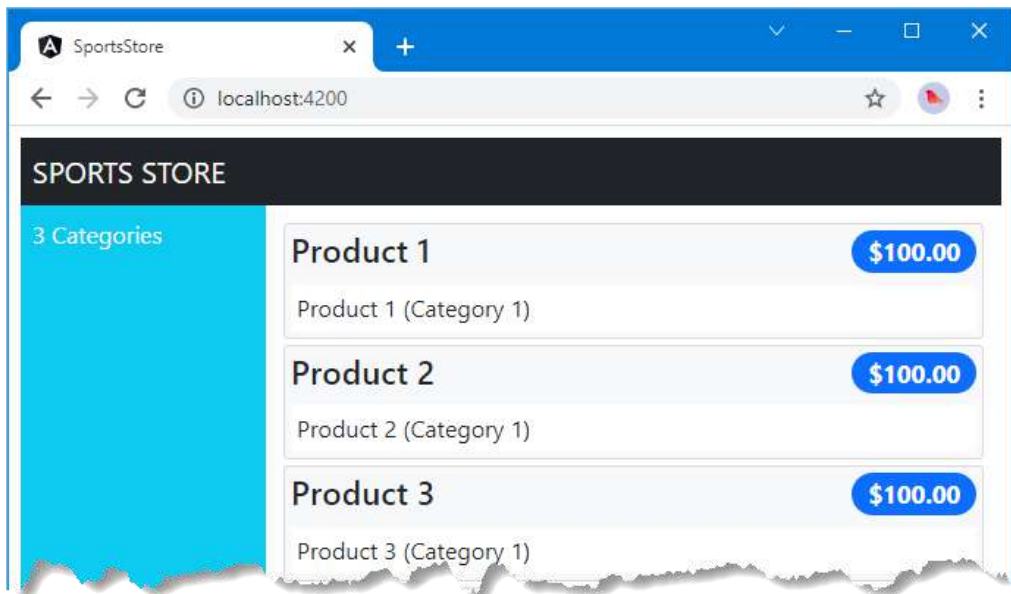


Figure 5.5. Displaying product information

5.5.2 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. Category filtering in the `store.component.ts` file in the `src/app/store` folder

```
import { Component, Signal, computed, signal } 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 {
  products: Signal<Product[]>;
  categories: Signal<string[]>;
  selectedCategory = signal<string | undefined>(undefined);

  constructor(private repository: ProductRepository) {
    this.products = computed(() => {
      if (this.selectedCategory() == undefined) {
        return this.repository.products();
      } else {
        return this.repository.products().filter(p =>
          p.category === this.selectedCategory());
      }
    })
    this.categories = repository.categories;
  }

  changeCategory(newCategory?: string) {
    this.selectedCategory.set(newCategory);
  }
}

```

The build on the foundation that took so long to create at the start of the chapter. The `selectedCategory` property is assigned a signal whose value is either a `string`, denoting a category selection, or `undefined`, denoting that no category has been chosen and that all categories should be displayed.

The `products` property is now a computed signal that uses the category selection and the repository product data to produce the set of products that should be displayed to the user. The dependencies between signals are tracked automatically and Angular will cache (or memoize) the result until either the selected product or the products in the repository change.

The `changeCategory` method will be invoked when the user selects a category and updates the value of the `selectedCategory` signal, which will invalidate the cached set of products and cause a computation the next time the `products` value is required.

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 for the method, which has the effect of setting the category to `null/undefined` 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.

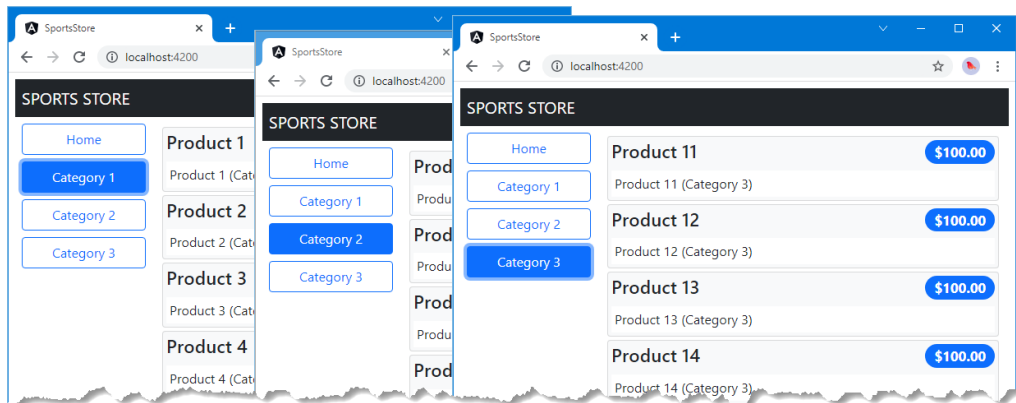


Figure 5.6. Selecting product categories

5.5.3 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 in the store.component.ts file in the src/app/store folder

```
import { Component, Signal, computed, signal } 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 {
  products: Signal<Product[]>;
  categories: Signal<string[]>;
  selectedCategory = signal<string | undefined>(undefined);
  productsPerPage = signal(4);
  selectedPage = signal(1);
  pagedProducts: Signal<Product[]>;
  pageNumbers: Signal<number[]>;

  constructor(private repository: ProductRepository) {
    this.products = computed(() => {
      if (this.selectedCategory() == undefined) {
        return this.repository.products();
      } else {
        return this.repository.products().filter(p =>
          p.category === this.selectedCategory());
      }
    });
  }
}
```

```

    this.categories = repository.categories;
    let pageIndex = computed(() => {
        return (this.selectedPage() - 1) * this.productsPerPage()
    });
    this.pagedProducts = computed(() => {
        return this.products().slice(pageIndex(),
            pageIndex() + this.productsPerPage());
    });
    this.pageNumbers = computed(() => {
        return Array(Math.ceil(this.products().length
            / this.productsPerPage()))
            .fill(0).map((x, i) => i + 1);
    });
}

changeCategory(newCategory?: string) {
    this.selectedCategory.set(newCategory);
    this.changePage(1);
}

changePage(newPage: number) {
    this.selectedPage.set(newPage);
}

changePageSize(newSize: number) {
    this.productsPerPage.set(Number(newSize));
    this.changePage(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. You can see how I use a series of computed signals to define the values that will be used by these features, explicitly defining the relationship between them so that Angular knows which values will need to be recomputed when there is a change.

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:

```

...
this.pageNumbers = computed(() => {
    return Array(Math.ceil(this.products().length /
        this.productsPerPage()))
        .fill(0).map((x, i) => i + 1);
});
...

```

This signal's value is computed by creating a new array whose length is set to the number of pages required to display the product data, filling it with the value 0, and then using the `map` method to generate a new array with a 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 pagedProducts()"
        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.

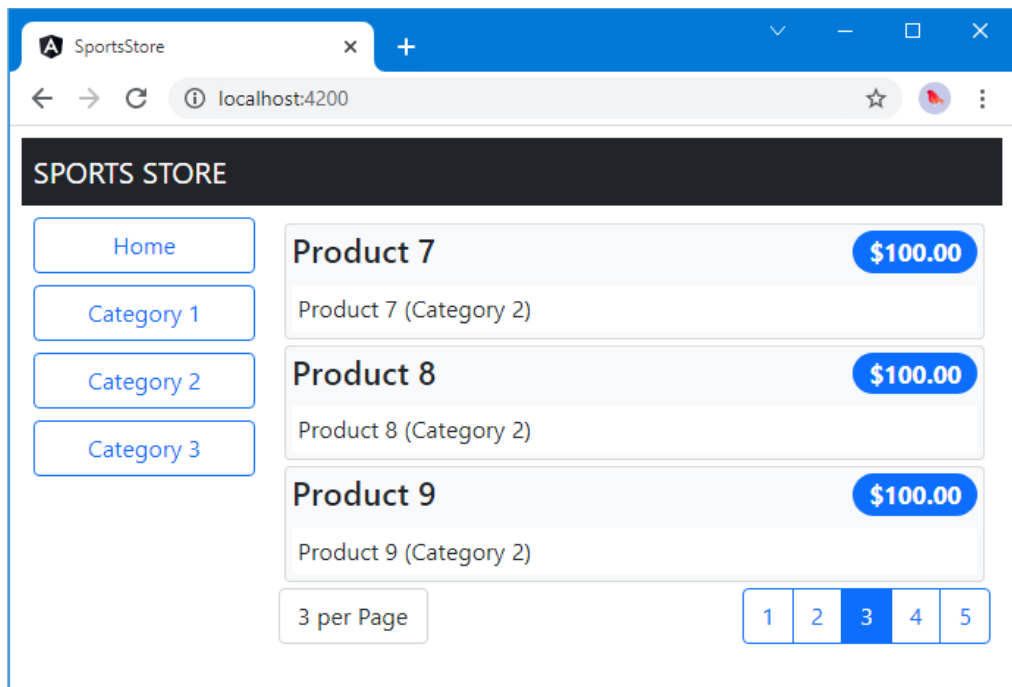


Figure 5.7. Pagination for products

5.5.4 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 part 2. 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 part 2.

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 pagedProducts()"
        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>
    </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>
</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, Signal, computed, signal } 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 {
  products: Signal<Product[]>;
  categories: Signal<string[]>;
  selectedCategory = signal<string | undefined>(undefined);
  productsPerPage = signal(4);
  selectedPage = signal(1);
  pagedProducts: Signal<Product[]>;
  //pageNumbers: Signal<number[]>;
  pageCount: Signal<number>;

  constructor(private repository: ProductRepository) {
    this.products = computed(() => {
      if (this.selectedCategory() == undefined) {
        return this.repository.products();
      } else {
        return this.repository.products().filter(p =>
          p.category === this.selectedCategory());
      }
    })
  }
}

```

```

    this.categories = repository.categories;
    let pageIndex = computed(() => {
        return (this.selectedPage() - 1) * this.productsPerPage()
    });
    this.pagedProducts = computed(() => {
        return this.products().slice(pageIndex(),
            pageIndex() + this.productsPerPage());
    });
    // this.pageNumbers = computed(() => {
    //     return Array(Math.ceil(this.products().length
    //         / this.productsPerPage()))
    //         .fill(0).map((x, i) => i + 1);
    // });
    this.pageCount = computed(() => {
        return Math.ceil(this.products().length
            / this.productsPerPage());
    });
}

changeCategory(newCategory?: string) {
    this.selectedCategory.set(newCategory);
    this.changePage(1);
}

changePage(newPage: number) {
    this.selectedPage.set(newPage);
}

changePageSize(newSize: number) {
    this.productsPerPage.set(Number(newSize));
    this.changePage(1);
}
}

```

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.

5.6 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.

6

SportsStore: orders and checkout

This chapter covers

- Creating a shopping cart
- Using URLs to navigate within the application
- Creating and storing customer orders
- Using a RESTful web service

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.

6.1 *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/manningbooks/pro-angular-16>. See chapter 1 for how to get help if you have problems running the examples.

6.2 *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.

6.2.1 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, Signal, WritableSignal, computed, signal }
  from "@angular/core";
import { Product } from "../product.model";

@Injectable()
export class Cart {
  private linesSignal: WritableSignal<CartLine[]>;
  public summary: Signal<CartSummary>;

  constructor() {
    this.linesSignal = signal([]);

    this.summary = computed(() => {
      let newSummary = new CartSummary();
      this.linesSignal().forEach(l => {
        newSummary.itemCount += l.quantity;
        newSummary.cartPrice += l.lineTotal;
      });
      return newSummary;
    })
  }

  get lines(): Signal<CartLine[]> {
    return this.linesSignal.asReadonly();
  }

  addLine(product: Product, quantity: number = 1) {
    this.linesSignal.mutate(linesArray => {
      let line = linesArray.find(l => l.product.id == product.id);
      if (line != undefined) {
        line.quantity += quantity;
      } else {
        linesArray.push(new CartLine(product, quantity));
      }
    });
  }

  updateQuantity(product: Product, quantity: number) {
    this.linesSignal.mutate(linesArray => {
      let line = linesArray.find(l => l.product.id == product.id);
      if (line != undefined) {
        line.quantity = Number(quantity);
      }
    });
  }

  removeLine(id: number) {
    this.linesSignal.mutate(linesArray => {
      let index = linesArray.findIndex(l => l.product.id == id);
```



```

        linesArray.splice(index, 1);
    });
}

clear() {
    this.linesSignal.set([]);
}

}

export class CartLine {

    constructor(public product: Product,
        public quantity: number) {}

    get lineTotal() {
        return this.quantity * (this.product.price ?? 0);
    }
}

export class CartSummary {
    itemCount: number = 0;
    cartPrice: number = 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 using the new signals feature, which ensures that any change made to the product selections automatically updates the summary that provides details of the number of products selected and the total price.

The result of the `signal<T>` function is a `WritableSignal<T>` object that, as its name suggests, can be modified. The simplest way to modify the signal is using the `set` method, which I introduced in chapter 5, and which replaces the signal's value, like this:

```

...
this.linesSignal.set([]);
...

```

There is also a `mutate` method, which is useful when a change is based on the signal's current value. I use the `mutate` method to add and remove product selections, like this:

```

...
removeLine(id: number) {
    this.linesSignal.mutate(linesArray => {
        let index = linesArray.findIndex(l => l.product.id == id);
        linesArray.splice(index, 1);
    });
}
...

```

This code finds a product and removes it from the selection using the signal's current value, which makes incremental signal changes possible.

I want to ensure that changes to the product selection are made only through the methods defined by the `Cart` class and so I don't want to expose the `WritableSignal` for use elsewhere in the application. Instead, I defined a property that uses the `asReadonly` method to return a `Signal<CartLine[]>` object that allows the current value to be read, and for computed signals to be created, but doesn't allow any modifications:

```

...
get lines(): Signal<CartLine[]> {
    return this.linesSignal.asReadOnly();
}
...

```

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 is described in detail in part 2. 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 an indication 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 { }

```

6.2.2 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 `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. Using a service means that a single `Cart` object will be created and shared throughout the application, as described in part 2 of this book.

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 `cartSummary.component.html` file in the `src/app/store` folder

```

<div class="float-end">
  <small class="fs-6">
    Your cart:
    <span *ngIf="cart.summary().itemCount > 0">
      {{ cart.summary().itemCount }} item(s)
      {{ cart.summary().cartPrice
        | currency:"USD":"symbol":"2.2-2" }}
    </span>
    <span *ngIf="cart.summary().itemCount == 0">
      (empty)
    </span>
  </small>
  <button class="btn btn-sm bg-dark text-white"
    [disabled]="cart.summary().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 { }

```

6.2.3 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, Signal, computed, signal } 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 {
  products: Signal<Product[]>;
  categories: Signal<string[]>;
  selectedCategory = signal<string | undefined>(undefined);
  productsPerPage = signal(4);
  selectedPage = signal(1);
  pagedProducts: Signal<Product[]>;
  pageCount: Signal<number>;

  constructor(private repository: ProductRepository,
    private cart: Cart) {
    this.products = computed(() => {
      if (this.selectedCategory() == undefined) {
        return this.repository.products();
      } else {
        return this.repository.products().filter(p =>
          p.category === this.selectedCategory());
      }
    })

    this.categories = repository.categories;

    let pageIndex = computed(() => {
      return (this.selectedPage() - 1) * this.productsPerPage()
    });

    this.pagedProducts = computed(() => {

```

```

        return this.products().slice(pageIndex(),
            pageIndex() + this.productsPerPage());
    });

    this.pageCount = computed(() => {
        return Math.ceil(this.products().length
            / this.productsPerPage());
    });
}

changeCategory(newCategory?: string) {
    this.selectedCategory.set(newCategory);
    this.changePage(1);
}

changePage(newPage: number) {
    this.selectedPage.set(newPage);
}

changePageSize(newSize: number) {
    this.productsPerPage.set(Number(newSize));
    this.changePage(1);
}

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 pagedProducts()"
        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>
</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.

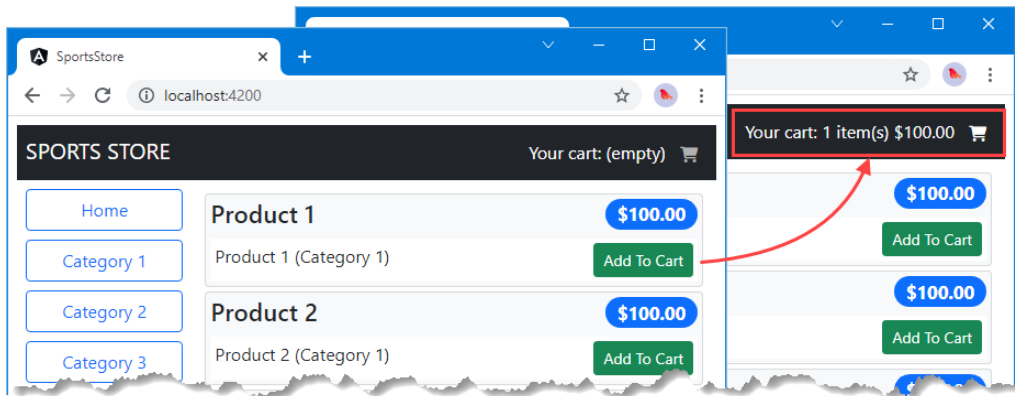


Figure 6.1. Adding cart support to the SportsStore application

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.

6.3 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 part 3.

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.

6.3.1 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 { }
```

6.3.2 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.

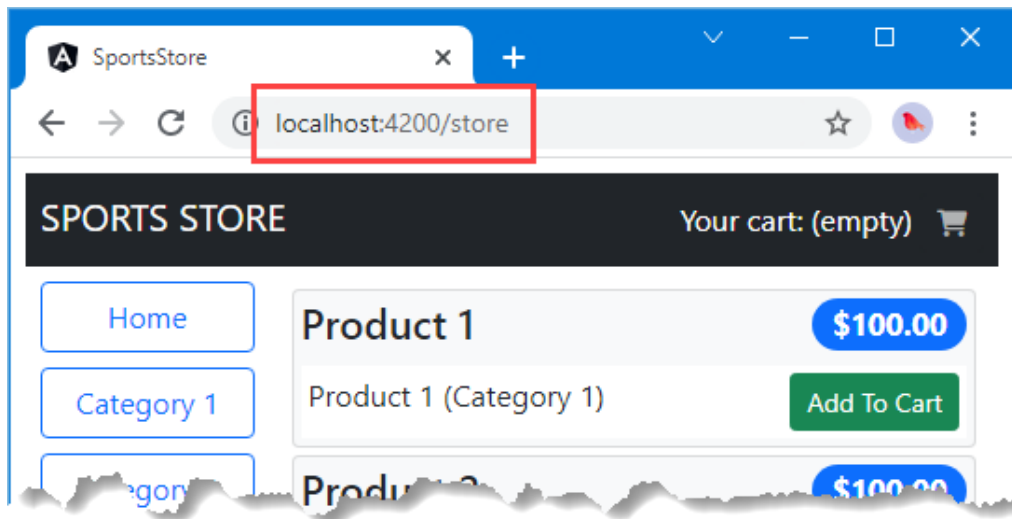


Figure 6.2. The effect of URL routing

6.3.3 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, Signal, computed, signal } 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 {
  products: Signal<Product[]>;
  categories: Signal<string[]>;
  selectedCategory = signal<string | undefined>(undefined);
  productsPerPage = signal(4);
  selectedPage = signal(1);
  pagedProducts: Signal<Product[]>;
  //pageNumbers: Signal<number[]>;
  pageCount: Signal<number>;

  constructor(private repository: ProductRepository,
               private cart: Cart,
               private router: Router) {

    // ...statements omitted for brevity...
  }

  // ...methods omitted for brevity...

  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.summary().itemCount > 0">
      {{ cart.summary().itemCount }} item(s)
      {{ cart.summary().cartPrice
        | currency:"USD":"symbol":"2.2-2" }}
    </span>
    <span *ngIf="cart.summary().itemCount == 0">
      (empty)
    </span>
  </small>
  <button class="btn btn-sm bg-dark text-white"
    [disabled]="cart.summary().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.

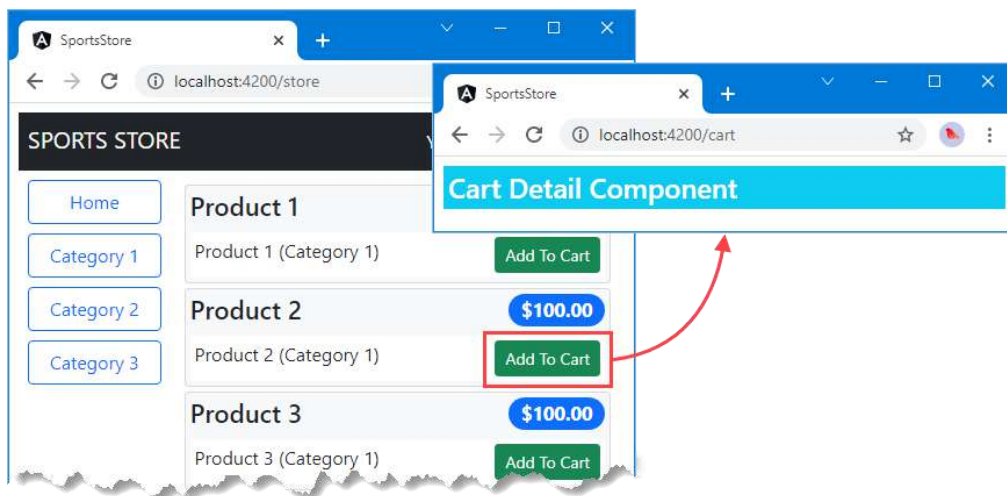


Figure 6.3. Using URL routing

6.3.4 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 part 3, 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.

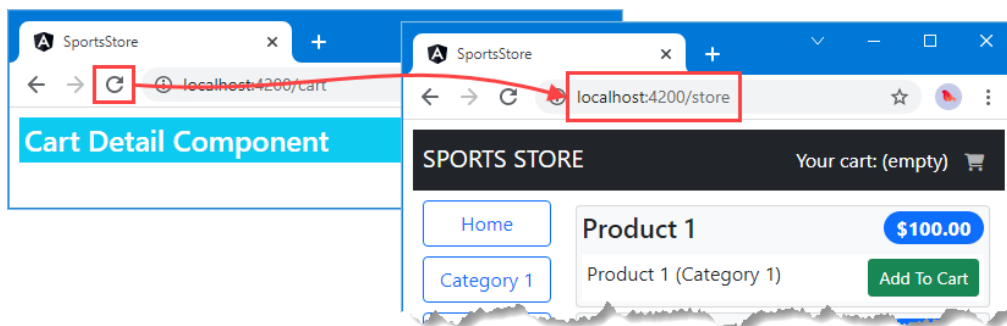


Figure 6.4. Guarding routes

6.4 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 `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>
            <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>
      </table>
    </div>
  </div>
</div>
```



```

        <tr>
          <td colspan="3" class="text-end">Total:</td>
          <td class="text-end">
            {{cart.summary().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>
</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 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.

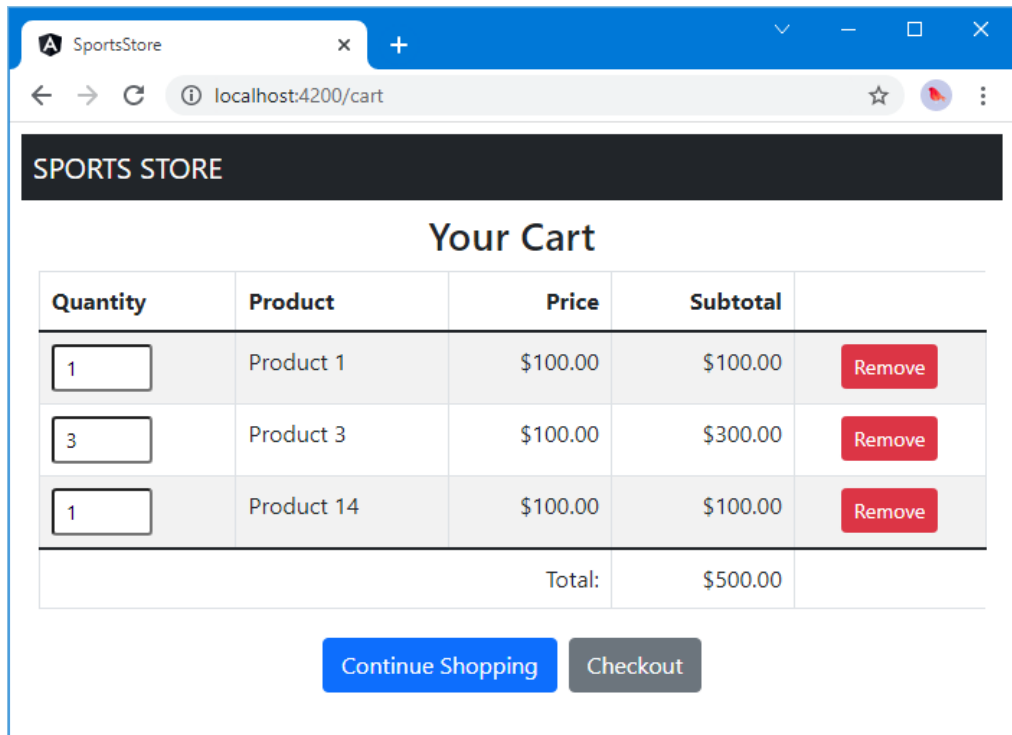


Figure 6.5. Completing the cart detail feature

6.5 Processing orders

Being able to receive orders from customers is the most important aspect of an online store. In the sections that follow, I build on the application to add support for receiving the final details from the user and checking them out. To keep the process simple, I am going to avoid dealing with payment and fulfillment platforms, which are generally back-end services that are not specific to Angular applications.

6.5.1 Extending the model

To describe orders placed by users, I added a file called `order.model.ts` in the `src/app/model` folder and defined the code shown in listing 6.20.

Listing 6.20. The contents of the `order.model.ts` file in the `src/app/model` folder

```
import { Injectable } from "@angular/core";
import { Cart, CartLine } from "../cart.model";

@Injectable()
export class Order {
  id?: number;
```

```

    name?: string;
    address?: string;
    city?: string;
    state?: string;
    zip?: string;
    country?: string;
    shipped: boolean = false;
    #cart: Cart;
    lines: CartLine[];

    constructor(c: Cart) {
        this.#cart = c;
        this.lines = c.lines();
    }

    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.

Notice that one of the property names is prefixed with a `#` character:

```

...
#cart: Cart;
...

```

I am going to use the built-in JavaScript serialization feature to create a string representation of the order when it is saved. The serialization process will include all of the properties defined by the `Order` class unless they are private. The TypeScript `private` keyword won't affect the serialization process because it will be erased from the JavaScript code generated by the TypeScript compiler. The `#` character is the JavaScript feature for identifying private class features and so I have to name the property `#cart` in order to exclude it from the serialization process.

This is a good reminder that TypeScript features can be more comprehensive and useful than those provided by pure JavaScript, but sometimes only the JavaScript features will work because they are the only ones that make it through the compilation process.

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, Signal, signal } from "@angular/core";
import { Product } from "../product.model";
import { Order } from "../order.model";
import { Observable, from } from "rxjs";

@Injectable()
export class StaticDataSource {
  private data: 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),
  ];

  products: Signal<Product[]> = signal<Product[]>(this.data)

  saveOrder(order: Order): Observable<Order> {
    console.log(JSON.stringify(order));
    return from([order]);
  }
}

```

The result from the `saveOrder` method is an *observable*, which is a representation of a sequence of asynchronous values. Observables are part of the RxJS package and are used by Angular for representing asynchronous results and values that change over time, although some uses of observables are likely to be replaced with signals in future releases of Angular. The `Observable<Order>` result represents an asynchronous sequence of `Order` objects, which will dovetail with the way that the built-in Angular support for making HTTP requests

represents data it receives later in this chapter. Observables are described in more detail in part 2, as is their relationship with signals.

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 when I create the administration features for the SportsStore application.

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 {

    constructor(private dataSource: StaticDataSource) {}

    saveOrder(order: Order): Observable<Order> {
        return this.dataSource.saveOrder(order);
    }
}
```

The repository is just a wrapper around the data source's `saveOrder` method for the moment, but I will add more features later.

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 { }
```

6.5.2 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 later chapters, 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. Add a file named `checkout.component.html` to the `src/app/store` folder with the content shown in listing 6.26.

Listing 6.26. 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"
      [(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 explain in part 3, 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.

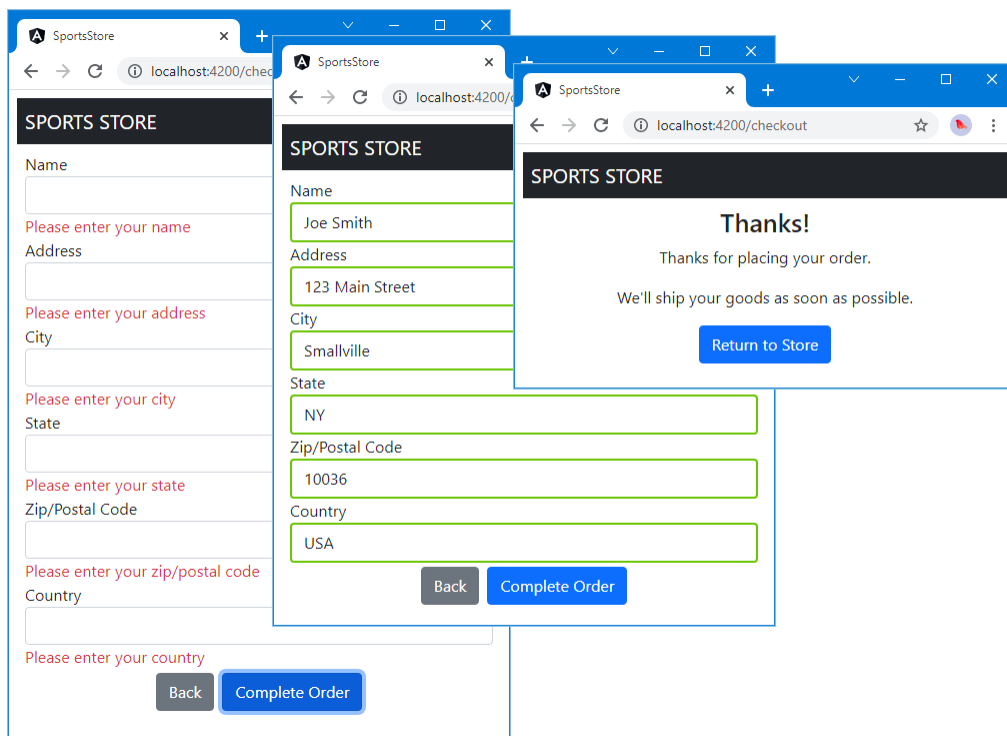


Figure 6.6. Completing an order

If you look at the browser's JavaScript console, you will see a JSON representation of the order like this:

```
{ "shipped": false,
  "lines": [ { "product": { "id": 1, "name": "Product 1", "category": "Category 1",
    "description": "Product 1 (Category 1)", "price": 100 }, "quantity": 1 },
  { "name": "Joe Smith", "address": "123 Main Street",
    "city": "Smallville", "state": "New York", "zip": "10036", "country": "USA" }
```

6.6 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}/`;
  }

  get products(): 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 members defined by the `RestDataSource` class correspond to the ones defined by the static data source but are implemented using the `HttpClient` service, which I describe in more detail in part 3 of this book.

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. I address this problem in chapter 8.

6.6.1 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. 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";

```

```

@NgModule({
  imports: [HttpClientModule],
  providers: [ProductRepository, StaticDataSource, Cart, Order,
              OrderRepository, 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 `providers` property adds the `RestDataSource` to the set of services, which allows the dummy data source to be replaced, as shown in listing 6.29.

Listing 6.29. Using a new data source in the `product.repository.ts` file in the `src/app/model` folder

```

import { Injectable, Signal, computed } from "@angular/core";
import { Product } from "../product.model";
import { StaticDataSource } from "../static.datasource";
import { RestDataSource } from "../rest.datasource";
import { toSignal } from "@angular/core/rxjs-interop";

@Injectable()
export class ProductRepository {
  products: Signal<Product[]>;
  categories: Signal<string[]>;

  constructor(private dataSource: RestDataSource) {
    this.products = toSignal(dataSource.products, {
      initialValue: []
    });
    this.categories = computed(() => {
      return this.products()
        .map(p => p.category ?? "(None)")
        .filter((c, index, array) =>
          array.indexOf(c) == index).sort();
    })
  }

  getProduct(id: number): Product | undefined {
    return this.products().find(p => p.id == id);
  }
}

```

The results produced by the `HttpClient` class are expressed as observables and I have to use an interoperability function to convert the observable into a signal that can be consumed by the rest of the application. I describe the interoperability in more detail in part 2, but for this chapter, it is enough to know that the `toSignal` accepts an observable and returns a signal.

Listing 6.30 updates the orders repository to use the new data source.

Listing 6.30. Using a new data source 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 {

    constructor(private dataSource: RestDataSource) {}

    saveOrder(order: Order): Observable<Order> {
        return this.dataSource.saveOrder(order);
    }
}

```

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.

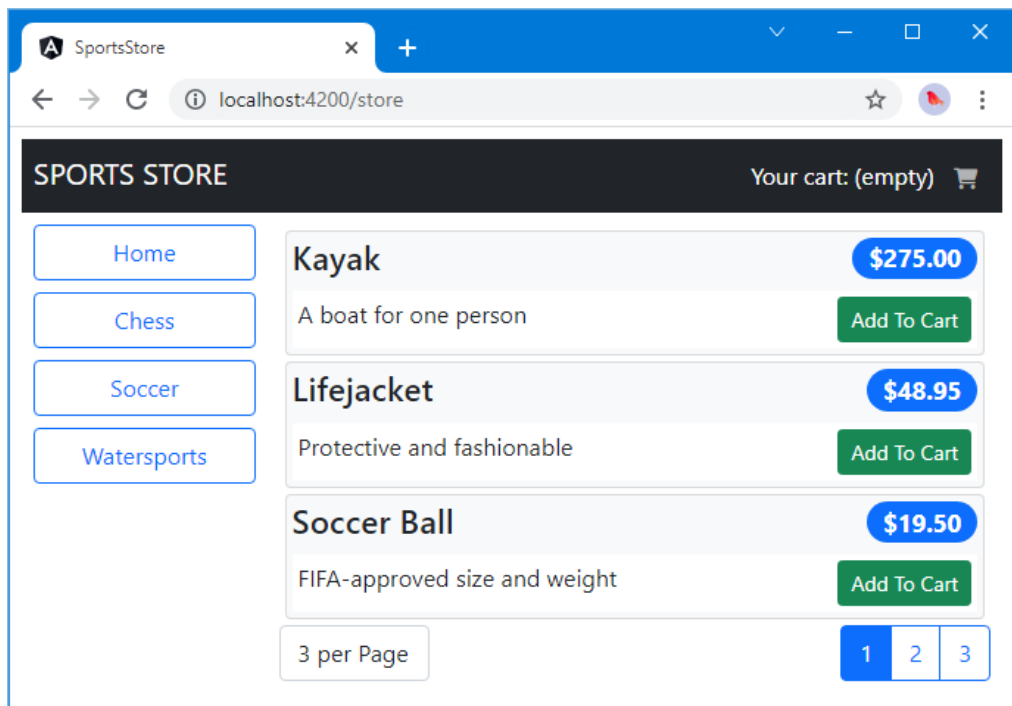


Figure 6.7. Using the RESTful web service

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.

6.7 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.

7

SportsStore: administration

This chapter covers

- Performing client authentication
- Creating a module that is loaded on demand
- Installing a component library
- Creating the administration features

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.

7.1 *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/manningbooks/pro-angular-16>. See chapter 1 for how to get help if you have problems running the examples.

7.1.1 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>
    <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.

7.1.2 *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]
    },
    {
      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.

7.1.3 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>
</div>

```

```

</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.

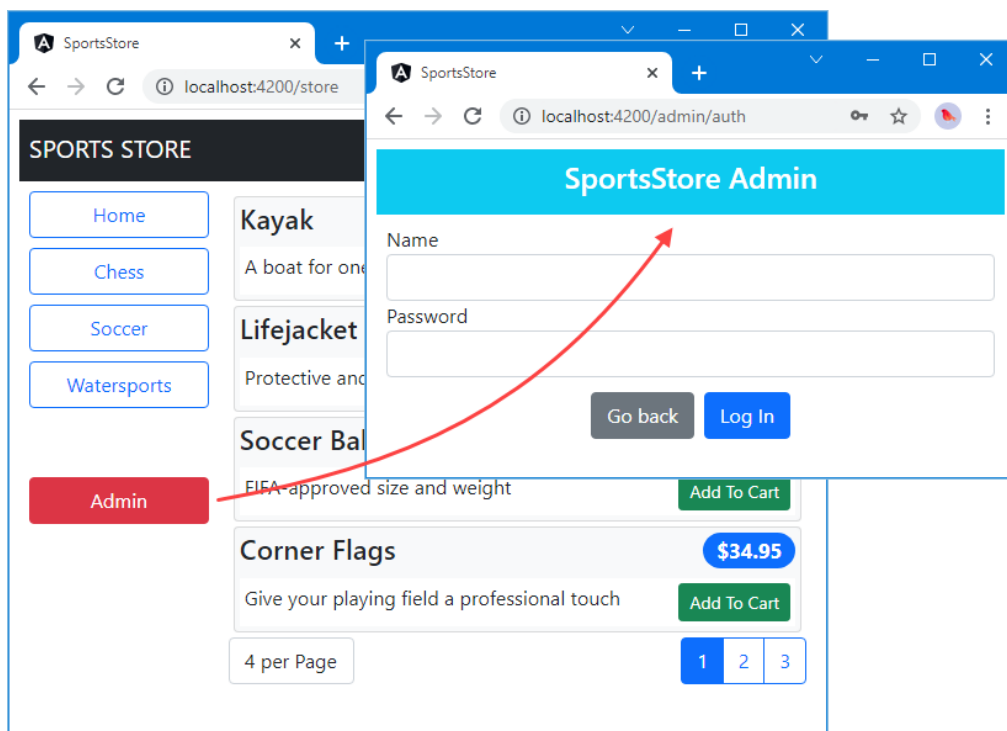


Figure 7.1. Using a dynamically loaded module

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.

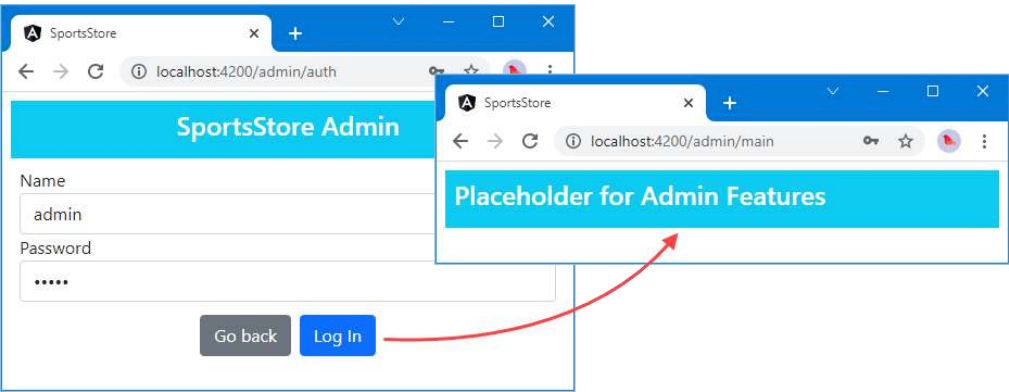


Figure 7.2. The placeholder administration features

7.2 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.

7.2.1 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://datatracker.ietf.org/doc/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 (but with a longer value for the `token` property, which I have edited to fit on the page):

```
{
  "success": true,
  "token": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJkYXN0bG8iOiJhbm91dCJ9.eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJkYXN0bG8iOiJhbm91dCJ9"
}
```

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.eyJkYXN0OyZlg8>
```

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
}
```

7.2.2 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}/`;
  }

  get products(): 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<boolean>` that is the result of the `authenticate` method.

7.2.3 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, RestDataSource, AuthService]
})
export class ModelModule { }

```

7.2.4 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, 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.

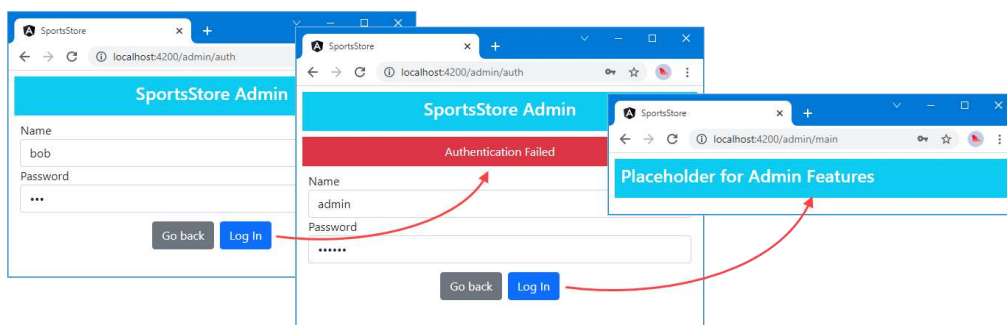


Figure 7.3. Testing the authentication feature

7.3 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}/`;
  }

  get products(): 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, Signal, WritableSignal, computed, signal } from "@angular/core";
import { Product } from "../product.model";
import { StaticDataSource } from "../static.datasource";
import { RestDataSource } from "../rest.datasource";
import { toSignal } from "@angular/core/rxjs-interop";

@Injectable()
export class ProductRepository {
  products = signal<Product[]>([]);
  categories: Signal<string[]>;

  constructor(private dataSource: RestDataSource) {
    dataSource.products.subscribe(data => {
      this.products.set(data);
    });
    this.categories = computed(() => {
      return this.products()
        .map(p => p.category ?? "(None)")
        .filter((c, index, array) =>
          array.indexOf(c) == index).sort();
    });
  }

  getProduct(id: number): Product | undefined {
    return this.products().find(p => p.id == id);
  }

  saveProduct(product: Product) {
    if (product.id == null || product.id == 0) {
      this.dataSource.saveProduct(product)
        .subscribe(p => {
          this.products.mutate(pdata => pdata.push(p));
        });
    } else {
      this.dataSource.updateProduct(product)
        .subscribe(p => {
          this.products.mutate(pdata => {
            pdata.splice(pdata.findIndex(p =>
              p.id == product.id), 1, product);
          });
        });
    }
  }
}
```

```

    deleteProduct(id: number) {
      this.dataSource.deleteProduct(id).subscribe(p => {
        this.products.mutate(pdata => {
          pdata.splice(pdata.findIndex(p => p.id == id), 1);
        })
      })
    }
  }
}

```

Previously, I just needed to display the data that was provided by the data source, but now I want to be able to make changes to that data. To do this, I changed the `products` signal so that it is defined by the repository class and use the `set` and `mutate` methods to change the value it contains. Instead of using the `toSignal` interoperability function, I `subscribe` to the observable returned by the data source and use the values I receive to update the signal.

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, signal } 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 ordersSignal = signal<Order[]>([]);
  private loaded: boolean = false;

  constructor(private dataSource: RestDataSource) {}

  loadOrders() {
    this.loaded = true;
    this.dataSource.getOrders().subscribe(data => {
      this.ordersSignal.set(data);
    });
  }

  get orders() {
    if (!this.loaded) {
      this.loadOrders();
    }
    return this.ordersSignal.asReadonly();
  }

  saveOrder(order: Order): Observable<Order> {
    this.loaded = false;
    return this.dataSource.saveOrder(order);
  }

  updateOrder(order: Order) {
    this.dataSource.updateOrder(order).subscribe(order => {
      this.ordersSignal.mutate(orders => {

```

```

        orders.splice(orders.findIndex(o =>
            o.id == order.id), 1, order);
    })
    });
}

deleteOrder(id: number) {
    this.dataSource.deleteOrder(id).subscribe(order => {
        this.ordersSignal.mutate(orders => {
            orders.splice(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.

7.4 *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@16.0.0 --defaults
```

Select the Yes option to install the package when prompted. 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, 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.

7.5 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.

<code>/admin/main/products/edit/1</code>	Navigating to this URL will present the user with a populated editor for editing an existing product.
<code>/admin/main/orders</code>	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.

7.5.1 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 `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 `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 `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 { }

```

7.5.2 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 `routerLinkActive` 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, 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 part 3. 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.

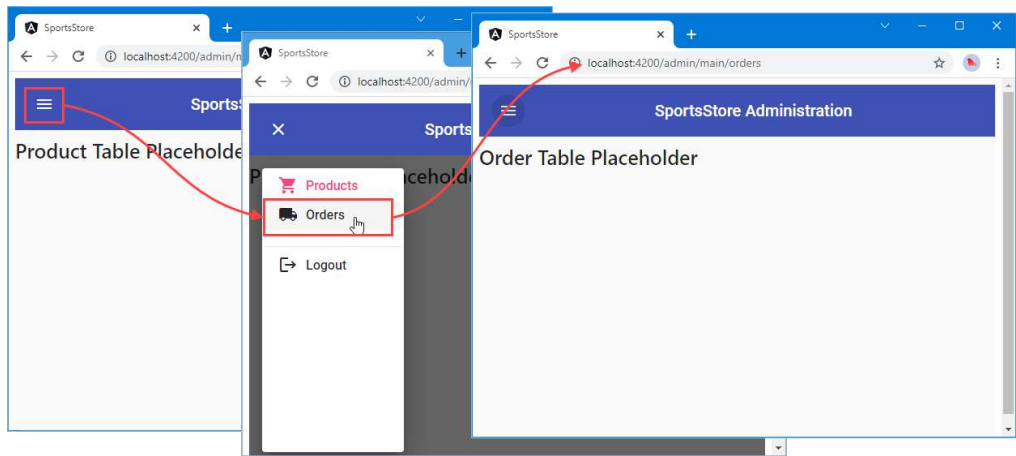


Figure 7.4. The administration layout structure

7.5.3 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 `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;}

```

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.products());
    differ: IterableDiffer<Product>;

    constructor(private repository: ProductRepository,
        differs: IterableDiffers) {
        this.differ = differs.find(this.repository.products()).create();
    }

    ngDoCheck() {
        let changes = this.differ?.diff(this.repository.products());
        if (changes != null) {
            this.dataSource.data = this.repository.products();
        }
    }

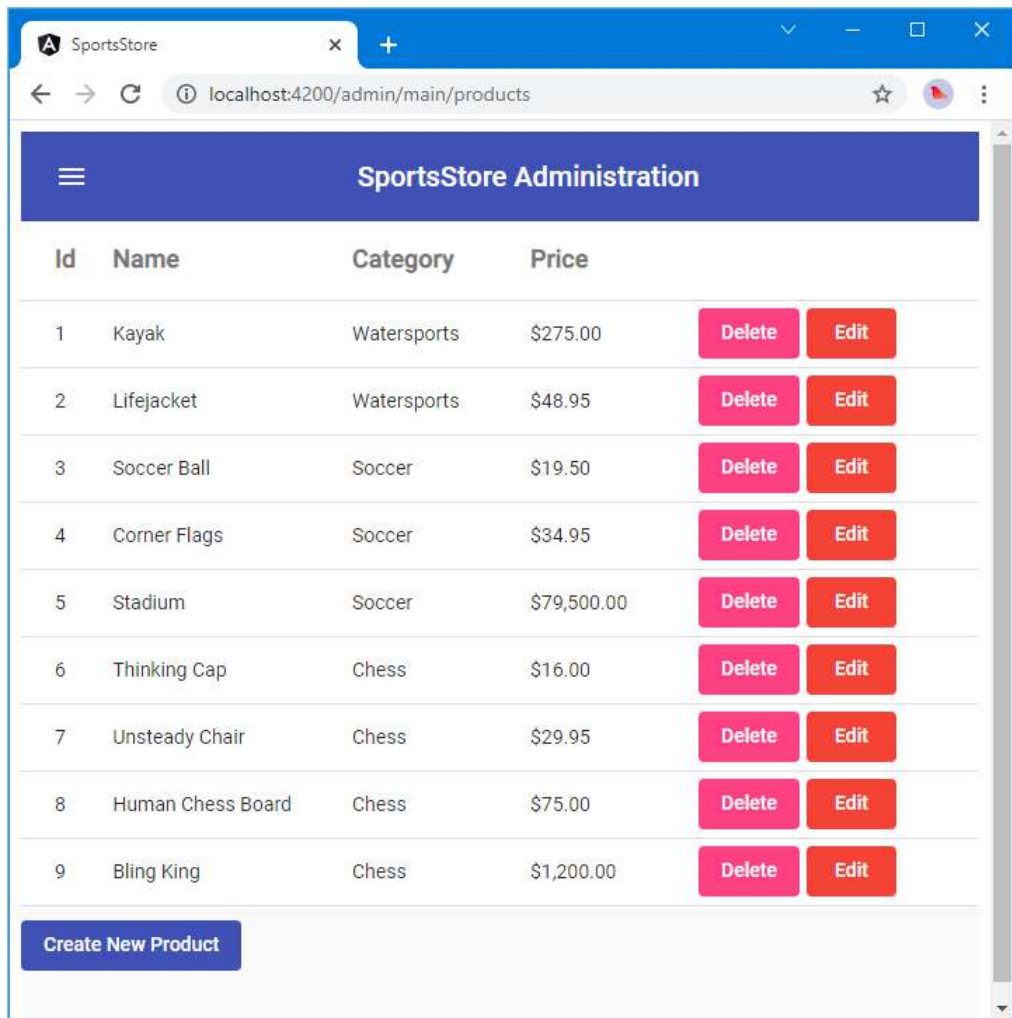
    deleteProduct(id: number) {
        this.repository.deleteProduct(id);
    }
}

```

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. This feature doesn't use signals, which are yet to be fully integrated into the Angular or Angular Material APIs.

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 with the title 'SportsStore' and the URL 'localhost:4200/admin/main/products'. The page has a dark blue header with a hamburger menu icon and the text 'SportsStore Administration'. Below the header is a table with 9 rows of product data. Each row has columns for 'Id', 'Name', 'Category', and 'Price', followed by 'Delete' and 'Edit' buttons. At the bottom of the table is a blue button labeled 'Create New Product'.

Id	Name	Category	Price		
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 paginating 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, IterableDiffers, 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.products());
  differ: IterableDiffer<Product>;

  constructor(private repository: ProductRepository,
    differs: IterableDiffers) {
    this.differ = differs.find(this.repository.products()).create();
  }

  ngDoCheck() {
    let changes = this.differ?.diff(this.repository.products());
    if (changes != null) {
      this.dataSource.data = this.repository.products();
    }
  }

  deleteProduct(id: number) {
    this.repository.deleteProduct(id);
  }

  @ViewChild(MatPaginator)
  paginator? : MatPaginator

  ngAfterViewInit() {
    if (this.paginator) {
      this.dataSource.paginator = this.paginator;
    }
  }
}

```

The `ViewChild` decorator is used to query the component's template content, as described in part 2, and is used here to find the paginator component. The `ngAfterViewInit` method is called after Angular has finished processing the template, also described in part 2, 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.

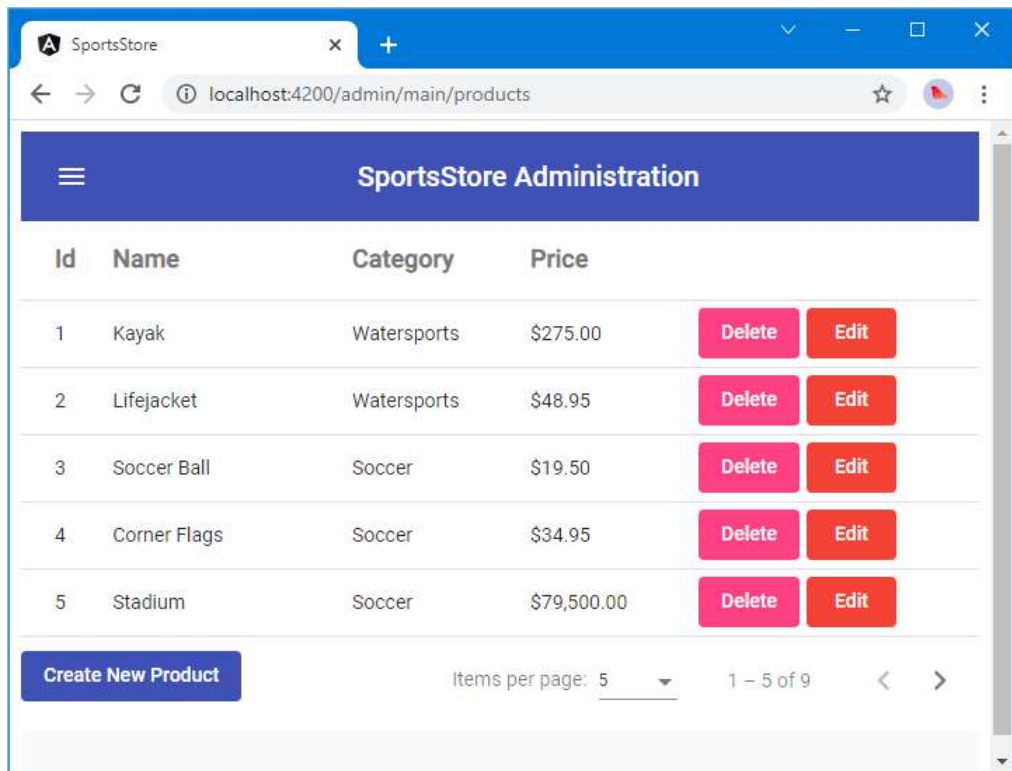


Figure 7.6. Using the Angular Material table paginator

7.5.4 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 `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.

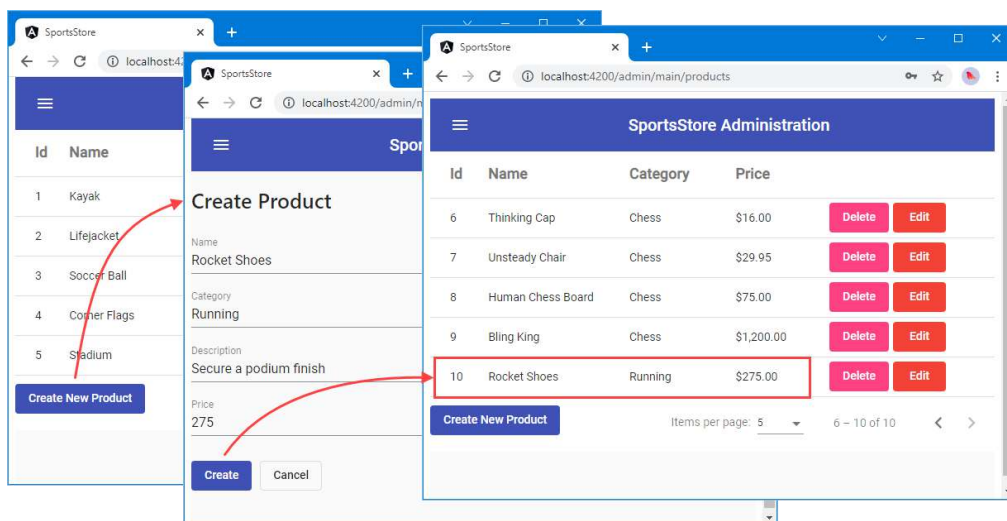


Figure 7.7. Creating a new product

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.

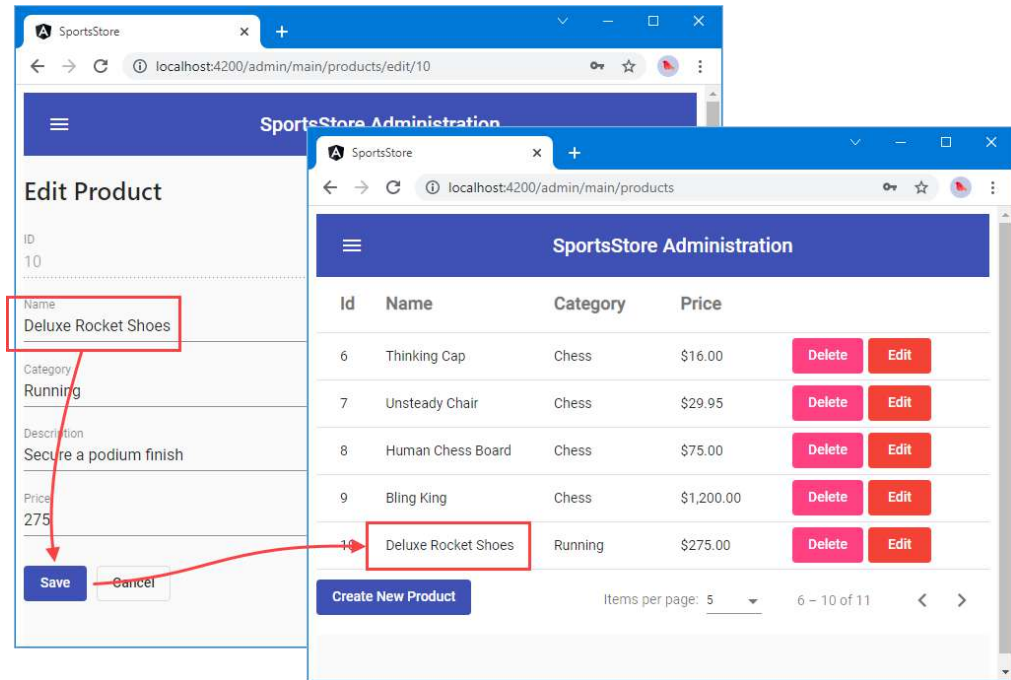


Figure 7.8. Editing an existing product

7.5.5 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 `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.lines">
        <ng-container matColumnDef="p">
          <th mat-header-cell *matHeaderCellDef>Product</th>
          <td mat-cell *matCellDef="let line">
            {{ line.product.name }}
          </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.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>

```

```

        </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>
      </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.orders());
  differ: IterableDiffer<Order>;

  constructor(private repository: OrderRepository,
    differs: IterableDiffers) {
    this.differ = differs.find(this.repository.orders()).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.orders());
        if (changes != null) {
            this.dataSource.data = this.repository.orders();
        }
    }
}

```

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; margin: 10px; display: block;}
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.

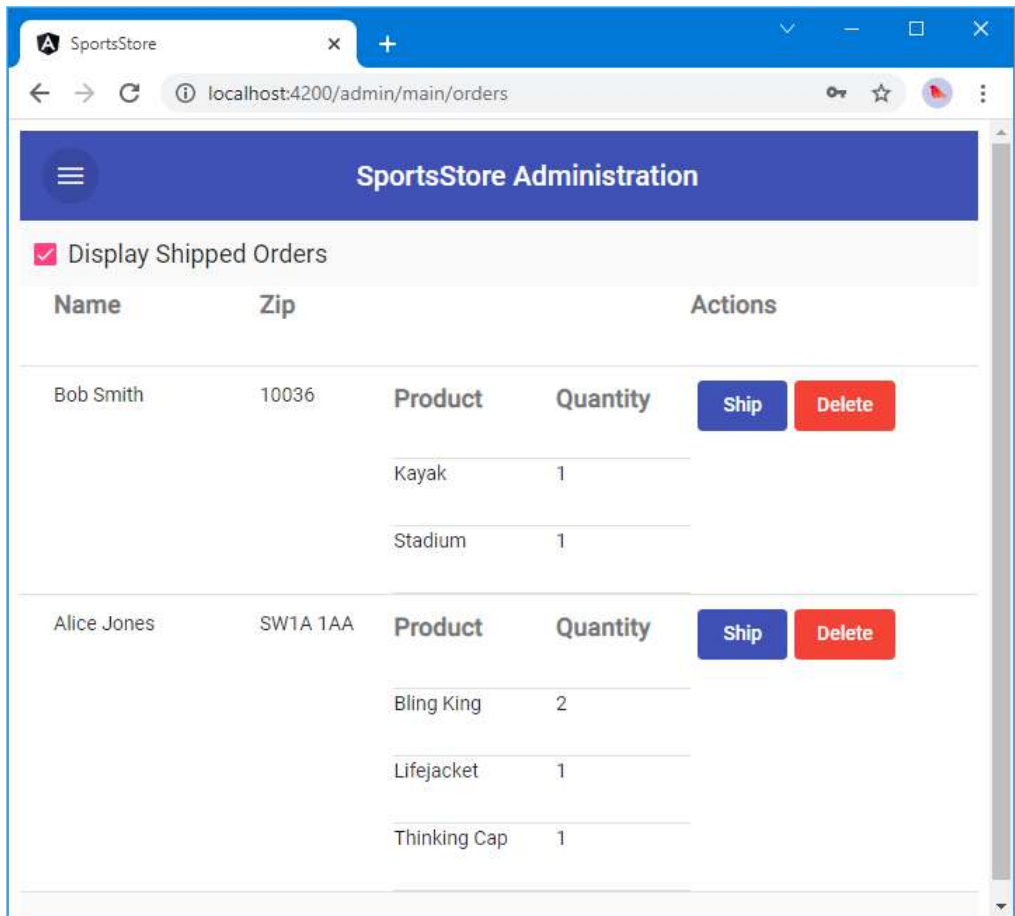


Figure 7.9. Managing orders

7.6 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.

8

SportsStore: deployment

This chapter covers

- Prerendering the application
- Adding progressive features
- Preparing for deployment
- Containerizing and running the deployed application

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.

8.1 *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/manningbooks/pro-angular-16>. See chapter 1 for how to get help if you have problems running the examples.

8.2 *Using pre-rendering*

One problem with JavaScript applications is that browsers have to download some relatively large files. Not everyone has a fast and reliable network connection, and that can mean that

the user is presented with an empty window while their browser gradually receives the JavaScript files that are required to start the application.

One way to improve this situation is to use server-side rendering (SSR), which presents the browser with an HTML-only version of the application, which the user can interact with while the JavaScript files are downloading. Once the JavaScript files are ready, the HTML elements are rehydrated, where they become the foundation for the content generated by the JavaScript code.

SSR can make an application usable over connections that are too slow to start the JavaScript version of the application quickly. The main drawback of SSR is that it requires substantial server-side resources because the application is being executed on the server to generate the HTML the user sees.

A variation of SSR is pre-rendering, which creates a single HTML representation of the application at build time and uses it for all users. This approach doesn't have the flexibility of full SSR because users all see the same HTML documents, but it requires no additional server resources, still works with rehydration, and is well-suited to applications that want to avoid showing users an empty browser screen without the investment that full SSR requires.

8.2.1 Installing the SSR packages

SSR uses a package named Angular Universal, which contains all of the build tools and the runtime configuration required for SSR and prerendering. The Angular Universal package can be picky about the package versions it requires and so the first step is to update to the latest Angular 16 releases. Stop the `ng serve` command and run the command shown in listing 8.1 in the `SportsStore` folder.

Listing 8.1. Updating the Angular packages

```
ng update @angular/cli@^16 @angular/core@^16
```

Once the updates have been installed, run the command shown in listing 8.2 in the `SportsStore` folder to install the Angular Universal package. Select "Yes" when prompted to proceed with the installation.

Listing 8.2. Installing the Angular Universal package

```
ng add @nguniversal/express-engine@^16
```

As part of the installation, the project will be configured for SSR, and a number of files will be created. Once the installation has completed, enable the rehydration feature, as shown in listing 8.3.

Listing 8.3. Enabling hydration in the `app.module.ts` file in the `src/app` folder

```
import { NgModule } from '@angular/core';
import { BrowserModule, provideClientHydration }
  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";
import { BrowserAnimationsModule }
  from '@angular/platform-browser/animations';

@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: "admin",
        loadChildren: () => import("../admin/admin.module")
          .then(m => m.AdminModule),
        canActivate: [StoreFirstGuard]
      },
      { path: "**", redirectTo: "/store" }
    ]), {
    initialNavigation: 'enabledBlocking'
  }]),
  BrowserAnimationsModule],
  providers: [StoreFirstGuard, provideClientHydration()],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

Rehydration is a preview feature that works but may change in future releases, much like the signals feature used in earlier chapters.

8.2.2 Creating the platform service

It can be useful to know when the code is being executed on the server. The Angular API provides a function that can be used to figure out where code is being executed and my preference is to make this information available in a service that can be used consistently throughout the application. Add a file named `platform.service.ts` to the `src/app` folder with the content shown in listing 8.4.

Listing 8.4. The contents of the `platform.service.ts` file in the `src/app` folder

```

import { isPlatformServer } from "@angular/common";
import { Inject, Injectable, PLATFORM_ID } from "@angular/core";

@Injectable()
export class PlatformService {

```



```

    constructor(@Inject(PLATFORM_ID) private platformId: Object) {}

    get isServer() { return isPlatformServer(this.platformId); }
}

```

The service defines an `isServer` property whose value will be true when the application is executed on the server for SSR. Listing 8.5 registers the service.

Listing 8.5. Registering a service in the `app.module.ts` file in the `src/app` folder

```

import { NgModule } from '@angular/core';
import { BrowserModule, provideClientHydration }
  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';
import { BrowserAnimationsModule }
  from '@angular/platform-browser/animations';
import { PlatformService } from './platform.service';

@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: "admin",
        loadChildren: () => import("./admin/admin.module")
          .then(m => m.AdminModule),
        canActivate: [StoreFirstGuard]
      },
      { path: "**", redirectTo: "/store" }
    ]), {
    initialNavigation: 'enabledBlocking'
  }]),
  BrowserAnimationsModule],
  providers: [StoreFirstGuard, provideClientHydration(), PlatformService],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

8.2.3 Changing the web service URL

One browser feature that isn't available in SSR is the `location` object, which I used in the RESTful data source to build the URL for requesting data. When the application is running on the server, I have to alter the URL, as shown in listing 8.6.

Listing 8.6. 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 { map, Observable } from "rxjs";
import { Product } from "../product.model";
import { Order } from "../order.model";
import { HttpHeaders } from '@angular/common/http';
import { PlatformService } from "../platform.service";

const PROTOCOL = "http";
const PORT = 3500;

@Injectable()
export class RestDataSource {
  baseUrl: string;
  auth_token?: string;

  constructor(private http: HttpClient, ps: PlatformService) {
    this.baseUrl = ps.isServer
      ? `http://localhost:${PORT}/`
      : `${PROTOCOL}://${location.hostname}:${PORT}/`;
  }

  // ...statements omitted for brevity...
}
```

Angular Universal is supposed to deal with URLs automatically but does not and this change relies on the server executing the Angular application and the web service running on the same machine.

8.2.4 Disabling elements

It is possible to support navigation in a prerendered or SSR application, as I demonstrate in part 3, but doing so requires replacing `button` elements with anchor elements (with the `a` tag) so the browser sends new HTTP requests to the server. For the prerendered SportsStore application, I am going to do something simpler, which is to disable all of the navigation elements in the prerendered content and leave them to be enabled by the rehydration process. Listing 8.7 adds a property to the `StoreComponent` class that will allow templates to determine when the application is being executed on the server, indicating the prerendering process.

Listing 8.7. Adding a property in the `store.component.ts` file in the `src/app/store` folder

```
import { Component, Signal, computed, signal } 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";
import { PlatformService } from "../platform.service";

@Component({
  selector: "store",
  templateUrl: "store.component.html"
})
export class StoreComponent {
  products: Signal<Product[]>;
  categories: Signal<string[]>;
  selectedCategory = signal<string | undefined>(undefined);
  productsPerPage = signal(4);
  selectedPage = signal(1);
  pagedProducts: Signal<Product[]>;
  pageCount: Signal<number>;

  constructor(private repository: ProductRepository,
    private cart: Cart,
    private router: Router,
    private ps: PlatformService) {

    this.products = computed(() => {
      if (this.selectedCategory() == undefined) {
        return this.repository.products();
      } else {
        return this.repository.products().filter(p =>
          p.category === this.selectedCategory());
      }
    })

    // ...statements omitted for brevity...
  }

  // ...methods omitted for brevity...

  get isServer() { return this.ps.isServer }
}

```

Listing 8.8 uses the new property to disable elements that would allow navigation in the component's template.

Listing 8.8. Disabling 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
        <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)"
        [disabled]="isServer">
        {{cat}}
    </button>
    <button class="btn btn-danger mt-5" routerLink="/admin"
        [disabled]="isServer">
        Admin
    </button>
</div>
</div>
<div class="col-9 p-2 text-dark">
    <div *ngFor="let product of pagedProducts()"
        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)"
                [disabled]="isServer">
                Add To Cart
            </button>
        </div>
    </div>
</div>
<div class="form-inline float-start mr-1">
    <select class="form-control" [value]="productsPerPage()"
        (change)="changePageSize($any($event).target.value)"
        [disabled]="isServer">
        <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()"
        [disabled]="isServer">
        {{page}}
    </button>
</div>
</div>
</div>

```

```
</div>
```

The user will still be able to see these elements, but they will be disabled in the HTML document that is generated by the prerendering process.

8.2.5 Prerendering the application

Run the command shown in listing 8.9 in the SportsStore folder to prerender the application, generating the HTML documents that will be sent to browsers while the JavaScript files are downloaded.

Listing 8.9. Prerendering the application

```
npm run prerender
```

This command creates a static HTML representation of the application in the `dist/SportsStore/browser` folder. There is no option to dynamically build and serve a pre-rendered application, which means that a separate HTTP server is required. Run the command shown in listing 8.10 in the `SportsStore` folder to download and execute the excellent JavaScript `http-server` package. (Make sure that the `npm run json` command is still running).

Listing 8.10. Running an HTTP server to deliver the prerendered application

```
npx http-server@14.1.1 .\dist\SportsStore\browser --port 4500
```

Most web browsers have support for simulating slow networks, and this is an excellent way to test the prerendered application. For Chrome, this feature is available in the Network pane of the F12 developer tools and there are presets for fast and slow 3G networks. Select the fast 3G option and request `http://localhost:4500`. This works best with a clean cache so that the browser is sure to download all of the JavaScript files. You will see the prerendered HTML representation of the application, which is then rehydrated once the JavaScript files are downloaded, activating the previously disabled buttons, as shown in figure 8.1.

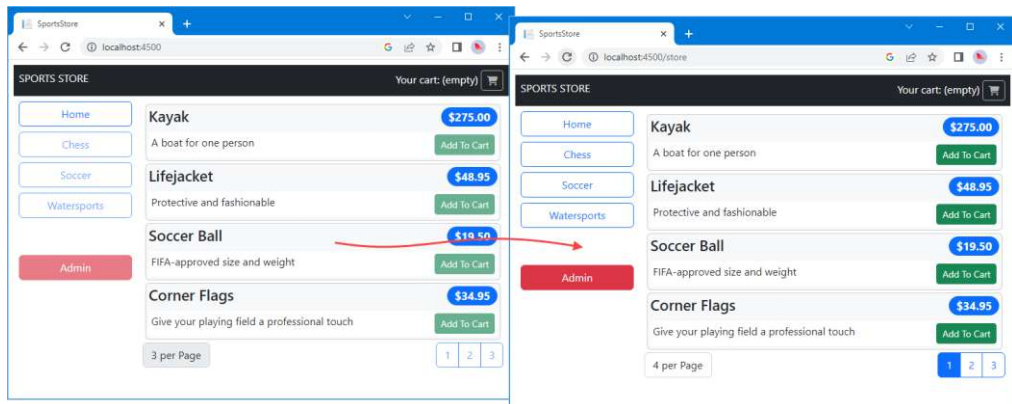


Figure 8.1. Prerendering an application

8.3 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.

8.3.1 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 8.11 in the `SportsStore` folder to download and install the PWA package.

Listing 8.11. Installing a package

```
ng add @angular/pwa@^16
```

8.3.2 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 8.12, 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 8.12. 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", "http://localhost:3500/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.

8.3.3 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 8.13.

Listing 8.13. The contents of the `connection.service.ts` file in the `src/app/model` folder

```
import { Injectable, Signal, WritableSignal, signal } from "@angular/core";
```

```

import { PlatformService } from "../platform.service";

@Injectable()
export class ConnectionService {
  private connectedSignal: WritableSignal<boolean>;

  constructor(ps: PlatformService) {
    this.connectedSignal = signal(ps.isServer
      ? true : window.navigator.onLine);

    if (!ps.isServer) {
      window.addEventListener("online", e =>
        this.connectedSignal.set(window.navigator.onLine));
      window.addEventListener("offline", e =>
        this.connectedSignal.set(window.navigator.onLine));
    }
  }

  get connected() : Signal<boolean> {
    return this.connectedSignal.asReadonly();
  }
}

```

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 8.14, I added the new service to the module for the data model.

Listing 8.14. 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, 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 8.15.

Listing 8.15. 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 {

  constructor(public cart: Cart,
    private connection: ConnectionService) {}

  get connected() {
    return this.connection.connected;
  }
}

```

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 8.16.

Listing 8.16. Reflecting connectivity in the `cartDetail.component.html` file in the `src/app/store` folder

```

...
<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()">
    Checkout
  </button>
</div>
</div>
...

```

8.3.4 Testing the progressive features

Run the command shown in listing 8.17 to build and prerender the application.

Listing 8.17. Prerendering the application

```
npm run prerender
```

Once the build process is complete, run the command shown in listing 8.18 in the `SportsStore` folder to start the web server. (Make sure that the `npm run json` command is still running because this will be the source for data).

Listing 8.18. Running an HTTP server to deliver the prerendered application

```
npx http-server@14.1.1 .\dist\SportsStore\browser --port 4500
```

Use the browser to request `http://localhost:4500`. Once the application has loaded, open the F12 development tools, navigate to the Network tab, click the arrow to the right of No throttling or Fast 3G, 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.

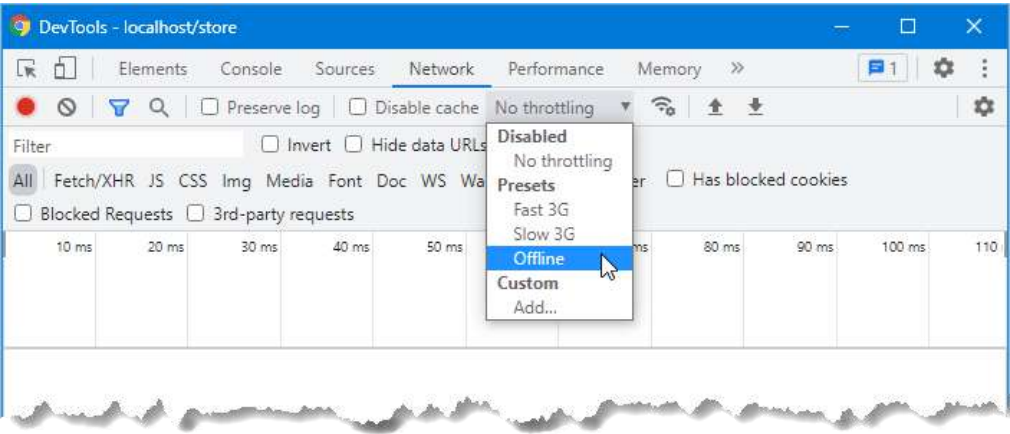


Figure 8.2. Going offline

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. Select the browser’s No throttling option, and the button will be enabled so the user can place an order.

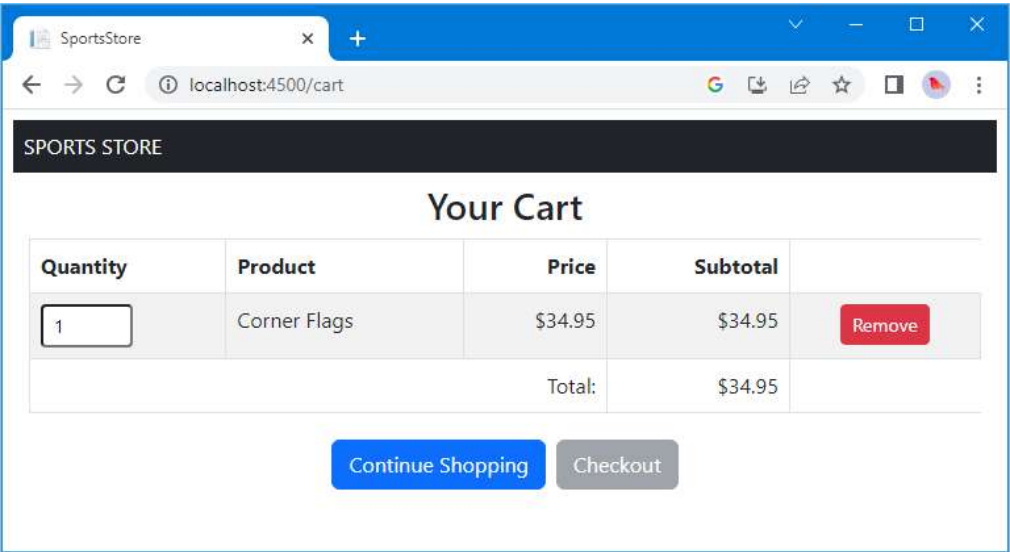


Figure 8.3. Reflecting the connection status in the application

8.4 Preparing the application for deployment

In the sections that follow, I prepare the SportsStore application so that it can be deployed.

8.4.1 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 8.19. When the `json-server` package is configured to use a JSON file, any changes that are made by the application will be persisted.

Listing 8.19. 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": []
}
```

8.4.2 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.

Run the commands shown in listing 8.20 in the `SportsStore` folder to install the packages that are required to create the HTTP/HTTPS server.

Listing 8.20. 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 8.21, 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 8.21. 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 useHttps = false;

const ssloptions = {}

if (useHttps) {
  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/browser"));

app.listen(80,
  () => console.log("HTTP Server running on port 80"));

if (useHttps) {
  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 have a certificate, then you can enable HTTPS by setting the `useHttps` value to `true`. You will still be able to test the application without a certificate, but you won't be able to use the progressive features in deployment.

8.4.3 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 8.22.

Listing 8.22. 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 { map, Observable, from } from "rxjs";
import { Product } from "../product.model";
import { Order } from "../order.model";
import { HttpHeaders } from '@angular/common/http';
import { PlatformService } from "../platform.service";

const PROTOCOL = "http";
const PORT = 3500;

@Injectable()
export class RestDataSource {
  baseUrl: string;
  auth_token?: string;

  constructor(private http: HttpClient, private ps: PlatformService) {
    this.baseUrl = ps.isServer
      ? `http://localhost:${PORT}/`
      : "/api/";
  }
}

```

```
    // ...methods omitted for brevity...  
  }
```

8.5 Building and testing the application

Before building the application, stop the HTTP server and the JSON web server. Run the command shown in listing 8.23 in the `SportsStore` folder to restart the JSON web server with the deployment data. This won't be required once the application has been built, but it will be used to produce the data for the prerendered HTML content.

Listing 8.23. Starting the JSON server

```
npx json-server serverdata.json -p 3500
```

To build the application, run the command shown in listing 8.24 in the `SportsStore` folder.

Listing 8.24. Building the application for production

```
npm run prerender
```

This will build and prerender the application, but with the URLs that are required to obtain data from the same server that delivers the HTML and JavaScript content.

Once the build is ready, run the command shown in listing 8.25 in the `SportsStore` folder to start the HTTP server. (You may have to use `sudo` for this command on Linux and macOS systems).

Listing 8.25. 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 8.4.

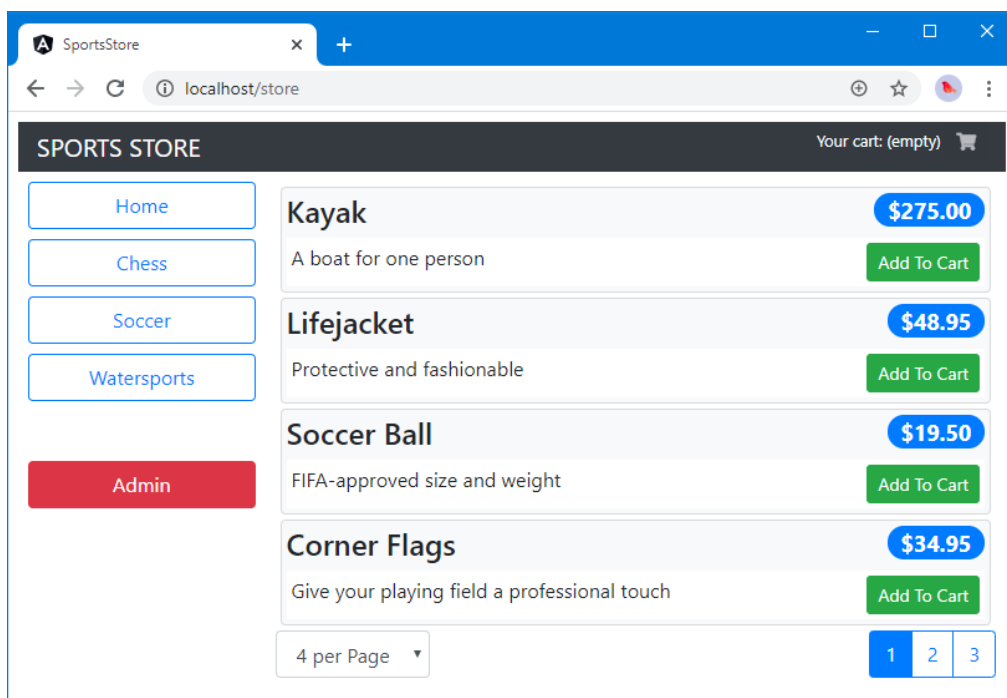


Figure 8.4. Testing the application

8.6 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.

8.6.1 Installing Docker

The first step is to download and install the Docker tools on your development machine, which are available from www.docker.com. 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 edition of Docker Desktop is sufficient for this chapter.

8.6.2 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 8.26.

Listing 8.26. The contents of the deploy-package.json file in the SportsStore folder

```
{
  "devDependencies": {
    "json-server": "0.17.3",
    "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 `devDependencies` section includes the tools required by the production HTTP/HTTPS server. No dependencies are required for the Angular packages because they have been incorporated into the prerendered files.

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.

8.6.3 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 8.27.

Listing 8.27. The contents of the Dockerfile file in the SportsStore folder

```
FROM node:18.14.0

RUN mkdir -p /usr/src/sportsstore

COPY dist/SportsStore/browser /usr/src/sportsstore/dist/SportsStore/browser
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 8.28. This tells Docker to ignore the `node_modules` folder, which is not required in the container and takes a long time to process.

Listing 8.28. The contents of the `.dockerignore` file in the `SportsStore` folder

```
node_modules
```

Run the command shown in listing 8.29 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 8.29. 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.

8.6.4 *Running the application*

Once the image has been created, create and start a new container using the command shown in listing 8.30.

TIP Make sure you stop the test server you started in listing 8.25 before starting the Docker container since both use the same ports to listen for requests.

Listing 8.30. 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 8.5.

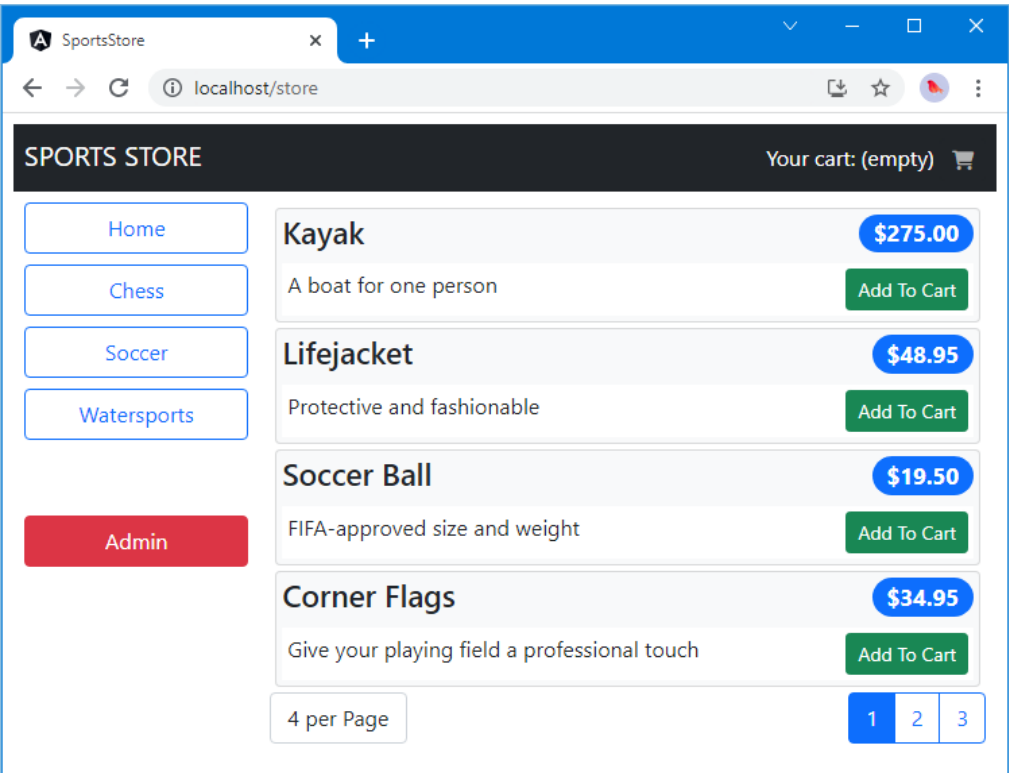


Figure 8.5. Running the containerized SportsStore application

To stop the container, run the command shown in listing 8.31.

Listing 8.31. 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	sportsstore	"docker-entrypoint.s..."	33 seconds ago

Using the value in the Container ID column, run the command shown in listing 8.32.

Listing 8.32. 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.

8.7 *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.

Part II

9

Understanding Angular projects and tools

This chapter covers

- Understanding the contents of a new Angular project
- Using the development server and build process
- Using the linter to check source code in an Angular project
- Understanding how the parts of an Angular application work together
- Creating a production build
- Starting development of an Angular project with a data model

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/manningbooks/pro-angular-16>. See chapter 1 for how to get help if you have problems running the examples.

9.1 *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, the most useful of which are described in table 9.1.

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 part 3.
<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 are supported. Part 3 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.

9.2 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.

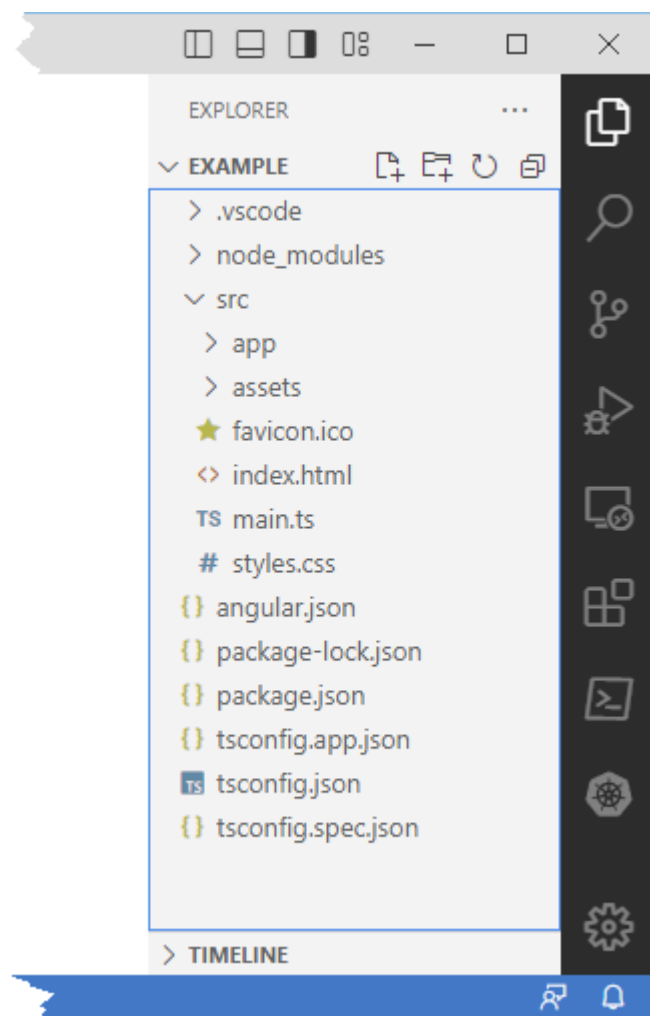


Figure 9.1. The contents of a new Angular project

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
<code>node_modules</code>	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.
<code>src</code>	This folder contains the application’s source code, resources, and configuration files, as described in the “Understanding the Source Code Folder” section.
<code>.editorconfig</code>	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 .
<code>.gitignore</code>	This file contains a list of files and folders that are excluded from version control when using Git.
<code>angular.json</code>	This file contains the configuration for the Angular development tools.
<code>package.json</code>	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.
<code>package-lock.json</code>	This file contains version information for all the packages that are installed in the <code>node_modules</code> folder, as described in the “Understanding the Packages Folder” section.
<code>README.md</code>	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.
<code>tsconfig.json</code>	This file contains the configuration settings for the TypeScript compiler. You don’t need to change the compiler configuration in most Angular projects.
<code>tsconfig.app.json</code>	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.
<code>tsconfig.spec.json</code>	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.

9.2.1 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.

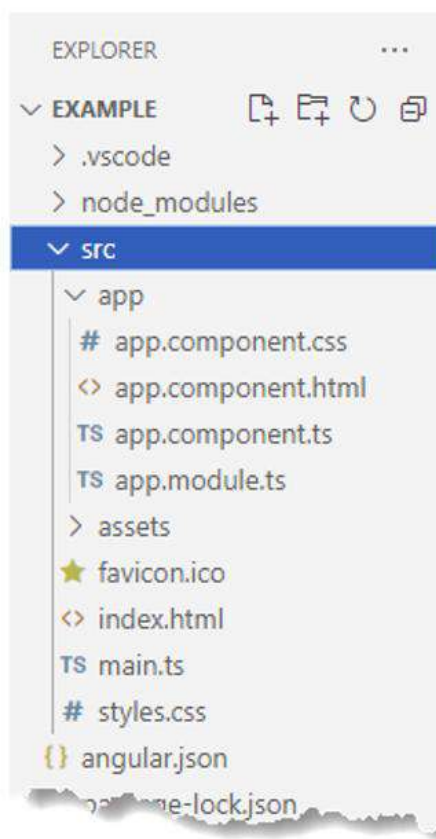


Figure 9.2. The contents of the `src` folder

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.
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.
styles.css	This file is used to define CSS styles that are applied throughout the application.

9.2.2 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": "^16.0.0",
  "@angular/common": "^16.0.0",
  "@angular/compiler": "^16.0.0",
  "@angular/core": "^16.0.0",
```

```

"@angular/forms": "^16.0.0",
"@angular/platform-browser": "^16.0.0",
"@angular/platform-browser-dynamic": "^16.0.0",
"@angular/router": "^16.0.0",
"rxjs": "~7.8.0",
"tslib": "^2.3.0",
"zone.js": "~0.13.0"
},
...

```

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
16.0.0	Expressing a version number directly will accept only the package with the exact matching version number, e.g., 16.0.0.
*	Using an asterisk accepts any version of the package to be installed.
>16.0.0 >=16.0.0	Prefixing a version number with > or >= accepts any version of the package that is greater than or greater than or equal to a given version.
<16.0.0 <=16.0.0	Prefixing a version number with < or <= accepts any version of the package that is less than or less than or equal to a given version.
~16.0.0	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 ~16.0.0 means you will accept version 16.0.1 or 16.0.2 (which would contain patches to version 16.0.0) but not version 16.1.0 (which would be a new minor release).
^16.0.0	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 ^16.0.0 means you will accept versions 16.1.0, and 16.2.0, for example, but not version 16.0.0.

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": "^16.0.0",
  "@angular/cli": "~16.0.0",

```

```

"@angular/compiler-cli": "^16.0.0",
"@types/jasmine": "~4.3.0",
"jasmine-core": "~4.6.0",
"karma": "~6.4.0",
"karma-chrome-launcher": "~3.2.0",
"karma-coverage": "~2.2.0",
"karma-jasmine": "~5.1.0",
"karma-jasmine-html-reporter": "~2.0.0",
"typescript": "~5.0.2"
}
...

```

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 Angular 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 was installed globally so that it can be 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 later chapters in this part of the book.

Table 9.6. The commands in the `scripts` section of the `package.json` file

Name	Description
<code>ng</code>	This command runs the <code>ng</code> command, which provides access to the Angular development tools.
<code>start</code>	This command starts the development tools and is equivalent to the <code>ng serve</code> command.
<code>build</code>	This command performs the production build process.
<code>watch</code>	This command starts the build process in development mode and watches for changes.
<code>test</code>	This command starts the unit testing tools, which are described in part 3, and is equivalent to the <code>ng test</code> 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 `npm` 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 `npm`, 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@16.0.0
```

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@16.0.0 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
  Deep Purple/Amber
  Pink/Blue Grey
  Purple/Green
  Custom
? Set up global Angular Material typography styles? (y/N)
? Include the Angular animations module? (Use arrow keys)
> Include and enable animations
```

```
Include, but disable animations
Do not include
```

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 package.json (1104 bytes)
UPDATE src/app/app.module.ts (423 bytes)
UPDATE angular.json (3487 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>.

9.3 Using the Angular development tools

Projects created using the `ng new` command include 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:

```
...
Initial Chunk Files | Names          | Raw Size
vendor.js           | vendor         | 2.17 MB |
styles.css, styles.js | styles        | 341.65 kB |
polyfills.js        | polyfills     | 328.81 kB |
main.js             | main          | 46.32 kB |
runtime.js          | runtime       | 6.51 kB |
                    | Initial Total | 2.87 MB
```

```
Build at: 2023-05-04T06:48:23.992Z - Hash: 4df1f6585a7d5932 - Time: 12901ms
```

```
** Angular Live Development Server is listening on localhost:4200, open
your browser on http://localhost:4200/ **
...
```

9.3.1 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.

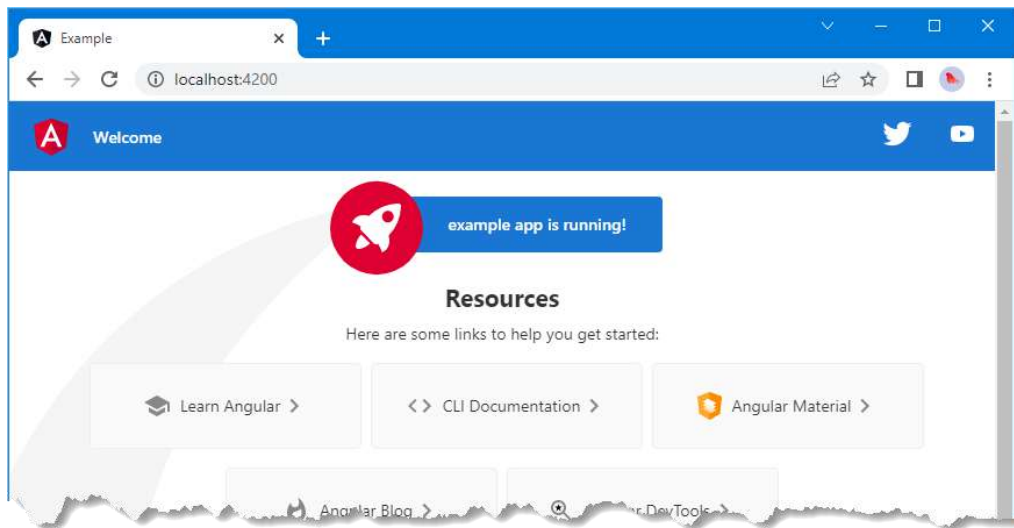


Figure 9.3. Using the HTTP development server

9.3.2 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          | Raw Size
vendor.js           | vendor         | 2.17 MB |
styles.css, styles.js | styles        | 341.65 kB |
polyfills.js        | polyfills     | 328.81 kB |
main.js             | main          | 46.32 kB |
runtime.js          | runtime       | 6.51 kB |
                    | Initial Total | 2.87 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 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      | Raw Size
runtime.js          | runtime   | 6.51 kB |
main.js             | main      | 4.39 kB |
```

```
3 unchanged chunks
Build at: 2023-05-04T07:11:32.801Z - Hash: c9566a8321a0595b - Time: 642ms
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.

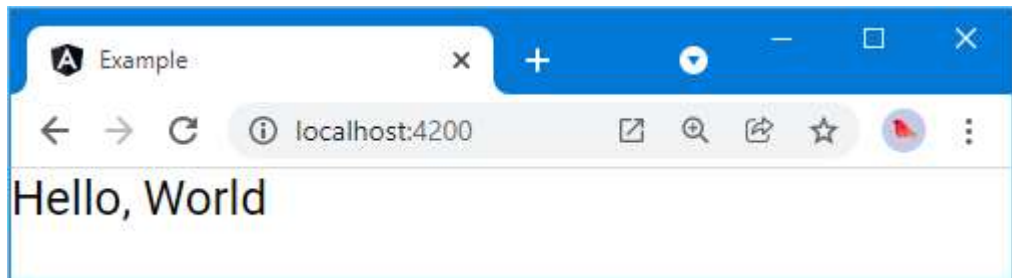


Figure 9.4. Changing a file used in the main.js bundle

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.

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@5.2.3
```

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.json` 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 as a single line and do not introduce additional spaces, quotes, or line returns. This command has been tested with the Bash shell, but you may need to make adjustments for other shells or edit the configuration file directly, as shown in the sidebar.

Listing 9.7. 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 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": [
        "zone.js"
      ],
      "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.)

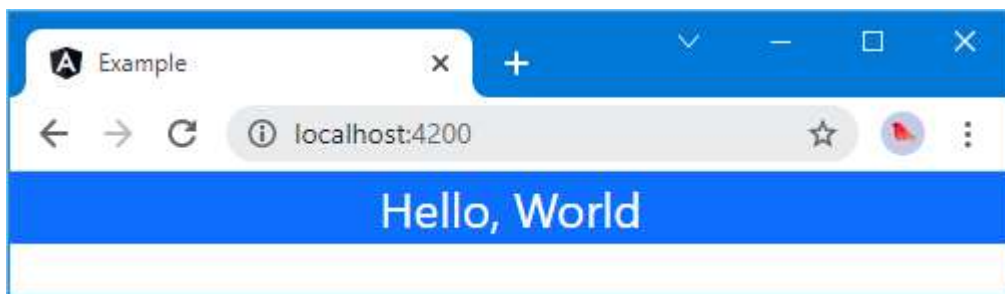


Figure 9.5. Adding a stylesheet

The original bundle contained just the `styles.css` file in the `src` folder, which is empty by default (but 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          | 530.97 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.

9.3.3 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@16.0.0
```

To demonstrate how the linter works, I made two changes to a TypeScript file, as shown in listing 9.12.

Listing 9.12. 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 9.13 in the `example` folder to run the linter.

Listing 9.13. Running the linter

```
ng lint
```

The linter inspects the files in the project and reports any problems that it encounters. The changes in listing 9.12 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 and start with one of
these prefixes: "app" (https://angular.io/guide/styleguide#style-05-02 and
https://angular.io/guide/styleguide#style-02-07) @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.json` file, which is created when the linting package is installed. 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.json` file can be used to disable rules, as shown in listing 9.14.

Listing 9.14. Disabling a rule in the `.eslintrc.json` 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 9.15. I have also commented out the `debugger` statement, even though it will no longer be detected by the linter.

Listing 9.15. 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';
}

```


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.

9.4 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 9.16 in the `example` folder to start them again.

Listing 9.16. 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.

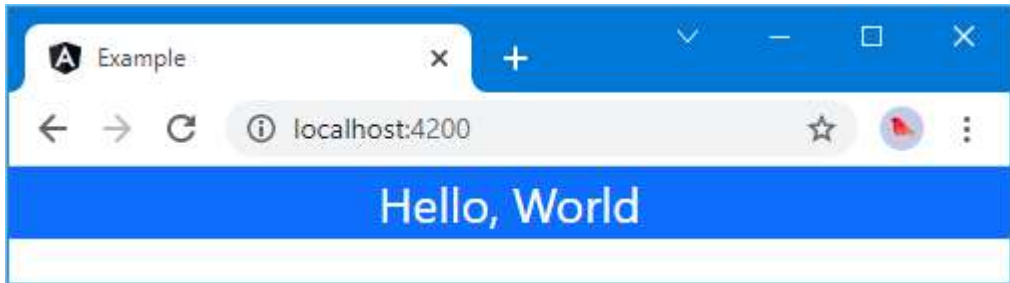


Figure 9.6. Running the example application

In the sections that follow, I explain how the files in the project are combined to produce the response shown in the figure.

9.4.1 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>

```

9.4.2 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 { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';

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 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.

9.4.3 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.

9.4.4 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.

9.4.5 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 Document 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;charset=utf-8;
    base64,eyJ2ZXJzaW9uIjozLCJzb3VyY2VzIjpbXnNvdXJjZVJvb3QiOiIiIiwq= */
</style>
</head>
<body>
  <app-root _ngghost-ng-c1368071736="" ng-version="16.0.0">
    <div _ngcontent-ng-c1368071736=""
      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></html>

```

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.

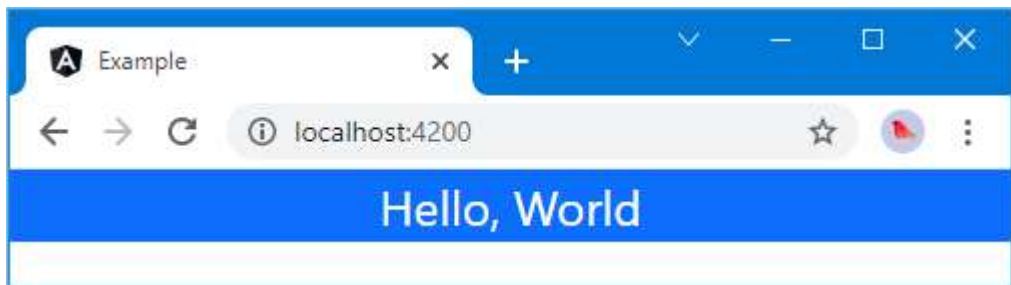


Figure 9.7. Displaying a component's content

9.5 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 compiler and 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 9.17 in the `example` folder.

Listing 9.17. 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 see the details of the bundles that are produced in the messages generated by the compiler:

```
...
Initial Chunk Files | Names | Raw Size | Transfer Size
styles.900b5704e247.css | styles | 292.08 kB | 28.54 kB
main.5661883fdab4.js | main | 155.62 kB | 43.44 kB
polyfills.ac3a5a892ea.js | polyfills | 32.98 kB | 10.64 kB
runtime.92c8b83a3d99.js | runtime | 892 bytes | 506 bytes
...
Initial Total | 481.55 kB | 83.12 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.

9.5.1 Running the production build

To test the production build, run the command shown in listing 9.18 in the `example` folder.

Listing 9.18. Running the production build

```
npx http-server@14.1.1 dist/example --port 5000
```

This command will download and execute 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).

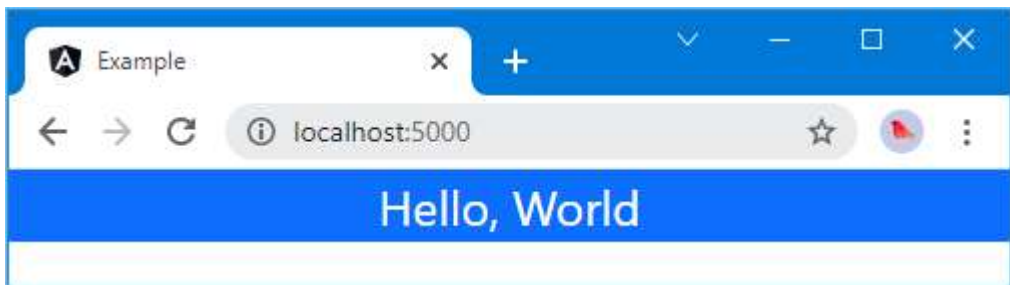


Figure 9.8. Running the production build

Once you have tested the production build, stop the HTTP server using `Control+C`.

9.6 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.

9.6.1 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 9.19.

The name of the file follows the Angular descriptive naming convention. The `product` and `model` sections 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 9.19. 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 13.

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 part 3. 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 9.20.

Listing 9.20. 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 9.21.

Listing 9.21. 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 == undefined) {
            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 methods defined in the repository

Name	Description
<code>getProducts</code>	This method returns an array containing all the <code>Product</code> objects in the model.
<code>getProduct</code>	This method returns a single <code>Product</code> object based on its ID.
<code>saveProduct</code>	This method updates an existing <code>Product</code> object or adds a new one to the model.
<code>deleteProduct</code>	This method removes a <code>Product</code> 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 iterations 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 10, 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.

9.6.2 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 9.22.

Listing 9.22. The contents of the `template.html` file in the `src/app` folder

```
<div class="bg-info text-white p-2">
  There are {{ count }} 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 9.23.

Listing 9.23. 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 {
  private model: Model = new Model();

  get count() : number {
    return this.model.getProducts().length;
  }
}
```

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 getter, called `count`, which produces the number of products in the `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 9.24, I have modified the `index.html` file to introduce an `app` element to match the component's selector.

Listing 9.24. Changing the custom element in the `index.html` file in the `src/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.

9.6.3 *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 9.25, I used the `import` keyword to import the component, and I used the `@NgModule` configuration properties to register the component.

Listing 9.25. 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 9.26 in the `example` folder to start the Angular development tools.

Listing 9.26. 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.

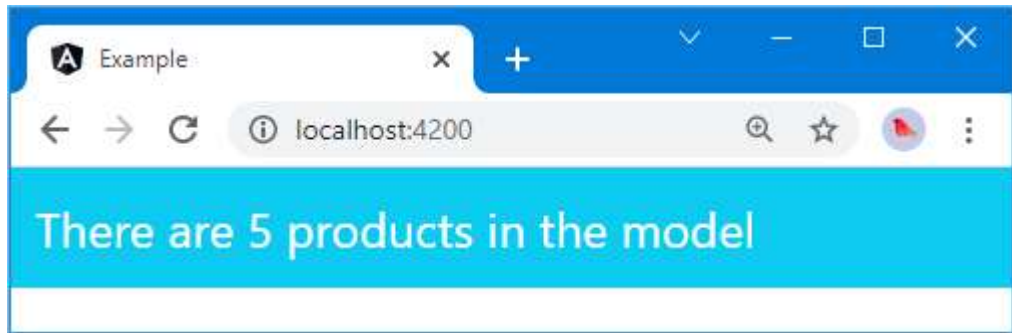


Figure 9.9. The effect of a new component and template

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.

9.7 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.

- Angular projects are created with all of the configuration and code files required to start development.
- The development tools include a web server that builds the application when a change is detected and hot reloads the client.
- The Angular tools perform a fast build during development and a slow optimized build when the application is prepared for deployment.
- A linter can be used to identify common problems with code that are legal TypeScript or JavaScript code, but which can cause issues in production.
- The Angular application has a well-defined set of parts, each of which works together to present content to the user.
- The data model is usually the best place to start development but is the part of the application on which Angular imposes the fewest restrictions.

In the next chapter, I start digging into the Angular features, starting with how Angular responds to changes.

10

Angular reactivity and signals

This chapter covers

- Understanding the way that Angular responds to changes
- Using the new Angular Signals feature to describe data relationships
- Using writable signals and computed signals for efficient change detection
- Using signals with observable sequences of values

In this chapter, I explain how Angular responds to changes in the application state to update the HTML presented to the user. I describe the approach that Angular has conventionally used and introduce a new feature, called *signals*, which can be used to make dealing with changes more efficient. Table 10.1 puts data and reactivity in context.

Table 10.1. Putting reactivity in context

Question	Answer
What is it?	Change detection is the process by which Angular identifies changes in the application state and reacts by updating the HTML presented to the user.
Why is it useful?	Change detection is the basis by which user interaction or data updates are reflected in the HTML content displayed by an application.
How is it used?	Conventionally, the developer simply presents the data that is displayed in templates and Angular has to determine which data values have changed. With the introduction of signals, the developer describes the relationships between data values, which Angular uses to minimize the amount of work required to respond to changes.

Are there any pitfalls or limitations?	Conventional Angular change detection is simple but relatively expensive and requires careful application design to ensure performance. The new signals feature requires more developer effort but means that Angular does less work.
Are there any alternatives?	No. Some form of change detection is required to manage the changes made to the HTML content presented to the user.

Table 10.2 summarizes the chapter.

Table 10.2. Chapter Summary

Problem	Solution	Listing
Update HTML based on user interaction	Update the data values affected by the interaction and let Angular change detection selectively update the HTML elements that have data bindings for those values.	1-7
Describe the relationship between data values to simplify change detection	Use writable signals for mutable data values and computed signals for derived data values.	8-10, 12-14
Execute statements when a signal is updated or recomputed	Create an effect	11
Represent an asynchronous sequence of values	Use an observable	15-17
Use an observable value in a computed signal	Use the <code>toSignal</code> interoperability function	18, 19

10.1 *Preparing for this chapter*

For this chapter, I continue using the `example` project from chapter 9. No changes are required to prepare for this chapter.

TIP You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/manningbooks/pro-angular-16>. See chapter 1 for how to get help if you have problems running the examples.

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.

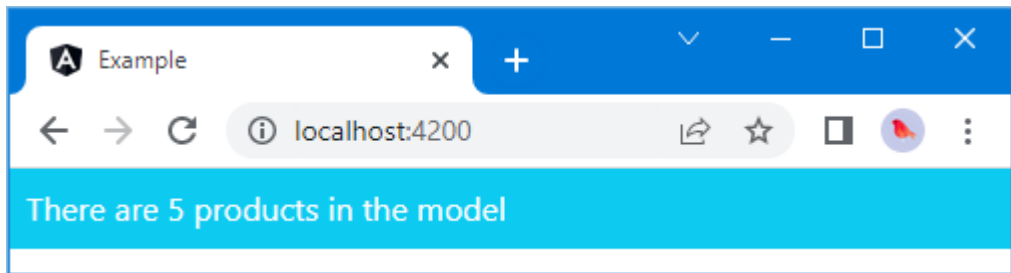


Figure 10.1. Running the example application

10.2 Understanding Angular data flow

Angular is the bridge between a web application's data and the HTML content that is presented to the user. The key building block is the component, which allows the developer to use TypeScript code to select and prepare data, and a template that tells Angular how to display that data in HTML elements. Chapter 16 describes components in depth, but for this chapter, a simple example is enough. Listing 10.1 adds statements to the component already in the example project to prepare data that will be displayed to the user.

Listing 10.1. Preparing data 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 {
  private model: Model = new Model();
  private messages = ["Total", "Price"];
  private index = 0;

  get count(): number {
    return this.model.getProducts().length;
  }

  get total(): string {
    return this.model.getProducts()
      .reduce((total, p) => total + (p.price ?? 0), 0).toFixed(2);
  }

  get message(): string {
    return `${this.messages[this.index]} ${this.total}`;
  }
}
```

The new statements define `total` and `message` getters that produce `string` values. There are also `private` properties that are used in the generation of the display values. Listing 10.2 changes the component's template to display the `message` value to the user.

Listing 10.2. Displaying data values in the template.html file in the src/app folder

```
<div class="bg-info text-white p-2">
  There are {{ count }} products in the model
</div>
<div class="bg-primary text-white p-2">
  {{ message }}
</div>
```

The link between the template and its component is the data binding. Angular provides a range of bindings, and the ones used in listing 10.2, which are denoted by double curly braces (`{{` and `}}`) inserts the `count` and `message` values from the component in the `div` elements. Save the changes and you will see the content shown in figure 10.2.

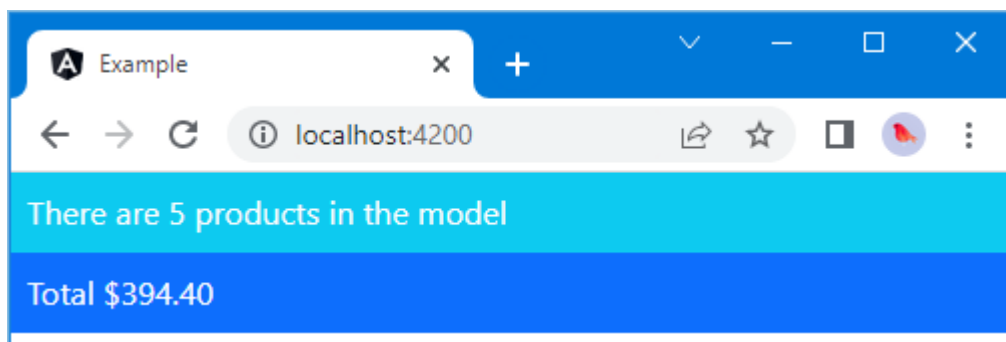


Figure 10.2. Displaying data derived from the data model

Angular has processed the template, evaluated the data bindings, read the values from the component, and use the Document Object Model (DOM) API to create the HTML elements displayed by the browser, as shown in figure 10.3.

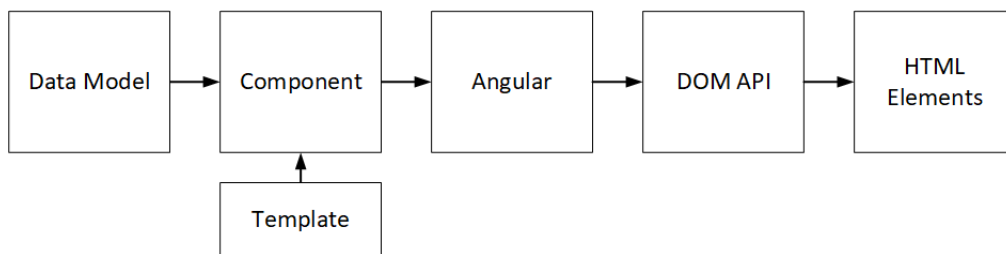


Figure 10.3. Data flowing from the data model

10.2.1 Adding user interaction

Almost every Angular application allows the user to interact with the content that is presented by the user. You will see more complex and realistic types of interaction in later chapters,

including allowing the user to enter data values. For this chapter, all I need is some buttons for the user to click. The first step is to define methods in the component that will be invoked when the buttons are clicked, as shown in listing 10.3.

Listing 10.3. Defining 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 {
  private model: Model = new Model();
  private messages = ["Total", "Price"];
  private index = 0;

  get count(): number {
    return this.model.getProducts().length;
  }

  get total(): string {
    return this.model.getProducts()
      .reduce((total, p) => total + (p.price ?? 0), 0).toFixed(2);
  }

  get message(): string {
    return `${this.messages[this.index]} ${this.total}`;
  }

  toggleMessage() {
    this.index = (this.index + 1) % 2;
  }

  removeProduct() {
    this.model.deleteProduct(this.model.getProducts()[0].id ?? 0);
  }
}
```

When the `toggleMessage` method is invoked, the value of the private `index` property is changed, toggling between 0 and 1 with each click. When the `removeProduct` method is clicked, a product is removed from the data repository.

Listing 10.4 adds button elements to the component's template.

Listing 10.4. Adding elements in the template.html file in the src/app folder

```
<div class="bg-info text-white p-2">
  There are {{ count }} products in the model
</div>
<div class="bg-primary text-white p-2">
  {{ message }}
</div>
<button class="btn btn-primary m-2" (click)="toggleMessage()">
  Toggle
</button>
```

```

<button class="btn btn-primary m-2" (click)="removeProduct()">
  Remove
</button>

```

The button elements are configured with an event binding, the `(click)` attribute, which tells Angular how to respond when the button is clicked. I describe event bindings in detail in chapter 13, but it is enough to understand that clicking the buttons will invoke the `toggleMessage` or `removeProduct` methods.

The final step is to import the Angular features that support user interaction, as shown in listing 10.5. As the name of the Angular module suggests, these features are usually used with HTML form elements, which I describe in detail in part 3.

Listing 10.5. Importing features in the `app.module.ts` file in the `src/app` folder

```

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

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

import { ProductComponent } from './component';

@NgModule({
  declarations: [
    ProductComponent
  ],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule
  ],
  providers: [],
  bootstrap: [ProductComponent]
})
export class AppModule { }

```

Save the changes and you will see the button. Clicking on the buttons causes the content displayed in the browser to change, as shown in figure 10.4.

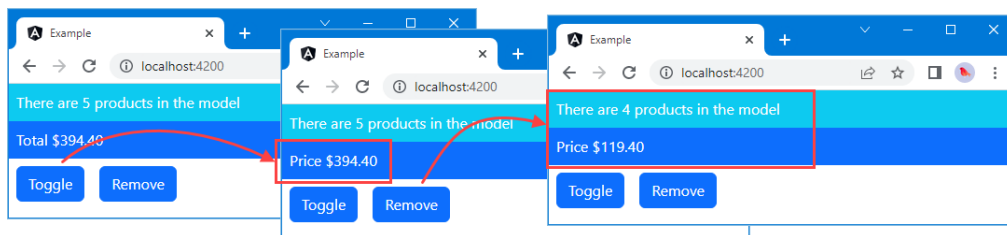


Figure 10.4. Changing content by clicking the button

Angular responds to user interaction by invoking one of the methods defined by the component, which alters the application's data, as shown in figure 10.5.

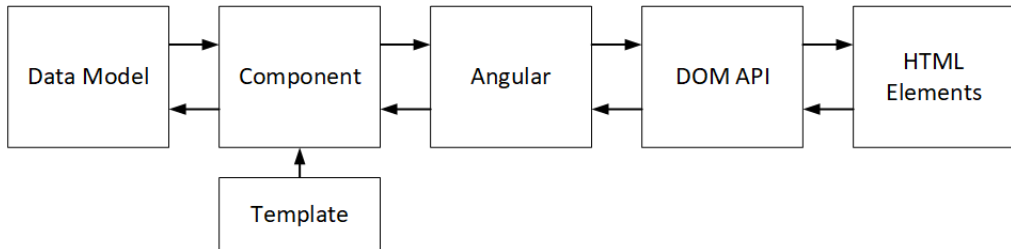


Figure 10.5. The effect of user interaction

When you click either of the buttons, Angular automatically reflects the changed state of the application in the HTML content displayed by the browser.

10.3 Understanding Angular change detection

Angular automatically reflects changes in the application state in the HTML presented to the user. Modern browsers are excellent at dealing with the complexities of displaying HTML, but operations using the browser's DOM API are still relatively slow and expensive to perform and Angular is careful to change as little content as possible when reflecting a state change.

You can get a sense of how this works by making the changes shown in listing 10.6 to the getters and methods defined by the component in the example application.

Listing 10.6. Adding statements 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 {
  private model: Model = new Model();
  private messages = ["Total", "Price"];
  private index = 0;

  get count(): number {
    let result = this.model.getProducts().length;
    console.log(`count value read: ${result}`);
    return result;
  }

  get total(): string {
    let result = this.model.getProducts()
      .reduce((total, p) => total + (p.price ?? 0), 0).toFixed(2);
    console.log(`total value read: ${result}`);
  }
}

```

```

        return result;
    }

    get message(): string {
        let result = `${this.messages[this.index]} ${this.total}`;
        console.log(`message value read: ${result}`);
        return result;
    }

    toggleMessage() {
        console.clear();
        console.log("toggleMessage method invoked");
        this.index = (this.index + 1) % 2;
    }

    removeProduct() {
        console.clear();
        console.log("removeProduct method invoked");
        this.model.deleteProduct(this.model.getProducts()[0].id ?? 0);
    }
}

```

The changes write messages to the browser's JavaScript console when the methods are called and when the value of the getters is read. Save the changes, press F12 to open the browser's developer tool windows, and click the `Toggle` button, which will produce the messages shown in figure 10.6.

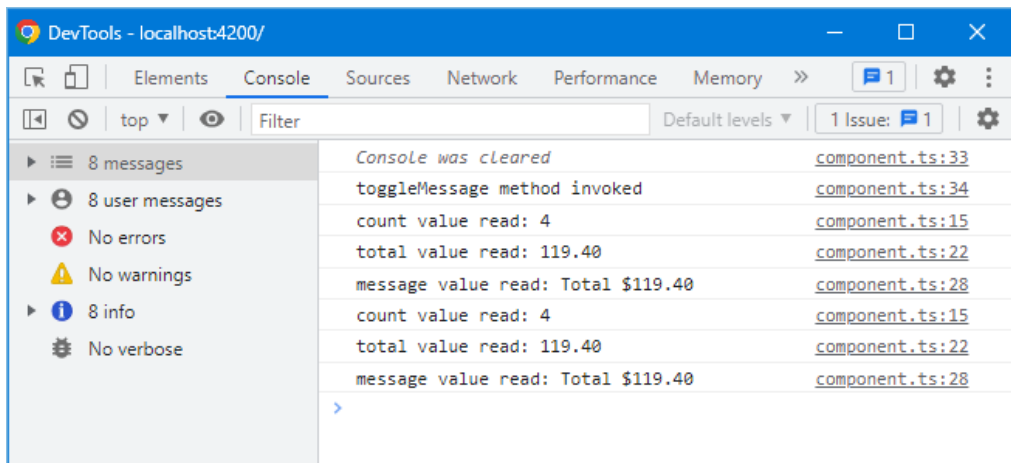


Figure 10.6. Output in the browser's JavaScript console

Angular uses a library called Zone.js, which modifies the standard browser JavaScript API so that operations that are likely to indicate a change in application state – such as using an event handler – trigger its change detection process.

During the change detection process, Angular works its way through the components in the application and evaluates the data bindings expressions in their templates, looking for values that change. When a change is detected, Angular is able to update only the section of the HTML content that is affected by the change, making as efficient use of the browser's DOM API as possible. Here is the output produced by the changes in listing 10.6 when the `Toggle` button is clicked, which may be difficult to make out from the figure:

```
toggleMessage method invoked
count value read: 4
total value read: 119.40
message value read: Total $119.40
count value read: 4
total value read: 119.40
message value read: Total $119.40
```

During development, Angular evaluates the expressions in templates multiple times to check that components produce stable values for use in templates. This is why the `count`, `total`, and `message` values are read twice. This check is not performed when an application is built for deployment and can be ignored.

10.3.1 The advantage of Angular change detection

The advantage of the way that Angular deals with changes is simplicity for the developer. The `zone.js` package seamlessly modifies the JavaScript API to trigger change detection and the entire process happens smoothly and without the developer having to tell Angular what the impact of any particular change will be.

The template in listing 10.4 tells Angular to invoke the `toggleMessage` method when a button is clicked, for example, but there was no need to describe the changes in state caused by the `toggleMessage` method and which bindings in the template would be affected. For every operation that may represent a change, Angular will automatically go through all of the components that are displaying content and evaluate all of the bindings in their templates to make sure that the HTML displayed to the user is up to date.

10.3.2 The disadvantage of Angular change detection

The main problem with change detection is that Angular doesn't have any insight into the relationship between data values or the impact of a change on those values. You can see this in the output that was generated when the `Toggle` button was clicked:

```
...
toggleMessage method invoked
count value read: 4
total value read: 119.40
message value read: Total $119.40
...
```

The `toggleMessage` method is invoked when the button is clicked, which has the effect of modifying the component's `index` property:

```
...
this.index = (this.index + 1) % 2;
...
```


Angular has no way of knowing what the `toggleMessage` method does; it just knows that an event has been dispatched, which means that change detection is required. A new `index` value leads to a new value for the `message` property, but Angular has to evaluate *all* of the template expressions to figure that out. This means that any change will trigger the same detection process, which you can confirm by clicking the `Remove` button, which produces the same sequence of messages in the JavaScript console:

```
...
removeProduct method invoked
count value read: 4
total value read: 119.40
message value read: Total $119.40
...
```

Angular has to evaluate all of the template expressions regardless of what change – if any – has been made to the application’s data.

Once Angular has evaluated all of the template expressions, it can discard any values that have not changed and efficiently update the HTML displayed to the user. But this means that template expressions are evaluated even when the data they rely on hasn’t changed, and that can be a problem for expensive or time-consuming operations, which can be performed only for the result to be discarded. Listing 10.7 modifies the way the `count` value is produced to increase the amount of work that is performed.

Listing 10.7. Performing additional work in the `component.ts` file in the `src/app` folder

```
...
get count(): number {
  let result = this.model.getProducts().length;
  let total = 0;
  for (let i = 0; i < 1000000000; i++) {
    total += 1;
  }
  console.log(`count value read: ${result}`);
  return result;
}
...
```

The changes use a `for` loop to sum integer values to simulate a complex operation required to produce a value. You may need to adjust the maximum value in the loop for your system, but the value in the listing takes a few seconds to complete on my development PC.

When you save the files, you will see that it takes a few seconds for the application to reload and that clicking on either button causes a short pause before the content is updated.

The extra work required to get a `count` value is performed for every change, even those that have no impact on the `count` getter or the template binding that uses it. This undermines the simplicity of development because the developer has to take care to write code that accommodates the evaluation of all template bindings.

10.4 Understanding Angular Signals

Angular is going through a period of transformation that affects how change detection is performed. The key feature is called *signals*, and they change how data is prepared for use in a template to capture the relationships between values.

Signals require the developer to describe the connections between data values, which undercuts the simplicity of the approach conventionally used by Angular but means that Angular knows which values are affected by a change in state and doesn't have to evaluate every template expression.

Signals are available in Angular 16 as a developer preview, which means they can be used in projects, but the API may change in Angular 17, where signals will be integrated more closely into the rest of Angular. But even in preview, signals can still be a useful feature and they are stable enough and useful enough for inclusion in projects today.

Signals are created using the functions described in table 10.3, which are included in the `@angular/core` module.

Table 10.3. The signals functions

Name	Description
<code>signal</code>	This function creates a writable signal, which has a value that can be changed. The result is an object that implements the <code>WritableSignal<T></code> interface, where <code>T</code> represents the type of the data value managed by the signal.
<code>computed</code>	This function creates a read-only signal, whose value is derived from one or more other signals. The result is an object that implements the <code>Signal<T></code> interface, where <code>T</code> represents the type of the data value computed by the signal.
<code>effect</code>	This function creates an effect, which is a function that is executed when one or more specified writable or computed signals changes. The result is an <code>EffectRef</code> object.

10.4.1 Using writable signals

Writable signals are created with an initial value that can be modified to reflect changes in the application state. Listing 10.8 adds a writable signal to the component in the example project.

Listing 10.8. Using a signal in the `component.ts` file in the `src/app` folder

```
import { Component, signal } from "@angular/core";
import { Model } from "../repository.model";

@Component({
  selector: "app",
  templateUrl: "template.html"
})
export class ProductComponent {
  private model: Model = new Model();
  private messages = ["Total", "Price"];
```

```

private index = signal<number>(0);

get count(): number {
    let result = this.model.getProducts().length;
    //let total = 0;
    //for (let i = 0; i < 1000000000; i++) {
    //    total += 1;
    //}
    console.log(`count value read: ${result}`);
    return result;
}

get total(): string {
    let result = this.model.getProducts()
        .reduce((total, p) => total + (p.price ?? 0), 0).toFixed(2);
    console.log(`total value read: ${result}`);
    return result;
}

get message(): string {
    let result = `${this.messages[this.index()]} ${this.total}`;
    console.log(`message value read: ${result}`);
    return result;
}

toggleMessage() {
    console.clear();
    console.log("toggleMessage method invoked");
    //this.index = (this.index + 1) % 2;
    this.index.update(currentVal => (currentVal + 1) % 2);
}

removeProduct() {
    console.clear();
    console.log("removeProduct method invoked");
    this.model.deleteProduct(this.model.getProducts()[0].id ?? 0);
}
}

```

Writable signals are created with the `signal` function, which accepts a type parameter:

```

...
private index = signal<number>(0);
...

```

The type parameter can be omitted and the TypeScript compiler will infer the data type based on the initial value, which is passed to the `signal` function.

The value of a signal is read by invoking the signal as a function, like this:

```

...
let result = `${this.messages[this.index()]} ${this.total}`;
...

```

Unlike a normal JavaScript value, signals are not modified using the assignment operator. Instead, signals are modified using one of the methods described in table 10.4.

Table 10.4. The writable signal methods

Name	Description
set	This method accepts a new value for the signal
update	This method accepts a function that receives the current signal value and returns a new value.
mutate	This method accepts a function that receives the current signal value and modifies it in place.
asReadonly	This method returns a <code>Signal<T></code> object, which provides a read-only version of the signal, as demonstrated in the “Working with Reactive Extensions” section.

I used the `update` method in listing 10.8, which allowed me to set a new value for the index signal based on the existing value:

```
...
this.index.update(currentVal => (currentVal + 1) % 2);
...
```

There is no change to the way the application works yet because writeable signals are just the basic building blocks used to start the description of how data values are related.

Should you use signals in your projects?

Signals are available as a developer preview, which means the API they present may change in Angular 17 and they have not been fully integrated into the framework. But even so, my advice is to adopt signals in any new Angular project. Signals make the relationship between data values explicit, which makes the effect of user interaction easier to understand when maintaining code, and easier for Angular to process at runtime.

I don't recommend rewriting existing projects to use signals, especially if they have been deployed and have entered a stable lifecycle. The disruption caused by wide-ranging changes is likely to introduce defects, which will undermine the benefits offered by signals.

But, regardless of whether you heed or ignore my advice, it is clear that signals represent the future of Angular development and will become more important in future Angular releases.

10.4.2 Using computed signals

A computed signal produces a value that is based – computed – from other signals. Computed signals are read-only and will be recomputed when the value is read after one of the signals used to produce a value changes. (Or, put another way, signals are lazily evaluated, and their values are memoized). Listing 10.9 adds a computed signal to the example component.

Listing 10.9. Adding a computed signal in the `component.ts` file in the `src/app` folder

```
import { Component, computed, signal } from "@angular/core";
```

```

import { Model } from "../repository.model";

@Component({
  selector: "app",
  templateUrl: "template.html"
})
export class ProductComponent {
  private model: Model = new Model();
  private messages = ["Total", "Price"];
  private index = signal<number>(0);

  get count(): number {
    let result = this.model.getProducts().length;
    console.log(`count value read: ${result}`);
    return result;
  }

  get total(): string {
    let result = this.model.getProducts()
      .reduce((total, p) => total + (p.price ?? 0), 0).toFixed(2);
    console.log(`total value read: ${result}`);
    return result;
  }

  // get message(): string {
  //   let result = `${this.messages[this.index()]} ${this.total}`;
  //   console.log(`message value read: ${result}`);
  //   return result;
  // }

  message = computed<string>(() => {
    let result = `${this.messages[this.index()]} ${this.total}`;
    console.log(`message value read: ${result}`);
    return result;
  });

  toggleMessage() {
    console.clear();
    console.log("toggleMessage method invoked");
    this.index.update(currentVal => (currentVal + 1) % 2);
  }

  removeProduct() {
    console.clear();
    console.log("removeProduct method invoked");
    this.model.deleteProduct(this.model.getProducts()[0].id ?? 0);
  }
}

```

Computed signals are created using the `computed` function, which accepts a function that produces a value. When the function is invoked, Angular keeps track of the signals whose values are read in order to understand the relationship between data values. In this example, the `message` computed signal reads the value of the `index` signal, which tells Angular that it should only invoke the `message` signal's function when a change is made to the `index` signal's value.

CAUTION It is important only to rely on other signals when computing a value, otherwise Angular won't realize that a change requires a new value to be produced.

A corresponding change is required to the component's template to read the value of the `message` signal, as shown in listing 10.10.

Listing 10.10. Reading a signal value in the `template.html` file in the `src/app` folder

```
<div class="bg-info text-white p-2">
  There are {{ count }} products in the model
</div>
<div class="bg-primary text-white p-2">
  {{ message() }}
</div>
<button class="btn btn-primary m-2" (click)="toggleMessage()">
  Toggle
</button>
<button class="btn btn-primary m-2" (click)="removeProduct()">
  Remove
</button>
```

Save the changes and click the `Remove` button once the application has been reloaded. The method invoked by the button click doesn't alter the value of the `index` signal, which means that Angular knows the value of the `message` signal can't have changed and doesn't need to be recalculated, which can be seen in the output in the browser's JavaScript console:

```
removeProduct method invoked
...
count value read: 4
count value read: 4
...
```

Signals can coexist alongside the traditional Angular change detection process, which can be seen by clicking the `Toggle` button. The method that is invoked does change the `index` signal, which means that a new value for the `message` signal is computed. But Angular doesn't know how the `count` value is related to the signal and has to read the value to see if it has changed, as shown in the output in the browser's JavaScript console:

```
...
toggleMessage method invoked
count value read: 4
total value read: 119.40
message value read: Price $119.40
count value read: 4
...
```

10.4.3 Using effects

Effects are used to execute statements when the value of another signal changes. Effects are less useful than writable or computed signals because the code they execute occurs outside of the change detection process. One use of effects is to generate logging messages when the value of another signal changes, as shown in listing 10.11.

Listing 10.11. Using an effect signal in the `component.ts` file in the `src/app` folder

```
import { Component, computed, effect, signal } from "@angular/core";
import { Model } from "../repository.model";

@Component({
  selector: "app",
  templateUrl: "template.html"
})
export class ProductComponent {
  private model: Model = new Model();
  private messages = ["Total", "Price"];
  private index = signal<number>(0);

  // ...statements omitted for brevity...

  message = computed<string>(() =>
    `${this.messages[this.index()]} ${this.total}`);

  messageEffect = effect(() =>
    console.log(`message value computed: ${this.message()}`));

  // ...statements omitted for brevity...
}
```

The `effect` function is passed a function that is evaluated at least once so that the signals it depends on can be detected. If any of the signals used by the effect changes, the function will be invoked again.

CAUTION Don't use effects to modify writeable signals or to alter any other part of the application's state. Doing so can cause unexpected results or errors.

Save the changes and click the `Toggle` button once the application has been reloaded by the browser. The button click causes the writable signal named `index` to be modified, which causes the `message` computed signal to change, and this triggers the effect, producing the following output in the browser's JavaScript console (I have highlighted the statement written by the effect signal):

```
...
toggleMessage method invoked
total value read: 394.40
message value computed: Price $394.40
count value read: 5
count value read: 5
...
```

10.4.4 Using signals outside of components

Signals can be used anywhere in an Angular application and shared resources can implement signals to ensure that changes made by one part of the application result in updates elsewhere. Listing 10.12 updates the data repository in the example application to use signals.

Listing 10.12. Using signals in the `repository.model.ts` file in the `src/app` folder

```
import { Product } from "../product.model";
import { SimpleDataSource } from "../datasource.model";
```

```

import { Signal, WritableSignal, signal } from "@angular/core";

export class Model {
  private dataSource: SimpleDataSource;
  private products: WritableSignal<Product[]>;
  private locator = (p: Product, id: number | any) => p.id == id;

  constructor() {
    this.dataSource = new SimpleDataSource();
    this.products = signal(new Array<Product>());
    //this.dataSource.getData().forEach(p => this.products.push(p));
    this.products.mutate(prods =>
      this.dataSource.getData().forEach(p => prods.push(p)));
  }

  get Products(): Signal<Product[]> {
    return this.products.asReadOnly();
  }

  getProduct(id: number): Product | undefined {
    return this.products().find(p => this.locator(p, id));
  }

  saveProduct(product: Product) {
    if (product.id == 0 || product.id == undefined) {
      product.id = this.generateID();
      this.products.mutate(prods => prods.push(product));
    } else {
      this.products.mutate(prods => {
        let index = prods.findIndex(p =>
          this.locator(p, product.id));
        prods.splice(index, 1, product);
      });
    }
  }

  deleteProduct(id: number) {
    this.products.mutate(prods => {
      let index = prods.findIndex(p => this.locator(p, id));
      if (index > -1) {
        prods.splice(index, 1);
      }
    });
  }

  private generateID(): number {
    let candidate = 100;
    while (this.getProduct(candidate) != null) {
      candidate++;
    }
    return candidate;
  }
}

```

I want to expose a signal from the repository so that other parts of the application can compute values that change when the product data changes, but I want to control how changes to the data are applied.

To provide access to the signal without allowing the value to be changed directly, I use the `WritableSignal<T>.asReadOnly` method, which allows me to return a read-only version of the writable signal, like this:

```
...
get Products(): Signal<Product[]> {
    return this.products.asReadOnly();
}
...
```

The writable signal is private and cannot be accessed outside of the `Model` class, ensuring that changes can only be made to the signal by the methods the `Model` class defines. The other changes in listing 10.12 accommodate the introduction of the signal, invoking the signal function to read the current value, and using the `mutate` method to make changes.

TIP I changed the `getProducts` method to a getter. Returning a signal from a method leads to an awkward syntax when the method is invoked and the signal value is read in a single statement: `getProducts()()`.

The next step is to update the component so that it creates computed signals based on the signal provided by the `Model` class, as shown in listing 10.13.

Listing 10.13. Using a signal in the `component.ts` file in the `src/app` folder

```
import { Component, computed, effect, signal } from "@angular/core";
import { Model } from "../repository.model";

@Component({
    selector: "app",
    templateUrl: "template.html"
})
export class ProductComponent {
    private model: Model = new Model();
    private messages = ["Total", "Price"];
    private index = signal<number>(0);

    // get count(): number {
    //     let result = this.model.getProducts().length;
    //     console.log(`count value read: ${result}`);
    //     return result;
    // }

    count = computed<number>(() => this.model.Products().length);

    countEffect = effect(() =>
        console.log(`count value computed: ${this.count()}`));

    get total(): string {
        let result = this.model.Products()
            .reduce((total, p) => total + (p.price ?? 0), 0).toFixed(2);
        console.log(`total value read: ${result}`);
        return result;
    }
}
```

```

message = computed<string>(() =>
  `${this.messages[this.index()]} ${this.total}`);

messageEffect = effect(() =>
  console.log(`message value computed: ${this.message()}`));

toggleMessage() {
  console.clear();
  console.log("toggleMessage method invoked");
  this.index.update(currentVal => (currentVal + 1) % 2);
}

removeProduct() {
  console.clear();
  console.log("removeProduct method invoked");
  this.model.deleteProduct(this.model.Products()[0].id ?? 0);
}
}

```

The final step is to update the template to read the value from the `count` signal, as shown in listing 10.14.

Listing 10.14. Reading a signal value in the `template.html` file in the `src/app` folder

```

<div class="bg-info text-white p-2">
  There are {{ count() }} products in the model
</div>
<div class="bg-primary text-white p-2">
  {{ message() }}
</div>
<button class="btn btn-primary m-2" (click)="toggleMessage()">
  Toggle
</button>
<button class="btn btn-primary m-2" (click)="removeProduct()">
  Remove
</button>

```

Save the changes and wait for the browser to reload the application. Click the `Toggle` button and you will see the following output in the browser's JavaScript console:

```

...
toggleMessage method invoked
total value read: 394.40
message value computed: Price $394.40
...

```

The change to signals means that only the values that have been affected by the `toggleMessage` method are recomputed. Clicking `Remove` button causes a change that affects all of the signals used by the component, producing the following output in the browser's JavaScript console:

```

...
removeProduct method invoked
count value computed: 4
total value read: 119.40
message value computed: Price $119.40
...

```

The repository uses signals but ensures that changes are carefully managed.

10.5 Working with Reactive Extensions

Angular has long relied on a package named RxJS, also known as *Reactive Extensions*, or *ReactiveX*. A basic knowledge of the RxJS building blocks, including how they work with signals, can be useful in Angular development. (See <https://rxjs.dev> for a full description of the features provided by the RxJS package.)

10.5.1 Understanding observables

The key Reactive Extensions building block is an observable, which is represented by the `Observable<T>` class, and which presents a sequence of values that are produced over time. This is most often encountered when making HTTP requests, described in part 3, where the outcome of the request is presented through an `Observable<T>` object. The generic type argument `<T>` denotes the type of data 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 a value is produced, allowing it to respond only when a new value 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 been completed.

NOTE If you are familiar with JavaScript, you may wonder if an `Observable<T>` is the same as a `Promise`, which is the typical way of dealing with asynchronous operations. The key difference is that an `Observable<T>` represents a series of values of type `T`, 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<T>` is `subscribe`, which accepts an object whose properties are set to functions that respond to the sequence of values. The property names and the purpose of the functions are described in table 10.5. If you only need to specify a function that receives values, then you can pass that function as the argument to the `subscribe` method.

Table 10.5. The `Observable<T>` `subscribe` argument properties

Name	Description
<code>next</code>	This function is invoked when a new value is produced.
<code>error</code>	This function is invoked when an error occurs.
<code>complete</code>	This function is invoked when the sequence of values ends.

Add a file named `ticker.model.ts` to the `src/app` folder with the code shown in listing 10.15.

Listing 10.15. The contents of the ticker.model.ts file in the src/app folder

```
import { Observable, interval } from "rxjs";

export class Ticker {

    value: Observable<number> = interval(500);

}
```

The `Ticker` class defines a `value` property that returns an `Observable<number>`, which produces an incremental number value every 500 milliseconds. The `RxJS` package contains a useful `interval` function that creates the `Observable<number>` object and takes care of generating the values. Part 3 contains a more realistic example of using `Observable<T>` to receive data over HTTP, but a simple sequence of numbers is enough for this chapter.

Listing 10.16 revises the application's component to use the new observable with conventional Angular change detection.

Listing 10.16. Using observable values in the component.ts file in the src/app folder

```
import { Component, computed, effect, signal } from "@angular/core";
import { Model } from "../repository.model";
import { Ticker } from "../ticker.model";

@Component({
    selector: "app",
    templateUrl: "template.html"
})
export class ProductComponent {
    private model: Model = new Model();
    private messages = ["Total", "Price"];
    private index = signal<number>(0);
    private ticker = new Ticker();
    tickerValue = 0;

    constructor() {
        this.ticker.value.subscribe(newValue =>
            this.tickerValue = newValue);
    }

    // ...statements omitted for brevity...
}
```

Angular has no way to understand the meaning of the values produced by the observable and so the constructor passes a function to the observable's `subscribe` method that updates a regular property with the latest value. Listing 10.17 incorporates the new property into the template.

Listing 10.17. Displaying a value in the template.html file in the src/app folder

```
<div class="bg-info text-white p-2">
    There are {{ count() }} products in the model
</div>
<div class="bg-primary text-white p-2">
    {{ message() }}
</div>
```

```

<div class="bg-info text-white p-2">
  Ticker: {{ tickerValue }}
</div>
<button class="btn btn-primary m-2" (click)="toggleMessage()">
  Toggle
</button>
<button class="btn btn-primary m-2" (click)="removeProduct()">
  Remove
</button>

```

Using observables in a component can be a little clunky, but it works, and change detection is triggered each time a new value is produced, as shown in figure 10.7.

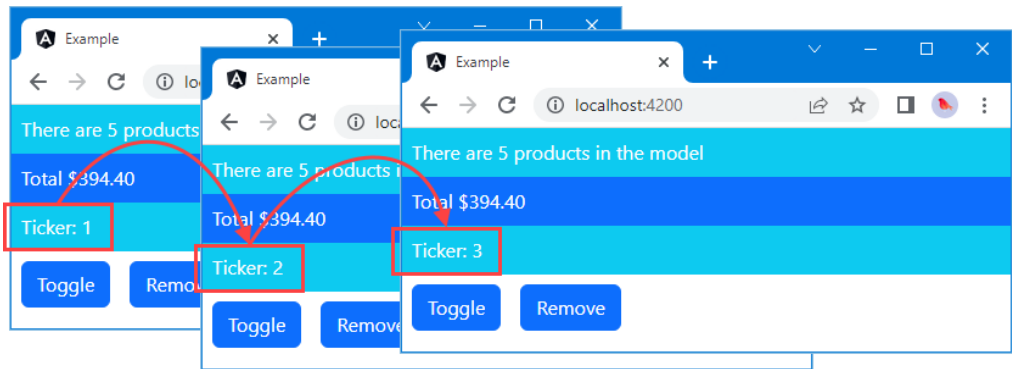


Figure 10.7. Using an observable with conventional change detection

10.5.2 Using observables with signals

Angular provides interoperability functions that allow RxJS observables to be used with signals. This is important because computed signals will only be updated when another signal on which they depend has changed, which means that values taken directly from an observable won't trigger an update.

Listing 10.18 uses the `toSignal` function, which creates a signal from an observable. This allows me to remove the code that processes values produced by the observable and depend on the observed values in a computed signal.

Listing 10.18. Creating a signal in the `component.ts` file in the `src/app` folder

```

import { Component, computed, effect, signal } from "@angular/core";
import { Model } from "../repository.model";
import { Ticker } from "../ticker.model";
import { toSignal } from "@angular/core/rxjs-interop";

@Component({
  selector: "app",
  templateUrl: "template.html"
})
export class ProductComponent {
  private model: Model = new Model();

```

```

private messages = ["Total", "Price"];
private index = signal<number>(0);
private ticker = new Ticker();
tickerValue = toSignal(this.ticker.value, { initialValue: 0 });

// constructor() {
//   this.ticker.value.subscribe(newValue =>
//     this.tickerValue = newValue);
// }

count = computed<number>(() => this.model.Products().length);

countEffect = effect(() =>
  console.log(`count value computed: ${this.count()}`));

get total(): string {
  let result = this.model.Products()
    .reduce((total, p) => total + (p.price ?? 0), 0).toFixed(2);
  console.log(`total value read: ${result}`);
  return result;
}

message = computed<string>(() =>
  `${this.messages[this.index()]} ${this.total} `
  + `Ticker: ${this.tickerValue()}`);

messageEffect = effect(() =>
  console.log(`message value computed: ${this.message()},`));

toggleMessage() {
  console.clear();
  console.log("toggleMessage method invoked");
  this.index.update(currentVal => (currentVal + 1) % 2);
}

removeProduct() {
  console.clear();
  console.log("removeProduct method invoked");
  this.model.deleteProduct(this.model.Products()[0].id ?? 0);
}
}

```

The `toSignal` function is defined in the `@angular/core/rxjs-interop` module and accepts an observable and a configuration object that shapes the way the observables values are managed as a signal. The most useful configuration properties are described in table 10.6.

NOTE There is also a `toObservable` functions that creates an `Observable<T>` from a signal and emits a value each time the signal is changed or is recomputed. See part 3 for examples that rely on this feature.

Table 10.6. Useful `toSignal` configuration object properties

Name	Description
------	-------------

<code>initialValue</code>	This property is used to specify the initial value of the signal, which will be used until the observable produces its next value.
<code>requireSync</code>	This property can be used to specify that the observable will produce a value immediately, which has the effect of changing the type of the signal produced by <code>toSignal</code> to <code>exclude undefined</code> .

In the listing, I used the `initialValue` property to set the value that will be used until the observable produces a value. Listing 10.19 updates the template to read the value from the new signal.

Listing 10.19. Reading a signal value in the `template.html` file in the `src/app` folder

```
<div class="bg-info text-white p-2">
  There are {{ count() }} products in the model
</div>
<div class="bg-primary text-white p-2">
  {{ message() }}
</div>
<div class="bg-info text-white p-2">
  Ticker: {{ tickerValue() }}
</div>
<button class="btn btn-primary m-2" (click)="toggleMessage()">
  Toggle
</button>
<button class="btn btn-primary m-2" (click)="removeProduct()">
  Remove
</button>
```

Save the changes and you will see that the new signal appears twice in the output, showing that its value is being incorporated into a computed signal, as shown in figure 10.8.

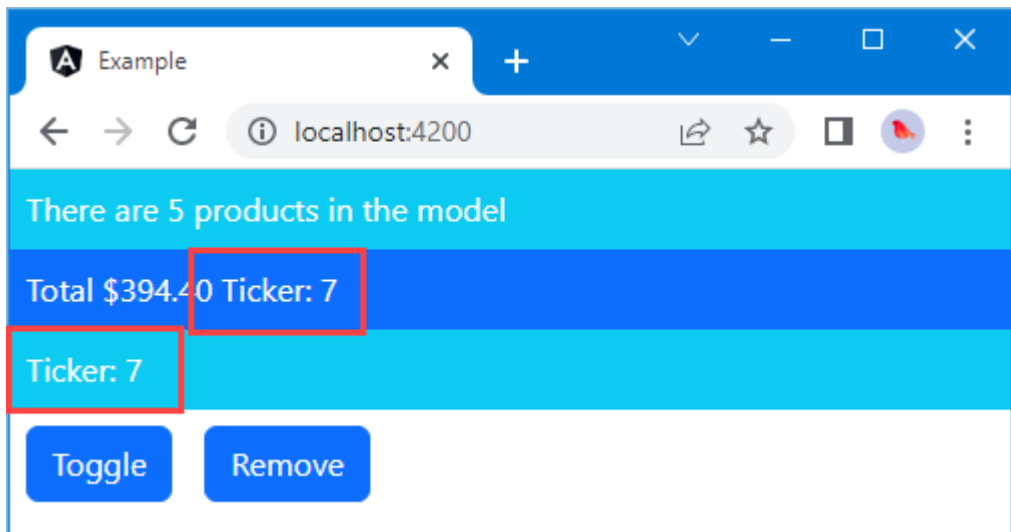


Figure 10.8. Creating a signal from an observable

10.6 Summary

In this chapter, I explained how Angular detects changes in the application state and reflects those changes in the HTML presented to the user. I introduced the new Signals feature, which changes the way that Angular detects and responds to changes.

- Angular conventionally responds to changes by evaluating every template expression used to generate HTML content.
- Angular has to evaluate every template expression because it has no insight into the relationship between data values.
- Signals describe the relationship between data values so the effect of a change can be determined without having to evaluate unaffected template expressions.
- There are three types of signals: writable signals, computed signals, and effects.
- Angular includes interoperability support for combining signals and observables.

In the next chapter, I describe how data bindings work in depth and describe the different types of binding that Angular provides.

11

Using Data Bindings

This chapter covers

- Understanding how to apply a data binding
- Using data bindings to set HTML element attributes
- Using data bindings to assign elements to classes
- Using data bindings to set element style properties
- Using the string interpolation binding

The previous chapter used some simple data bindings to explain how Angular detects changes and updates HTML content. In this chapter, I describe the basic data bindings that Angular provides in-depth 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 11.1 puts data bindings in context.

Table 11.1. 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 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 use signals to simplify change detection.
Are there any alternatives?	No. Data bindings are an essential part of Angular development.

Table 11.2 summarizes the chapter.

Table 11.2. Chapter summary

Problem	Solution	Listing
Displaying data dynamically in the HTML document	Define a data binding	1–5
Configuring an HTML element	Use a standard property or attribute binding	6, 9
Setting the contents of an element	Use a string interpolation binding	7, 8
Configuring the classes to which an element is assigned	Use a class binding	10–14
Configuring the individual styles applied to an element	Use a style binding	15–18

11.1 Preparing for this chapter

For this chapter, I continue using the `example` project from chapter 10. To prepare for this chapter, replace the contents of the `component.ts` file, as shown in listing 11.1.

TIP You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/manningbooks/pro-angular-16>. See chapter 1 for how to get help if you have problems running the examples.

Listing 11.1. Replacing the contents of the `component.ts` file in the `src/app` folder

```
import { Component, computed } from "@angular/core";
import { Model } from "../repository.model";
import { Product } from "../product.model";

@Component({
  selector: "app",
  templateUrl: "template.html"
})
export class ProductComponent {
  private model: Model = new Model();

  products = computed<Product[]>(() => this.model.Products());
```

```

    count = computed<number>(() => this.products().length);

    classes = computed<string>(() =>
      this.count() == 5 ? "bg-success" : "bg-warning");
  }
}

```

The revised component defines computed signals that will be used in the examples in this chapter. Next, replace the contents of the component's template with the static content shown in listing 11.2.

Listing 11.2. Replacing the contents of the template.html file in the src/app folder

```

<div class="bg-info text-white p-2">
  Hello, World
</div>

```

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 11.1, that will be displayed.

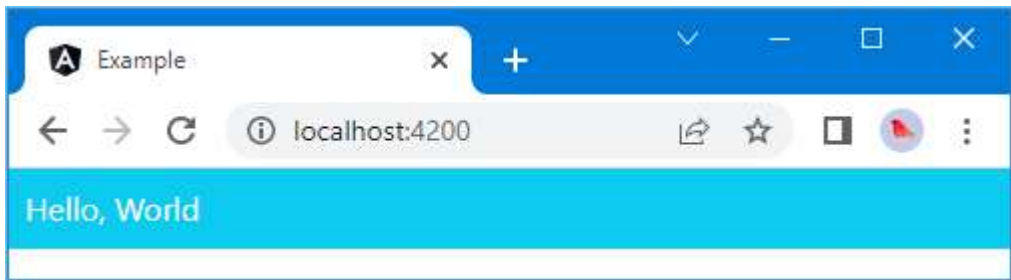


Figure 11.1. Running the example application

11.2 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 12 and 13 for details of other bindings.

To get started, I have applied a one-way data binding in the template, as shown in listing 11.3.

Listing 11.3. Applying a data binding in the template.html file in the src/app folder

```
<div [ngClass]="classes()" >
  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 11.2.

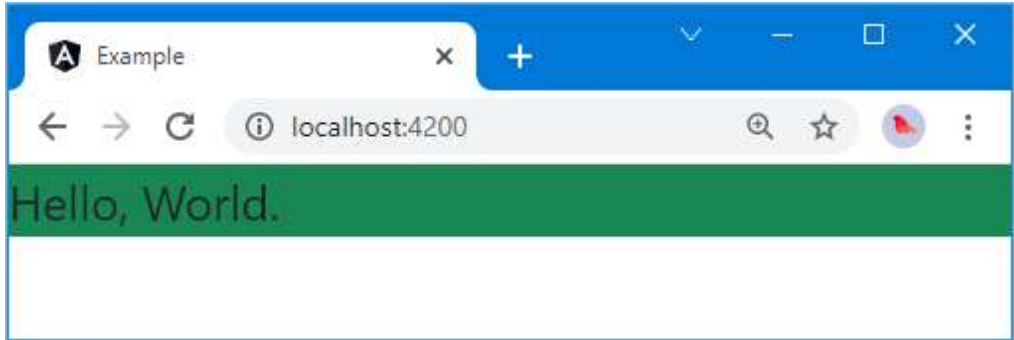


Figure 11.2. Using a one-way data binding

This is a simple example, but it shows the basic structure of a data binding, which is illustrated in figure 11.3.

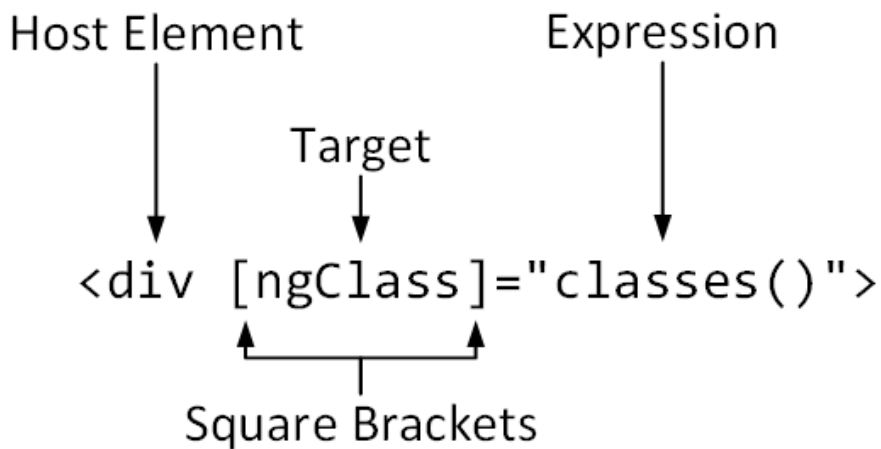


Figure 11.3. The anatomy of a data binding

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 a component's properties and methods can be included in the expression, like the `classes` signal property in the example binding.

Looking at the binding in listing 11.3, 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 reads the value of the component's `classes` property, which was defined at the start of the chapter. This property returns a signal whose value is a string containing a Bootstrap CSS class based on the number of objects in the data model:

```
...
classes = computed<string>(() =>
    this.count() == 5 ? "bg-success" : "bg-warning");
...
```

If there are five objects in the data model, then the value is `bg-success`, which is a Bootstrap class that applies a green background. Otherwise, the value is `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 data bindings. 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 value of the signal assigned to the `classes` property is `bg-success` and produces the result shown in figure 11.2, setting the background to the `div` element.

11.2.1 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 11.3 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 11.3 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 11.3. The basic built-in Angular directives

Name	Description
<code>ngClass</code>	This directive is used to assign host elements to classes, as described in the “Setting Classes and Styles” section.
<code>ngStyle</code>	This directive is used to set individual styles, as described in the “Setting Classes and Styles” section.
<code>ngIf</code>	This directive is used to insert content in the HTML document when its expression evaluates as <code>true</code> , as described in chapter 12.
<code>ngFor</code>	This directive inserts the same content into the HTML document for each item in a data source, as described in chapter 12.
<code>ngSwitch</code> <code>ngSwitchCase</code> <code>ngSwitchDefault</code>	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 12.
<code>ngTemplateOutlet</code>	This directive is used to repeat a block of content, as described in chapter 12.

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 four types of property binding, which are listed in table 11.4, along with the details of where they are described in detail.

Table 11.4. The Angular property bindings

Name	Description
<code>[property]</code>	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.
<code>[attr.name]</code>	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.

<code>[class.name]</code>	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.
<code>[style.name]</code>	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.

11.2.2 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 11.3 can read the value of the `classes` signal 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 11.4 shows an expression that has a literal string value being concatenated with the value read from the `classes` signal.

Listing 11.4. Performing an operation in the template.html file in the src/app folder

```
<div [ngClass]=" 'text-white p-2 ' + classes() " >
  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 three classes: `text-white` 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 11.4 shows the combination of these classes.

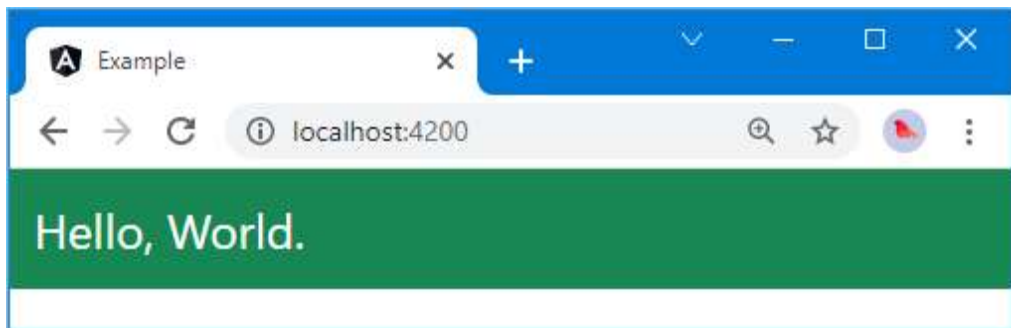


Figure 11.4. Combining classes in a JavaScript expression

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.

11.2.3 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 11.5 adds an element to the template with a binding that doesn't have square brackets.

Listing 11.5. Omitting the brackets in a data binding in the template.html file in the src/app folder

```
<div [ngClass]='text-white p-2 ' + classes()" >
  Hello, World.
</div>
<div ngClass='text-white p-2 ' + classes()' >
  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 ' + classes()"
```

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 don'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 11.5 provides the complete set of brackets, the meaning of each, and where they are described in detail.

Table 11.5. 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.

<code>(target) ="expr"</code>	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 13.
<code>[(target)] ="expr"</code>	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 13.

11.2.4 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.

11.3 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.

11.3.1 Using the standard property binding

The browser uses the Document Object Model (DOM) 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 11.6.

Listing 11.6. Using the standard property binding in the `template.html` file in the `src/app` folder

```
<div [ngClass]="`text-white p-2 ` + classes()" >
  Hello, World.
</div>
<div class="form-group m-2">
  <label>Name:</label>
  <input class="form-control" [value]="products()[1].name" />
</div>
```

The new binding in this example specifies that the `value` property should be bound to the result of an expression that reads an array of data from a signal defined by the component, accesses the object at index 1, and then reads the `name` property.

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. Begin 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 array read from the signal, as shown in figure 11.5.

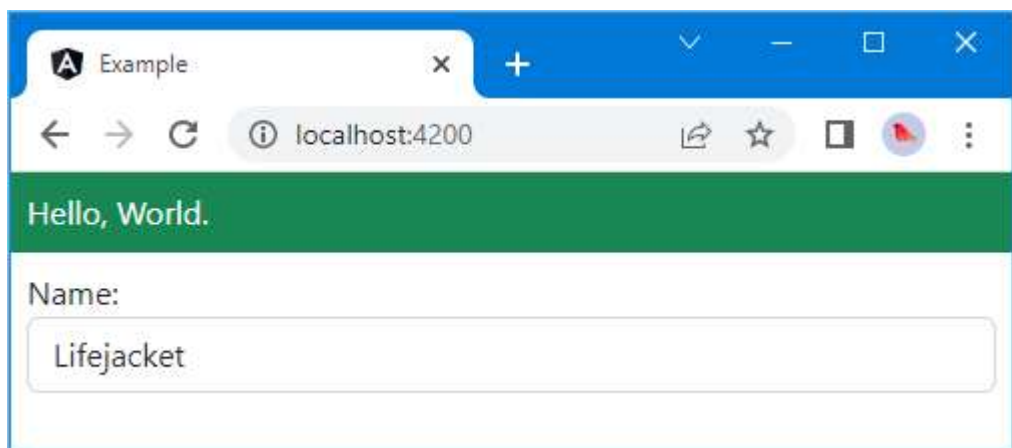


Figure 11.5. Using the standard property binding

11.3.2 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. This is the binding I used in chapter 10 to explain how Angular performs change detection.

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 11.7.

Listing 11.7. Setting an element's content in the `template.html` file in the `src/app` folder

```
<div [ngClass]=" 'text-white p-2 ' + classes() "
    [textContent]=" 'Name: ' + products() [1].name">
</div>
<div class="form-group m-2">
  <label>Name:</label>
  <input class="form-control" [value]="products() [1].name" />
</div>
```

The expression in the new binding concatenates a literal string with a value read from the component 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 11.8.

Listing 11.8. Using the string interpolation binding in the `template.html` file in the `src/app` folder

```
<div [ngClass]=" 'text-white p-2 ' + classes() ">
  Name: {{ products() [1].name }}
</div>
<div class="form-group m-2">
  <label>Name:</label>
  <input class="form-control" [value]="products() [1].name" />
</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 11.7, which is shown in figure 11.6, but the process of writing the binding is simpler and less error-prone.

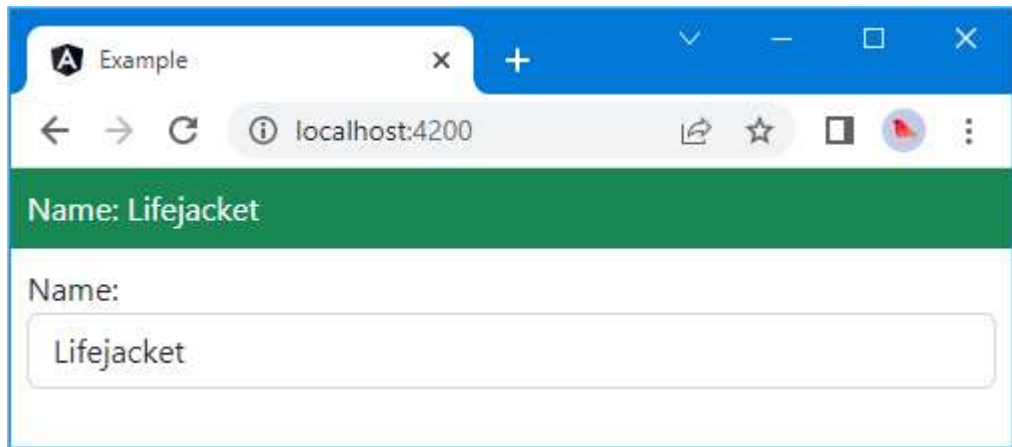


Figure 11.6. Using the string interpolation binding

11.3.3 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 11.9 shows using the attribute binding to set the `colspan` element based on the number of objects in the data model.

Listing 11.9. Using an attribute binding in the `template.html` file in the `src/app` folder

```
<div [ngClass]="text-white p-2 ' + classes()">
  Name: {{ products() [1].name }}
</div>
<div class="form-group m-2">
  <label>Name:</label>
  <input class="form-control" [value]="products() [1].name" />
</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]="count()">
      {{ products() [1].name }}
    </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]="count()">
...
```

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 11.7.

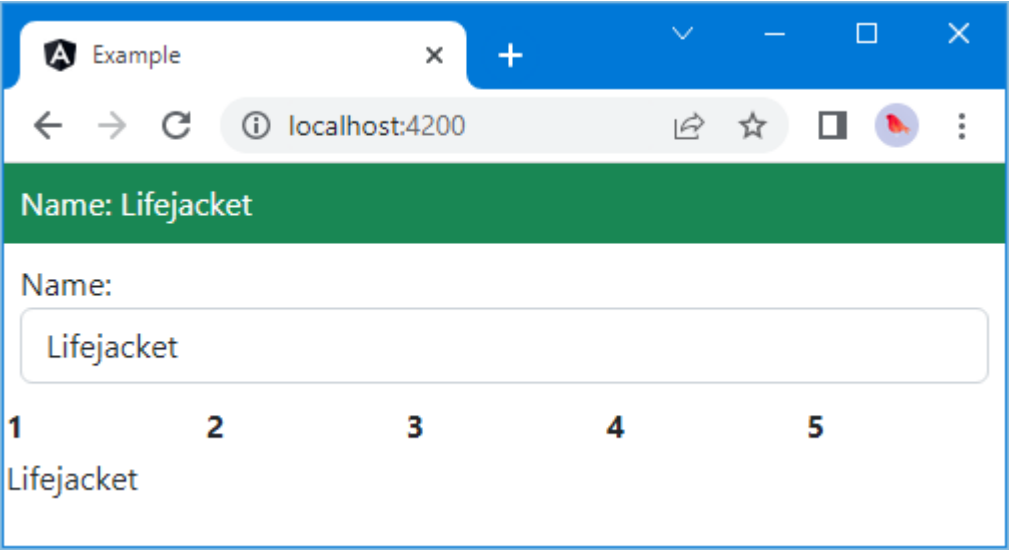


Figure 11.7. Using an attribute binding

11.4 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.

11.4.1 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 11.6, and each works in a slightly different way and is useful in different circumstances, as described in the sections that follow.

Table 11.6. 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 11.10 adds a method to the component that returns a different set of classes based on the `price` property of a selected element in the `products` array.

Listing 11.10. Adding a method in the `component.ts` file in the `src/app` folder

```
import { Component, computed } from "@angular/core";
import { Model } from "../repository.model";
import { Product } from "../product.model";

@Component({
  selector: "app",
  templateUrl: "template.html"
})
export class ProductComponent {
  private model: Model = new Model();

  products = computed<Product[]>(() => this.model.Products());

  count = computed<number>(() => this.products().length);

  classes = computed<string>(() =>
    this.count() == 5 ? "bg-success" : "bg-warning");

  getClasses(key: number) {
    return "p-2 " + (((this.products()[key].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 11.11 replaces the contents of the `template.html` file to show the standard property binding used to set the `class` property of host elements using the component's `getClasses` method.

Listing 11.11. Setting class memberships in the template.html file in the src/app folder

```
<div class="text-white">
  <div [class]="getClasses(0)">
    The first product is {{ products()[0].name }}
  </div>
  <div [class]="getClasses(1)">
    The second product is {{ products()[1].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 11.8.

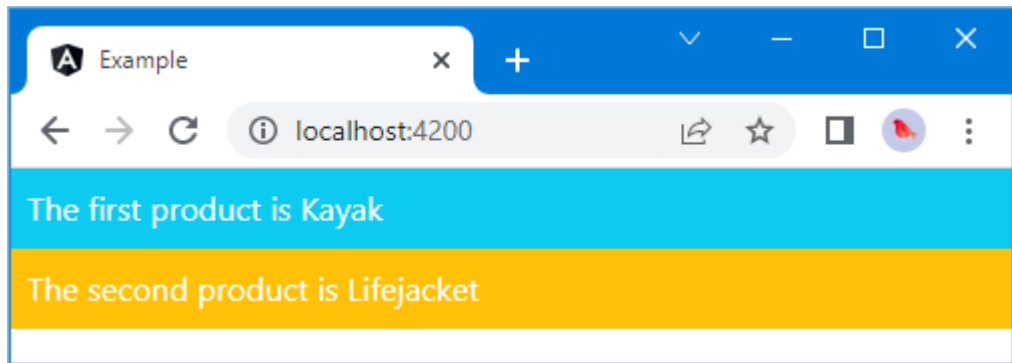


Figure 11.8. Setting class memberships

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 11.12 shows the use of the special class binding.

Listing 11.12. Using the special class binding in the template.html file in the src/app folder

```
<div class="text-white">
  <div [class]="getClasses(0)">
    The first product is {{ products()[0].name }}
  </div>
  <div class="p-2"
    [class.bg-success]="(products()[1].price ?? 0) < 50"
    [class.bg-info]="(products()[1].price ?? 0) >= 50">
    The second product is {{products()[1].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, producing the output shown in figure 11.9.

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.

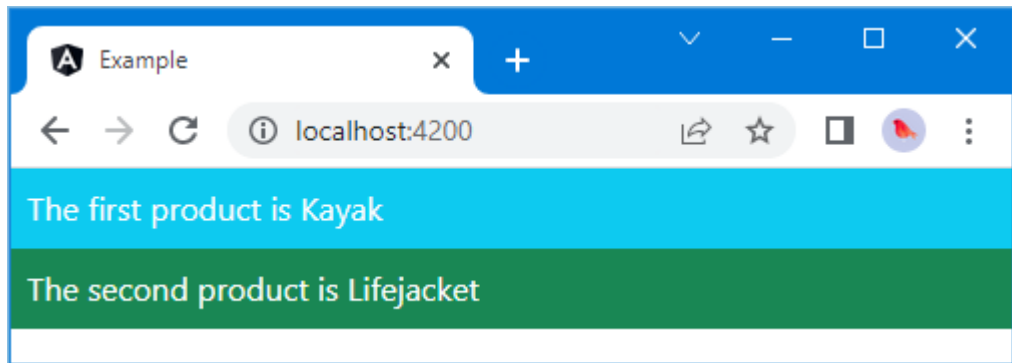


Figure 11.9. Using the special class binding

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 11.7.

Table 11.7. 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 11.13 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 11.13. Returning a class map object in the `component.ts` file in the `src/app` folder

```
import { Component, computed } from "@angular/core";
import { Model } from "../repository.model";
import { Product } from "../product.model";

@Component({
  selector: "app",
  templateUrl: "template.html"
})
export class ProductComponent {
  private model: Model = new Model();

  products = computed<Product[]>(() => this.model.Products());

  count = computed<number>(() => this.products().length);

  classes = computed<string>(() =>
    this.count() == 5 ? "bg-success" : "bg-warning");

  getClasses(key: number) {
    return "p-2 " + (((this.products()[key].price ?? 0) > 50)
      ? "bg-info" : "bg-warning");
  }

  getClassMap(key: number): Object {
    let product = this.products()[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 0, 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 11.14 shows the `getClassMap` and the map objects it returns used as the expression for data bindings that target the `ngClass` directive.

Listing 11.14. Using the `ngClass` directive in the `template.html` file in the `src/app` folder

```
<div class="text-white">
  <div class="p-2" [ngClass]="getClassMap(0)">
    The first product is {{ products()[0].name }}
  </div>
  <div class="p-2" [ngClass]="getClassMap(1)">
    The second product is {{ products()[1].name }}
  </div>
  <div class="p-2"
    [ngClass]="{'bg-success': (products()[2].price ?? 0) < 50,
      'bg-info': (products()[2].price ?? 0) >= 50}">
    The third product is {{products()[2].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-success` classes is tied to the value of the `price` property of a `Product` object, as shown in figure 11.10. Care should be taken with this technique because the expression contains JavaScript logic that cannot be readily tested.

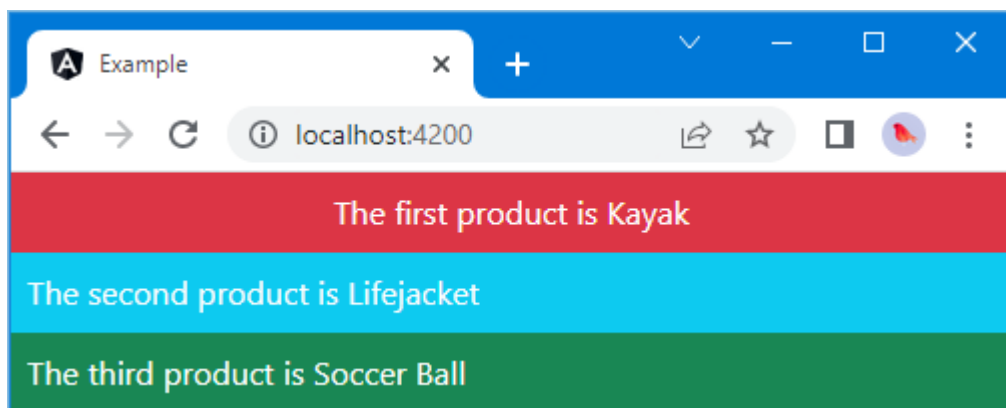


Figure 11.10. Using the `ngClass` directive

11.4.2 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 11.8 and demonstrated in the sections that follow.

Table 11.8. The Angular style bindings

Example	Description
<code><div [style.myStyle]="expr"> </div></code>	This is the standard property binding, which is used to set a single style property to the result of the expression.
<code><div [style.myStyle.units]="expr"> </div></code>	This is the special style binding, which allows the units for the style value to be specified as part of the target.
<code><div [ngStyle]="map"> </div></code>	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 11.15 adds two new properties to the component.

Listing 11.15. Adding properties in the `component.ts` file in the `src/app` folder

```
import { Component, computed } from "@angular/core";
import { Model } from "../repository.model";
import { Product } from "../product.model";

@Component({
  selector: "app",
  templateUrl: "template.html"
})
export class ProductComponent {
  private model: Model = new Model();

  products = computed<Product[]>(() => this.model.Products());

  count = computed<number>(() => this.products().length);

  classes = computed<string>(() =>
    this.count() == 5 ? "bg-success" : "bg-warning");

  getClasses(key: number) {
    return "p-2 " + (((this.products()[key].price ?? 0) > 50)
      ? "bg-info" : "bg-warning");
  }
}
```

```

    getClassMap(key: number): Object {
      let product = this.products()[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 11.16 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 11.16. 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 {{products()[0].name}}
  </div>
  <div class="p-2 bg-info">
    The <span [style.fontSize.px]="fontSizeWithoutUnits">second</span>
    product is {{products()[1].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.fontSize]`).

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 11.11.

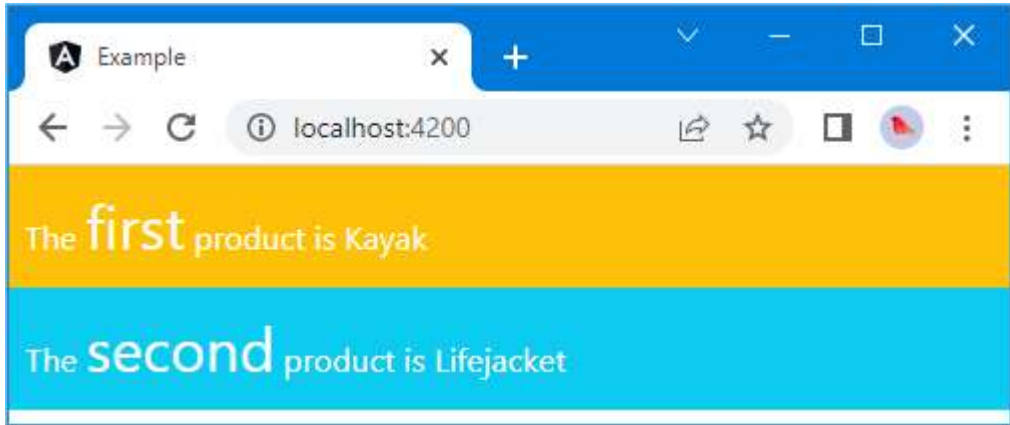


Figure 11.11. Setting individual style properties

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 11.17 shows the addition of a component method that returns a map containing style settings.

Listing 11.17. Creating a style map object in the `component.ts` file in the `src/app` folder

```
import { Component, computed } from "@angular/core";
import { Model } from "../repository.model";
import { Product } from "../product.model";

@Component({
  selector: "app",
  templateUrl: "template.html"
})
export class ProductComponent {
  private model: Model = new Model();

  products = computed<Product[]>(() => this.model.Products());

  count = computed<number>(() => this.products().length);

  classes = computed<string>(() =>
    this.count() == 5 ? "bg-success" : "bg-warning");

  getClasses(key: number) {
    return "p-2 " + (((this.products()[key].price ?? 0) > 50)
      ? "bg-info" : "bg-warning");
  }
}
```

```

getClassMap(key: number): Object {
  let product = this.products()[key];
  return {
    "text-center bg-danger": product.name == "Kayak",
    "bg-info": (product.price ?? 0) < 50
  };
}

fontSizeWithUnits: string = "30px";
fontSizeWithoutUnits: string= "30";

getStyles(key: number) {
  return {
    fontSize: "30px",
    "margin.px": 100,
    color: (this.products()[key].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 0:

```

...
{
  "fontSize":"30px",
  "margin.px":100,
  "color":"red"
}
...

```

Listing 11.18 shows data bindings in the template that use the `ngStyle` directive and whose expressions call the `getStyles` method.

Listing 11.18. 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(0)">first</span>
    product is {{products()[0].name}}
  </div>
  <div class="p-2 bg-info">
    The <span [ngStyle]="getStyles(1)">second</span>
    product is {{products()[1].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 11.12.

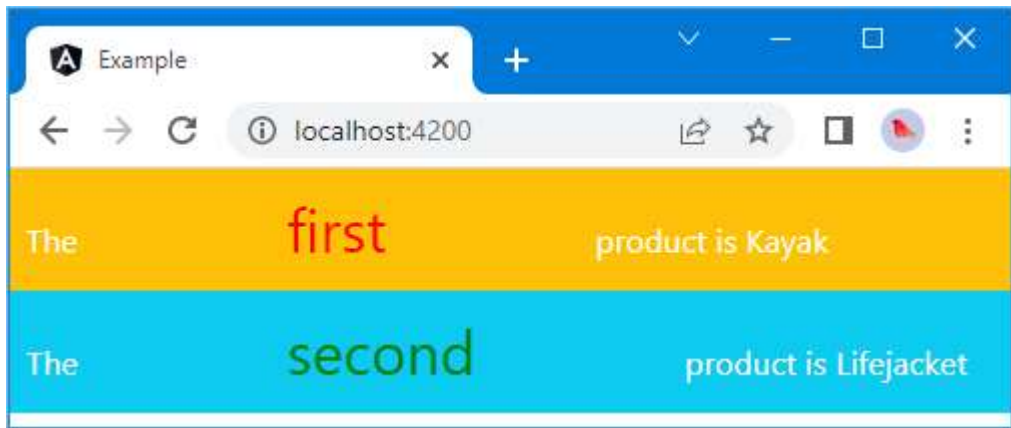


Figure 11.12. Using the `ngStyle` directive

11.5 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.

- One-way data bindings incorporate values provided by the component into the HTML generated from the template.
- Bindings have a host element, a target, and an expression.
- The `ngClass` binding is used to set class memberships for HTML elements.
- The `ngStyle` binding is used to set style properties for HTML elements.
- Classes and styles can also be set using the special property bindings.
- The string interpolation binding is used to configure the `textContent` property of HTML elements.

In the next chapter, I explain how more of the built-in directives work.

12

Using the built-in directives

This chapter covers

- Using the `ngIf` and `ngSwitch` directives to selectively include elements
- Using the `ngFor` directive to generate content for each element in a sequence
- Using micro-templates to repeat content
- Understanding the restrictions Angular places on data binding expressions

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 12.1 puts the built-in template directives in context.

Table 12.1. 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 custom directives—a process that I described in chapters 14 and 15—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 12.2 summarizes the chapter.

Table 12.2. 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	6–13
Repeating a block of content	Use the <code>ngTemplateOutlet</code> directive	14–15
Apply a directive without using an HTML element	Use the <code>ng-container</code> element	16
Preventing template errors	Avoid modifying the application state as a side effect of a data binding expression	17–20
Avoiding context errors	Ensure that data binding expressions use only the properties and methods provided by the template’s component	21–23

12.1 Preparing the example project

This chapter relies on the `example` project from chapter 11. To prepare for this chapter, listing 12.1 simplifies the component class.

TIP You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/manningbooks/pro-angular-16>. See chapter 1 for how to get help if you have problems running the examples.

Listing 12.1. The contents of the `component.ts` file in the `src/app` folder

```
import { Component, computed } from "@angular/core";
import { Model } from "../repository.model";
import { Product } from "../product.model";

@Component({
  selector: "app",
  templateUrl: "template.html"
})
export class ProductComponent {
  private model: Model = new Model();

  targetName: string = "Kayak";

  products = computed<Product[]>(() => this.model.Products());

  count = computed<number>(() => this.products().length);

  product(key: number): Product | undefined {
    return this.model.getProduct(key);
  }

  removeProduct() {
    this.model.deleteProduct(this.model.Products()[0].id ?? 0);
  }
}
```

Listing 12.2 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 12.2. The contents of the `template.html` file in the `src/app` folder

```
<div class="text-white">
  <div class="bg-info p-2">
    There are {{ count() }} products.
  </div>
</div>

<button class="btn btn-info text-white mt-2" (click)="removeProduct()">
  Remove
</button>
```

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 12.1.

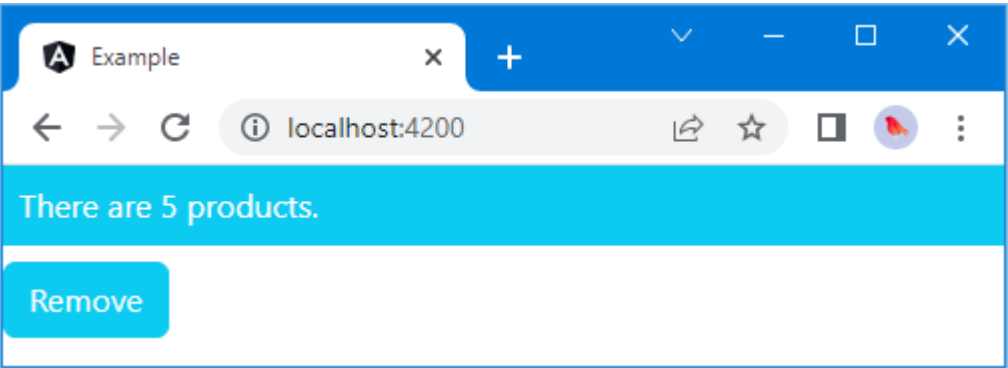


Figure 12.1. Running the example application

12.2 Using the built-in directives

Angular comes with a set of built-in directives that provide features commonly required in web applications. Table 12.3 describes the available directives, which I demonstrate in the sections that follow (except for the `ngClass` and `ngStyle` directives, which are covered in chapter 11).

Table 12.3. The built-in directives

Example	Description
<pre><div *ngIf="expr"> </div></pre>	The <code>ngIf</code> directive is used to include an element and its content in the HTML document if the expression evaluates as <code>true</code> . The asterisk before the directive name indicates that this is a micro-template directive, as described in the “Understanding micro-template directives” sidebar.
<pre><div [ngSwitch]="expr"> </div></pre>	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 11.
<code><div ngStyle="expr"> </div></code>	The <code>ngStyle</code> directive is used to manage styles applied directly to elements (as opposed to applying styles through classes), as described in chapter 11.
<code><ng-template [ngTemplateOutlet] ="myTempl"> </ngtemplate></code>	The <code>ngTemplateOutlet</code> directive is used to repeat a block of content in a template.

12.2.1 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 12.3.

Listing 12.3. 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 {{ count() }} products.
  </div>
  <div *ngIf="count() > 4" class="bg-info p-2 mt-1">
    There are more than 4 products in the model
  </div>
  <div *ngIf="product(0)?.name != 'Kayak'" class="bg-info p-2 mt-1">
    The first product isn't a Kayak
  </div>
</div>

<button class="btn btn-info text-white mt-2" (click)="removeProduct()">
  Remove
</button>
```

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 12.2 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 11, 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`.

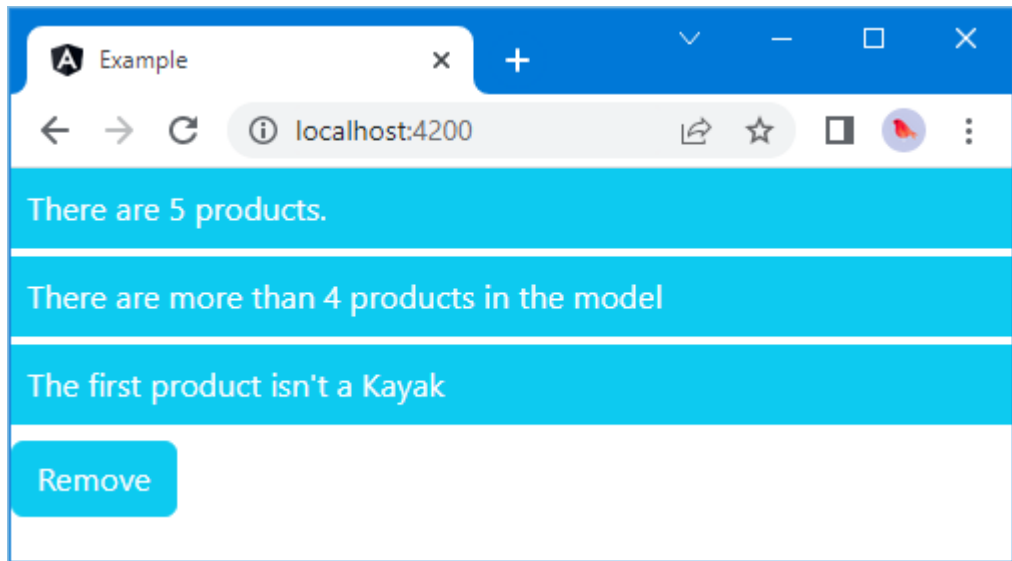


Figure 12.2. Using the `ngIf` directive

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 15 when I show you how to create them.

Listing 12.3 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="count() > 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.

Like all directives, the expression used for `ngIf` will be re-evaluated to reflect changes in the data model. Click the `Remove` button to invoke the component's `removeProduct` method and remove the first object from the data repository.

The effect of modifying the data 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 12.3 shows the change.

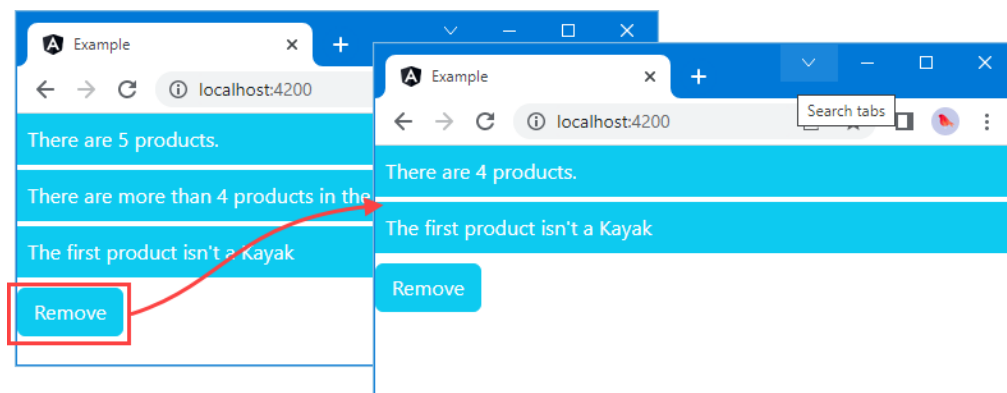


Figure 12.3. The effect of reevaluating directive expressions

12.2.2 Using the `ngSwitch` directive

The `ngSwitch` directive selects one of several elements based on the expression result, similar to a JavaScript `switch` statement. Listing 12.4 shows the `ngSwitch` directive being used to choose an element based on the number of objects in the model.

Listing 12.4. 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 {{ count() }} products.
  </div>
  <div class="bg-info p-2 mt-1" [ngSwitch]="count()">
    <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>

<button class="btn btn-info text-white mt-2" (click)="removeProduct()">
  Remove
```

```
</button>
```

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. Instead, it must be specified within square brackets, like this:

```
...
<div class="bg-info p-2 mt-1" [ngSwitch]="count()">
...

```

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 12.4 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 12.4.

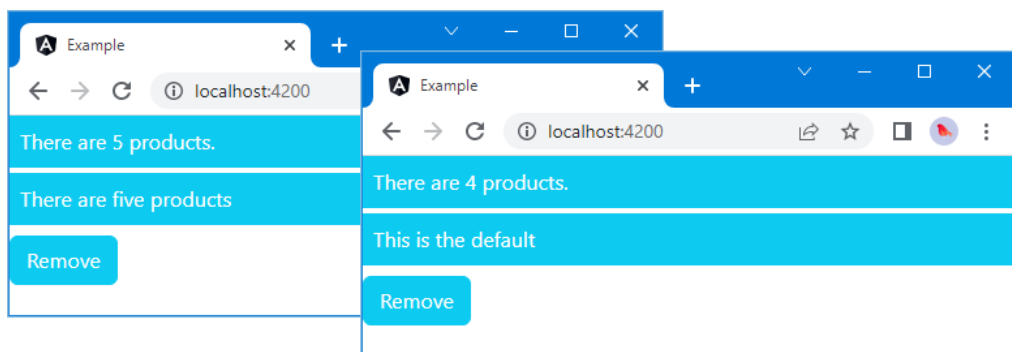


Figure 12.4. Using the `ngSwitch` directive

The `ngSwitch` binding responds to changes in the data model, which you can test by clicking the Remove button. 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 12.4.

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 12.5.

Listing 12.5. 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 {{ count() }} products.
  </div>
  <div class="bg-info p-2 mt-1" [ngSwitch]="product(1)?.name">
    <span *ngSwitchCase="targetName">Kayak</span>
    <span *ngSwitchCase="'Lifejacket'">Lifejacket</span>
    <span *ngSwitchDefault>Other Product</span>
  </div>
</div>

<button class="btn btn-info text-white mt-2" (click)="removeProduct()">
  Remove
</button>
```

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 12.5.

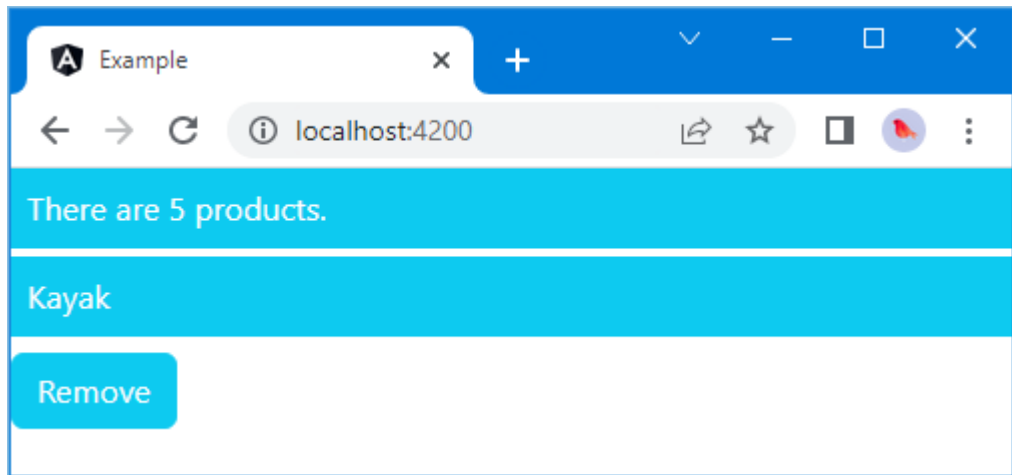


Figure 12.5. Using expressions and literal values with the ngSwitch directive

12.2.3 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 12.6, I have used the ngFor directive to populate a table by generating a row for each `Product` object in the model.

Listing 12.6. 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 {{ count() }} 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 products()">
        <td>{{ item.name }}</td>
        <td>{{ item.category }}</td>
        <td>{{ item.price }}</td>
      </tr>
    </table>
  </div>
</div>

<button class="btn btn-info text-white mt-2" (click)="removeProduct()">
  Remove
</button>
```

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 products()">
```

...

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 products()">
...
```

This example specifies the component’s `products` signal as the source of data, which allows content to be produced for each of the `Product` objects in the model.

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 products()">
...
```

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 `products` signal, 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 12.6, 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 12.6.

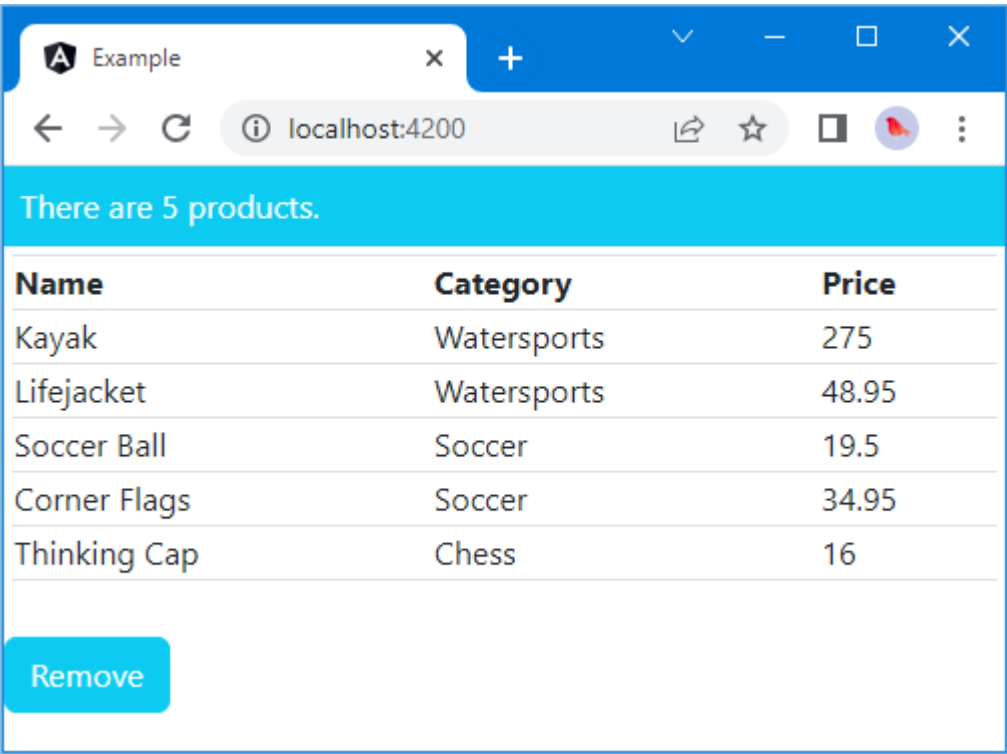


Figure 12.6. Using the ngFor directive to create table rows

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 12.4 and demonstrated in the sections that follow.

Table 12.4. The ngFor local template values

Name	Description
<code>index</code>	This <code>number</code> value is assigned to the position of the current object.
<code>count</code>	This <code>number</code> value is assigned the number of elements in the data source.
<code>odd</code>	This <code>boolean</code> value returns <code>true</code> 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 12.7, 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 12.7. 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 {{count()}} 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 products();
        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>

  <button class="btn btn-info text-white mt-2" (click)="removeProduct()">
    Remove
  </button>
```

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 products(); 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 12.7.

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	16

Remove

Figure 12.7. 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 12.8, the odd value is used to manage the class membership of the `tr` elements in the table.

Listing 12.8. 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 {{count()}} 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 products(); 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>

<button class="btn btn-info text-white mt-2" (click)="removeProduct()">
    Remove
</button>

```

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 products(); 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 12.8.

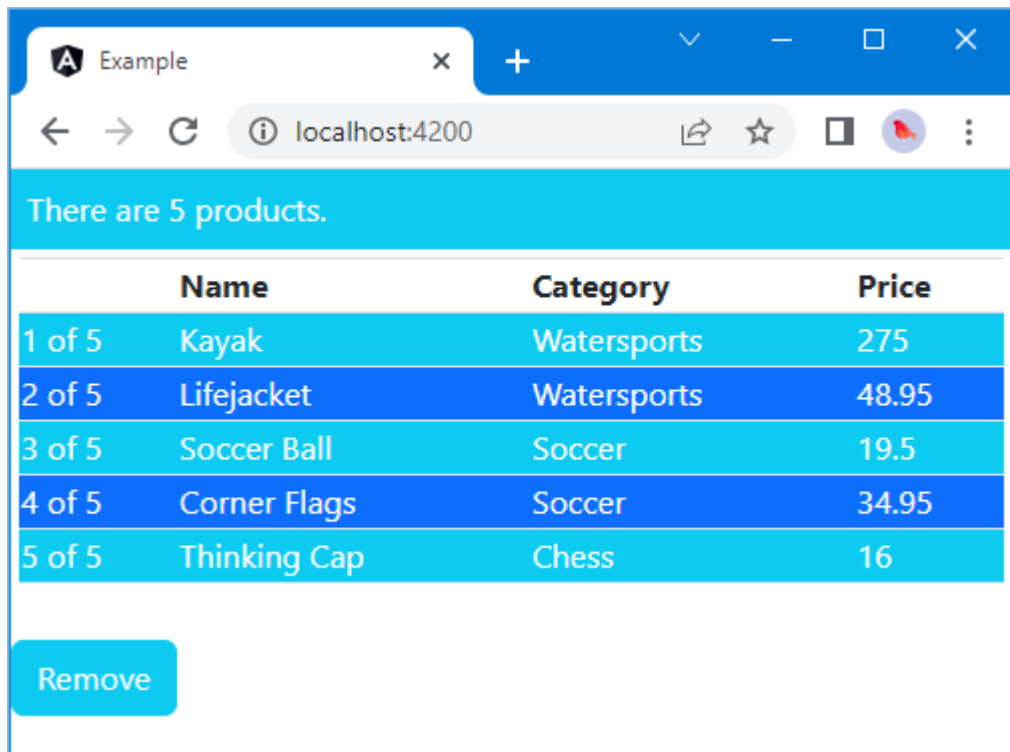


Figure 12.8. Using the odd value

Expanding the *ngFor directive

Notice that in listing 12.8, 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]="products()"
    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 15. 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 12.9 uses these values to treat the first and last objects differently from the others in the sequence.

Listing 12.9. 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 {{count()}} 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 products(); 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>

<button class="btn btn-info text-white mt-2" (click)="removeProduct()">
  Remove
</button>
```

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 12.9.

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	

Remove

Figure 12.9. 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 part 3. 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 12.10.

Listing 12.10. Replacing an object in the component.model.ts file in the src/app folder

```
import { Component, computed } from "@angular/core";
import { Model } from "../repository.model";
import { Product } from "../product.model";

@Component({
  selector: "app",
  templateUrl: "template.html"
})
export class ProductComponent {
  private model: Model = new Model();
```

```

targetName: string = "Kayak";

products = computed<Product[]>(() => this.model.Products());

count = computed<number>(() => this.products().length);

product(key: number): Product | undefined {
    return this.model.getProduct(key);
}

removeProduct() {
    this.model.deleteProduct(this.model.Products()[0].id ?? 0);
}

swapProduct() {
    let p = this.products()[0];
    if (p != null && p.id != null) {
        this.model.deleteProduct(p.id);
        this.model.saveProduct( { ...p, id: 0 });
    }
}
}

```

The `swapProduct` method removes the first object from the array and adds a new object that has the same values for the `name`, `category`, and `price` properties. This is an example of data values being represented by a new object. Listing 12.11 adds a `button` element to the component's template to invoke the new method.

Listing 12.11. Adding a button in the `template.html` file in the `src/app` folder

```

<div class="text-white">
  <div class="bg-info p-2">
    There are {{count()}} 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 products(); 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>

<button class="btn btn-info text-white mt-2" (click)="removeProduct()">
  Remove
</button>

```

```

<button class="btn btn-info text-white mt-2 mx-2" (click)="swapProduct()">
  Swap
</button>

```

Save the changes and click the `Swap` button once the browser reloads the application. The `swapProduct` method is invoked, the data model is modified, and the change detection process is triggered.

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 determining that the data objects it is dealing with have the same values and that it could perform its work more efficiently by moving, rather than recreating, HTML elements.

This problem affects only two sets of 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 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 12.12.

Listing 12.12. Adding a comparison in the component.ts file in the src/app folder

```

import { Component, computed } from "@angular/core";
import { Model } from "../repository.model";
import { Product } from "../product.model";

@Component({
  selector: "app",
  templateUrl: "template.html"
})
export class ProductComponent {
  private model: Model = new Model();

  // ...statements omitted for brevity...

  getKey(index: number, product: Product) {
    return product.name;
  }
}

```

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 `name` value. Telling the `ngFor` expression to use the comparison method is done by adding a `trackBy` term to the expression, as shown in listing 12.13.

Listing 12.13. 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 {{count()}} 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 products(); 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>

  <button class="btn btn-info text-white mt-2" (click)="removeProduct()">
    Remove
  </button>
  <button class="btn btn-info text-white mt-2 mx-2" (click)="swapProduct()">
    Swap
  </button>

```

With this change, the `ngFor` directive will know that the `Product` that is removed from the array using the `swapProduct` method defined in listing 12.13 is considered 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 click the Swap button. 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`.

12.2.4 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. Listing 12.14 replaces the contents of the `template.html` file to show the `ngTemplateOutlet` directive in use.

Listing 12.14. 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 {{count()}} 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 12.10 shows how the directive has used the repeated content.

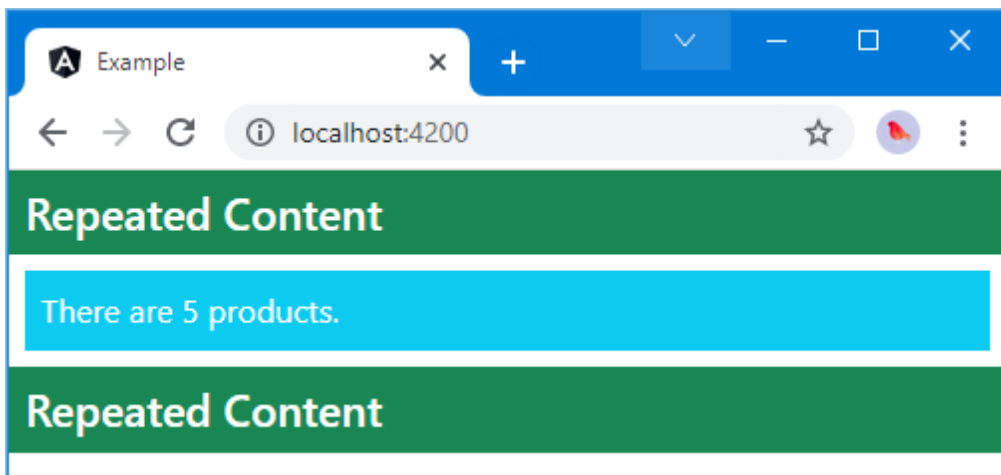


Figure 12.10. Using the `ngTemplateOutlet` directive

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 12.15.

Listing 12.15. 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 {{count()}} 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 an example of an *input property*, which some directives use to receive data values and that I describe in detail in chapter 14. 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 12.11.

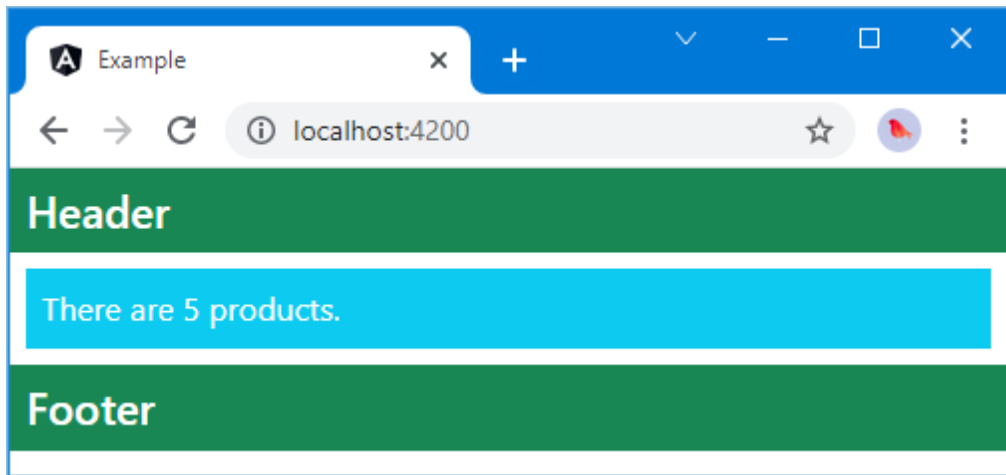


Figure 12.11. Providing context data for repeated content

12.2.5 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 12.16, which replaces the contents of the `template.html` file.

Listing 12.16. Generating content 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 products(); 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 12.12.

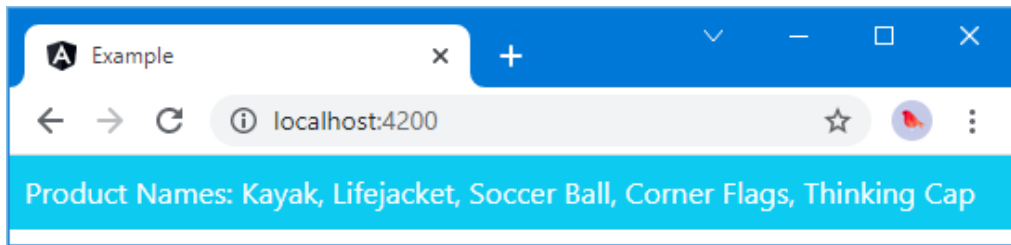


Figure 12.12. Using directives without an HTML element

12.3 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.

12.3.1 Using idempotent expressions

One-way data bindings must be *idempotent*, meaning that they can be evaluated repeatedly without changing the state of the application. This is why Angular repeatedly evaluates template expressions during development, as seen in chapter 10.

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 12.17, I added a `counter` property to the component to help demonstrate.

Listing 12.17. Adding a property in the `component.ts` file in the `src/app` folder

```
import { Component, computed } from "@angular/core";
import { Model } from "../repository.model";
import { Product } from "../product.model";

@Component({
  selector: "app",
  templateUrl: "template.html"
})
export class ProductComponent {
  private model: Model = new Model();

  // ...statements omitted for brevity...

  getKey(index: number, product: Product) {
    return product.name;
  }

  counter: number = 1;
}
```

In listing 12.18, I added a binding whose expression increments the counter when it is evaluated.

Listing 12.18. 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 products(); 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 build tools compile the code, you will see the following error:

```

...
Error: src/app/template.html:9:5 - error NG5002: Parser Error:
  Bindings cannot contain assignments
...

```

Angular will report an error if a data binding expression contains an operator that can be used to perform an assignment, such as `=`, `+=`, `--`, `++`, and `--`.

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 12.19 adds a property to the component that removes and returns a `Product` object from the model array.

Listing 12.19. Modifying data in the component.ts file in the src/app folder

```

import { Component, computed } from "@angular/core";
import { Model } from "../repository.model";
import { Product } from "../product.model";

@Component({
  selector: "app",
  templateUrl: "template.html"
})
export class ProductComponent {
  private model: Model = new Model();

  // ...statements omitted for brevity...

  counter: number = 1;

  get nextProduct(): Product | undefined {
    this.removeProduct();
    return this.products()[0];
  }
}

```

In listing 12.20, you can see the data binding that I used to read the `nextProduct` property.

Listing 12.20. 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 products(); let last = last">
    {{ item.name}}<ng-container *ngIf="!last">,</ng-container>
  </ng-container>
</div>

```

```

        </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: NG0600: Writing to signals is not allowed in a `computed` or
an `effect` by default.
...

```

Angular has detected the attempt to make a change while producing a value for inclusion in a template and generated an error.

12.3.2 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 in earlier chapters. 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 12.21 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 12.21. Using 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 products(); 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(product(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 12.22.

Listing 12.22. Defining a method in the component.ts file in the src/app folder

```
import { Component, computed } from "@angular/core";
import { Model } from "../repository.model";
import { Product } from "../product.model";

@Component({
  selector: "app",
  templateUrl: "template.html"
})
export class ProductComponent {
  private model: Model = new Model();

  // ...statements omitted for brevity...

  get nextProduct(): Product | undefined {
    this.removeProduct();
    return this.products()[0];
  }

  getProductPrice(index: number): number {
    return Math.floor(this.product(index)?.price ?? 0);
  }
}
```

In listing 12.23, I have changed the data binding in the template to use the newly defined method.

Listing 12.23. Accessing a method 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 products(); 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 12.13.

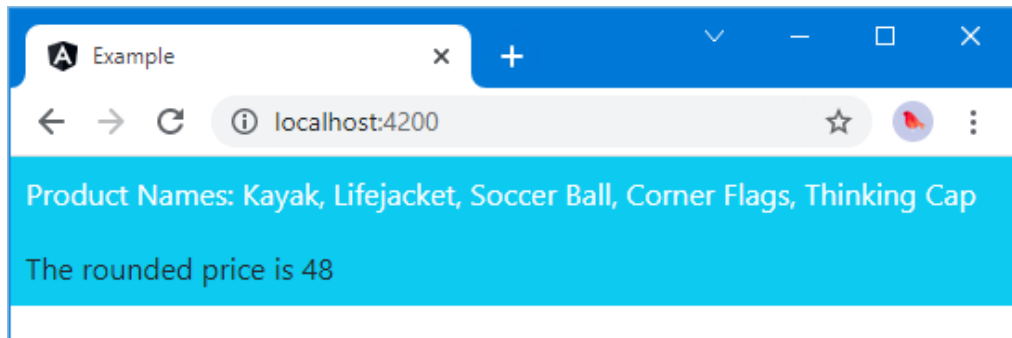


Figure 12.13. Accessing global namespace functionality

12.4 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.

- The `ngIf` directive is used to selectively include template elements in the output.
- The `ngSwitch` directive is used to choose between sets of elements.
- The `ngFor` directive is used to repeatedly generate content for the values in a sequence.
- The `ngTemplateOutlet` directive is used to apply a template to create repeated content.
- Data bindings are evaluated in the context of their component, which should not alter the state of the application.

In the next chapter, I describe how data bindings are used for events and form elements.

13

Using events and forms

This chapter covers

- Using bindings to respond to events
- Using template references
- Using two-way bindings to synchronize component properties and HTML elements
- Validating form data and displaying validation error messages

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 13.1 puts events and forms in context.

Table 13.1. 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 throughout the chapter 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 13.2 summarizes the chapter.

Table 13.2. Chapter summary

Problem	Solution	Listing
Responding to an event	Use an event binding	1–5
Getting details of an event	Use the <code>\$event</code> object	6–8
Referring to elements in the template	Define template variables	9
Enabling the flow of data in both directions between the element and the component	Use a two-way data binding	10–12
Capturing user input	Use an HTML form	13, 14
Validating the data provided by the user	Perform form validation	15–26

13.1 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/manningbooks/pro-angular-16>. See chapter 1 for how to get help if you have problems running the examples.

To prepare for this chapter, listing 13.1 simplifies the component class and adds a writable signal, named `selectedProduct`.

Listing 13.1. Simplifying the component in the `component.ts` file in the `src/app` folder

```
import { Component, computed, signal } from "@angular/core";
import { Model } from "../repository.model";
import { Product } from "../product.model";

@Component({
  selector: "app",
  templateUrl: "template.html"
```



```

    })
    export class ProductComponent {
        private model: Model = new Model();

        products = computed<Product[]>(() => this.model.Products());

        count = computed<number>(() => this.products().length);

        product(key: number): Product | undefined {
            return this.model.getProduct(key);
        }

        selectedProduct = signal<string | undefined>(undefined);
    }

```

Listing 13.2 simplifies the component's template, leaving just a table that is populated using the `ngFor` directive.

Listing 13.2. Simplifying the template in the `template.html` file in the `src/app` folder

```

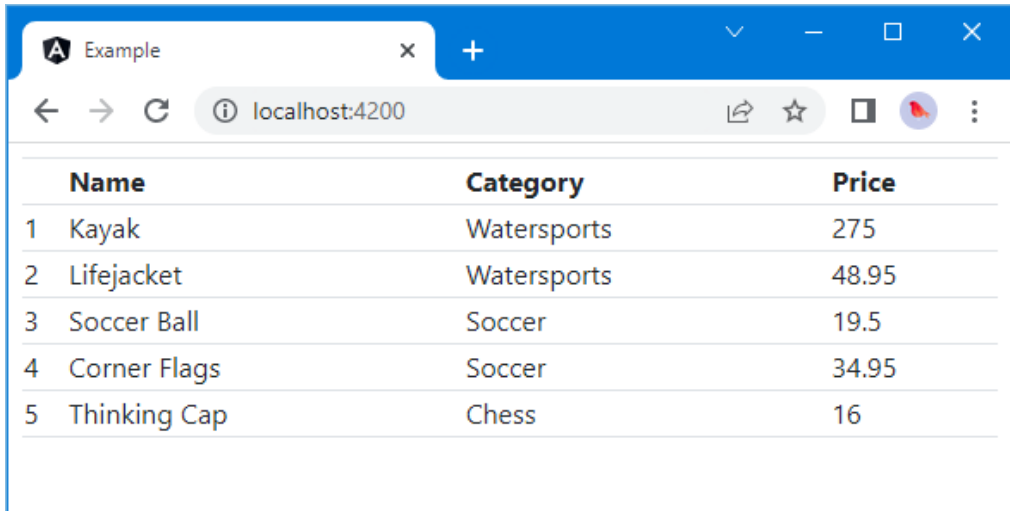
<div class="p-2">
  <table class="table table-sm table-bordered">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th></tr>
    <tr *ngFor="let item of products(); let i = index">
      <td>{{i + 1}}</td>
      <td>{{item.name}}</td>
      <td>{{item.category}}</td>
      <td>{{item.price}}</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 13.1.



	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 13.1. Running the example application

13.2 Using the event binding

The *event binding* is used to respond to the events sent by the host element. Listing 13.3 demonstrates the event binding, which allows a user to interact with an Angular application.

Listing 13.3. 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>
  <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 products(); let i = index">
      <td (mouseover)="selectedProduct.set(item.name)">{{i + 1}}</td>
      <td>{{item.name}}</td>
      <td>{{item.category}}</td>
      <td>{{item.price}}</td>
    </tr>
  </table>
</div>
```

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 13.2.

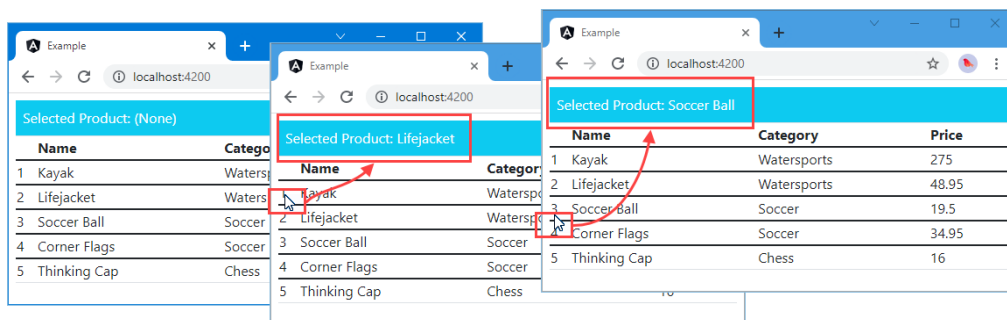


Figure 13.2. Using an event binding

This is a simple example, but it shows the structure of an event binding, which is illustrated in figure 13.3.

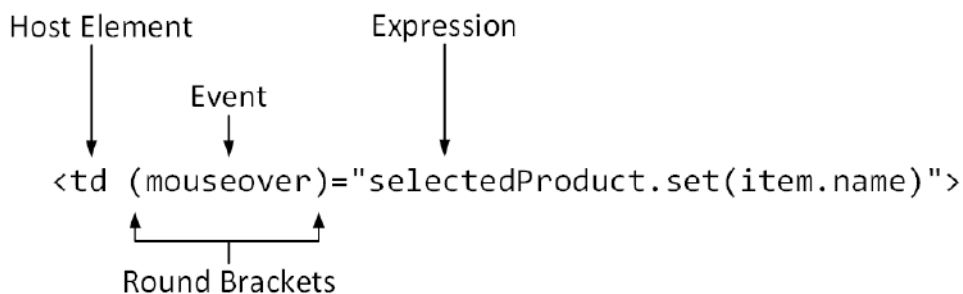


Figure 13.3. The anatomy of an event binding

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 13.3, 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 `=`, or invoke methods, such as the writable signal `set` method.

The expression for the binding updates the `selectedProduct` signal using the value of the `item.name` property. The `selectedProduct` signal 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` signal is changed by the event binding.

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](https://developer.mozilla.org/en-US/docs/Web/Events) 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 13.4.

Listing 13.4. Enhancing the component in the `component.ts` file in the `src/app` folder

```
import { Component, computed, signal } from "@angular/core";
import { Model } from "../repository.model";
import { Product } from "../product.model";

@Component({
  selector: "app",
  templateUrl: "template.html"
})
export class ProductComponent {
  private model: Model = new Model();
```

```

products = computed<Product[]>(() => this.model.Products());

count = computed<number>(() => this.products().length);

product(key: number): Product | undefined {
    return this.model.getProduct(key);
}

selectedProduct = signal<string | undefined>(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 value of the `selectedProduct` signal. In listing 13.5, 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 13.5. 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)'}}
  </div>
  <table class="table table-sm table-bordered">
    <tr *ngFor="let item of products(); let i = index"
      [class.bg-info]="getSelected(item)">
      <td (mouseover)="selectedProduct.set(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` signal value matches name of the `Product` object used to create them, which is changed by the event binding when the `mouseover` event is triggered, as shown in figure 13.4.

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 respond dynamically to changes in the application state, creating content generated and managed entirely within the browser.

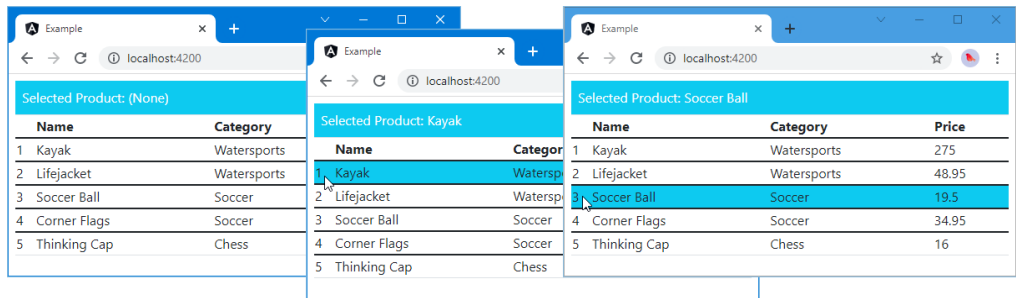


Figure 13.4. Highlighting table rows through an event binding

13.2.1 Using event data

The previous example used the event binding to connect two pieces of data provided by the component: when the `mouseenter` event is triggered, the binding's expression sets the `selectedProduct` signal 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 13.6 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 13.6. 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 products(); let i = index"
      [class.bg-info]="getSelected(item)">
      <td (mouseover)="selectedProduct.set(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.set($any($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 13.3.

Table 13.3. The properties common to all DOM event objects

Name	Description
type	This property returns a <code>string</code> that identifies the type of event that has been triggered.
target	This property returns the <code>object</code> that triggered the event, which will generally be the object that represents the HTML element in the DOM.
timeStamp	This property returns a <code>number</code> 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 13.6 uses this variable to access the event and its `target` property, like this:

```
...
<input class="form-control"
      (input)="selectedProduct.set($any($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 `$event` variable. The `InputEvent.target` property returns an `HTMLInputElement` object, which is how the DOM represents the `input` element that triggered the event. The `HTMLInputElement.value` property returns the content of the `input` element. Putting these types together means reading the value of `$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.set($any($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` signal 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 value of the `selectedProduct` signal matches the name of the product they represent. And, now that the value of the `selectedProduct` signal 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 13.5.

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 changes in the data model.

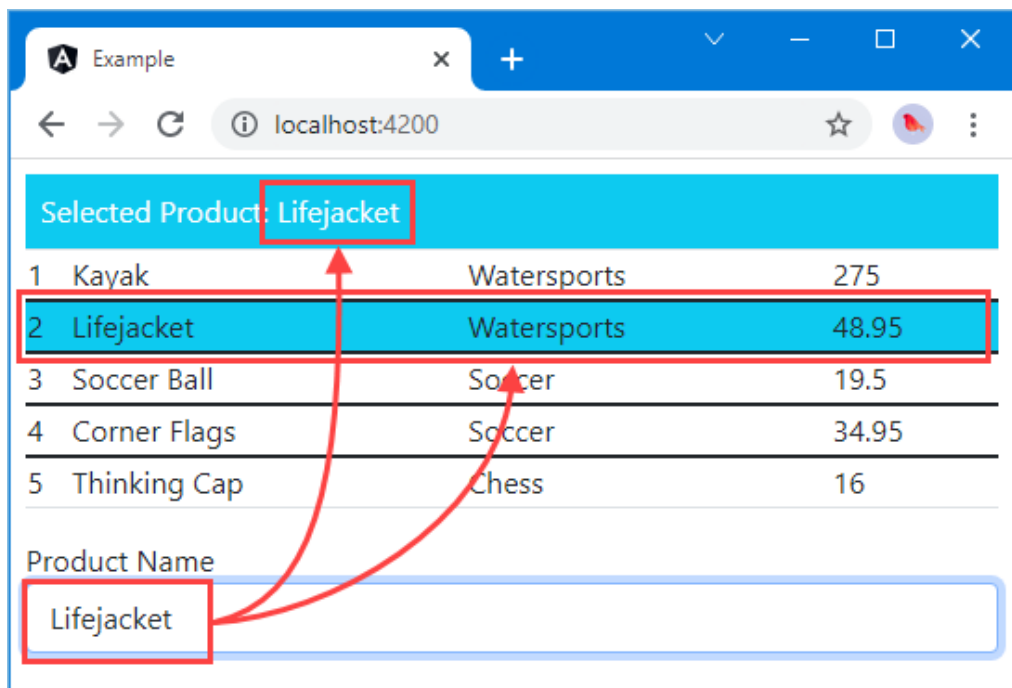


Figure 13.5. Using event data

13.2.2 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 13.7, which provides a way to handle events without needing to use the any type.

Listing 13.7. Defining a method in the component.ts file in the src/app folder

```
import { Component, computed, signal } from "@angular/core";
import { Model } from "../repository.model";
import { Product } from "../product.model";

@Component({
  selector: "app",
  templateUrl: "template.html"
})
export class ProductComponent {
  private model: Model = new Model();

  products = computed<Product[]>(() => this.model.Products());

  count = computed<number>(() => this.products().length);

  product(key: number): Product | undefined {
    return this.model.getProduct(key);
  }

  selectedProduct = signal<string | undefined>(undefined);

  getSelected(product: Product): boolean {
    return product.name == this.selectedProduct();
  }

  handleInputEvent(ev: Event) {
    if (ev.target instanceof HTMLInputElement) {
      this.selectedProduct.set(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 used to set the `selectedProduct` signal. Listing 13.8 updates the template to use the new method to handle events.

Listing 13.8. Handling an event 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 products(); let i = index"
      [class.bg-info]="getSelected(item)">
      <td (mouseover)="selectedProduct.set(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.

13.2.3 Using template reference variables

In chapter 12, 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 13.9.

Listing 13.9. 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 products(); 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.set(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.set(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.

13.3 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 13.10.

Listing 13.10. 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 products(); let i = index"
        [class.bg-info]="getSelected(item)">
      <td (mouseover)="selectedProduct.set(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.set($any($event).target.value)"
          [value]="selectedProduct() ?? ''" />
  </div>
  <div class="form-group">
    <label>Product Name</label>
    <input class="form-control"
          (input)="selectedProduct.set($any($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` signal. The property binding ties the value of the `selectedProduct` signal 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` signal, 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 13.6.

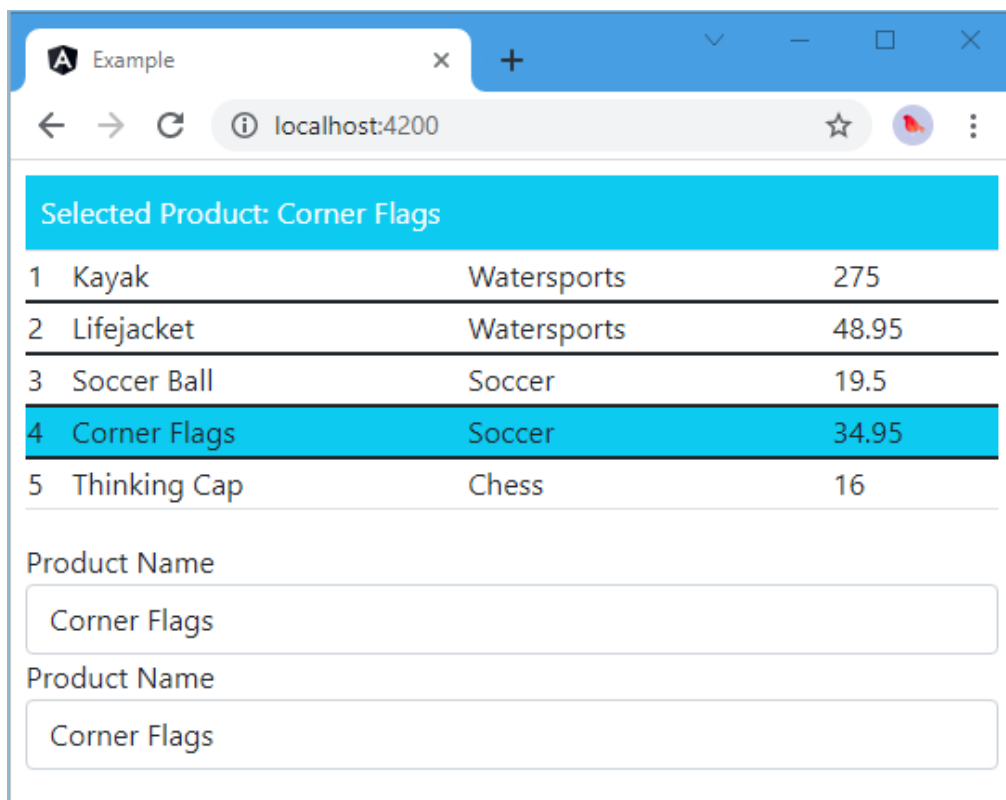


Figure 13.6. Creating a two-way data binding

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 Lifjacket, 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` signal will cause the `input` elements to display the product name.

13.3.1 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. In Angular 16, this directive doesn't work with signals, which have yet to be integrated into every Angular feature. The `ngModel` directive can be used with regular class properties or access signals via a getter and setter, which is the approach I have taken in listing 13.11.

Importing the forms module

The features demonstrated in this chapter rely on the Angular forms module, which I added to the example application in chapter 10 to demonstrate change detection and signals. As a reminder, here are the contents of the `app.module.ts` file with the required statements emphasized:

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

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

import { ProductComponent } from './component';

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

Adding `FormsModule` to the list of dependencies enables the form features and makes them available for use throughout the application.

Listing 13.11. Accessing a signal in the `component.ts` file in the `src/app` folder

```
import { Component, computed, signal } from '@angular/core';
import { Model } from '../repository.model';
import { Product } from '../product.model';

@Component({
  selector: 'app',
  templateUrl: 'template.html'
```

```

    })
    export class ProductComponent {
        private model: Model = new Model();

        products = computed<Product[]>(() => this.model.Products());

        count = computed<number>(() => this.products().length);

        product(key: number): Product | undefined {
            return this.model.getProduct(key);
        }

        selectedProduct = signal<string | undefined>(undefined);

        get selectedProductProp() { return this.selectedProduct(); }
        set selectedProductProp(val) { this.selectedProduct.set(val); }

        getSelected(product: Product): boolean {
            return product.name == this.selectedProduct();
        }

        handleInputEvent(ev: Event) {
            if (ev.target instanceof HTMLInputElement) {
                this.selectedProduct.set(ev.target.value);
            }
        }
    }
}

```

This awkwardness reflects the Angular transition to signals and is only practical for small numbers of bindings. Listing 13.12 shows how to replace the separate bindings with the `ngModel` directive.

Listing 13.12. 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 products(); let i = index"
            [class.bg-info]="getSelected(item)">
            <td (mouseover)="selectedProduct.set(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)]="selectedProductProp" />
    </div>
    <div class="form-group">
        <label>Product Name</label>
        <input class="form-control" [(ngModel)]="selectedProductProp" />
    </div>

```

```
</div>
```

Using the `ngModel` directive requires combining the syntax of the property and event bindings, as illustrated in figure 13.7. 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: `[()]`. 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.

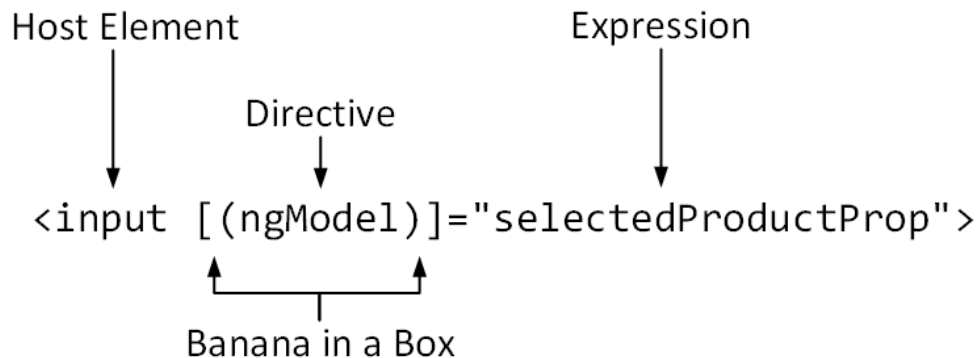


Figure 13.7. The anatomy of a two-way data binding

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 `selectedProductProp` property, which will update the value of the `selectedProduct` signal. Equally, when the value of the `selectedProduct` signal changes, it will be used to update the contents of the element, via the `selectedProductProp` getter.

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.

13.4 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.

13.4.1 Adding a form to the example application

Listing 13.13 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 13.13. Enhancing the component in the `component.ts` file in the `src/app` folder

```
import { Component, computed, signal } from "@angular/core";
import { Model } from "../repository.model";
import { Product } from "../product.model";

@Component({
  selector: "app",
  templateUrl: "template.html"
})
export class ProductComponent {
  private model: Model = new Model();

  products = computed<Product[]>(() => this.model.Products());

  count = computed<number>(() => this.products().length);

  product(key: number): Product | undefined {
    return this.model.getProduct(key);
  }

  // selectedProduct = signal<string | undefined>(undefined);

  // get selectedProductProp() { return this.selectedProduct(); }
  // set selectedProductProp(val) { this.selectedProduct.set(val); }

  // getSelected(product: Product): boolean {
  //   return product.name == this.selectedProduct();
  // }

  // handleInputEvent(ev: Event) {
  //   if (ev.target instanceof HTMLInputElement) {
  //     this.selectedProduct.set(ev.target.value);
  //   }
  // }

  newProduct: Product = new Product();

  get jsonProduct() {
    return JSON.stringify(this.newProduct);
  }
}
```

```

    addProduct(p: Product) {
      console.log("New Product: " + this.jsonProduct);
    }
  }
}

```

I have returned to using the conventional Angular change detection approach. Signals can be used with two-way bindings, as the previous example demonstrated, but this is limited to simple values and doesn't extend to updating properties of a data object, which is how forms are usually managed in Angular applications. As noted, this will change in future releases when signals are fully integrated into the Angular framework.

The listing adds a new `Product` 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 12, 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 13.14, the template content has been replaced with a series of `input` elements for each of the properties defined by the `Product` class.

Listing 13.14. 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 13.8, 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.

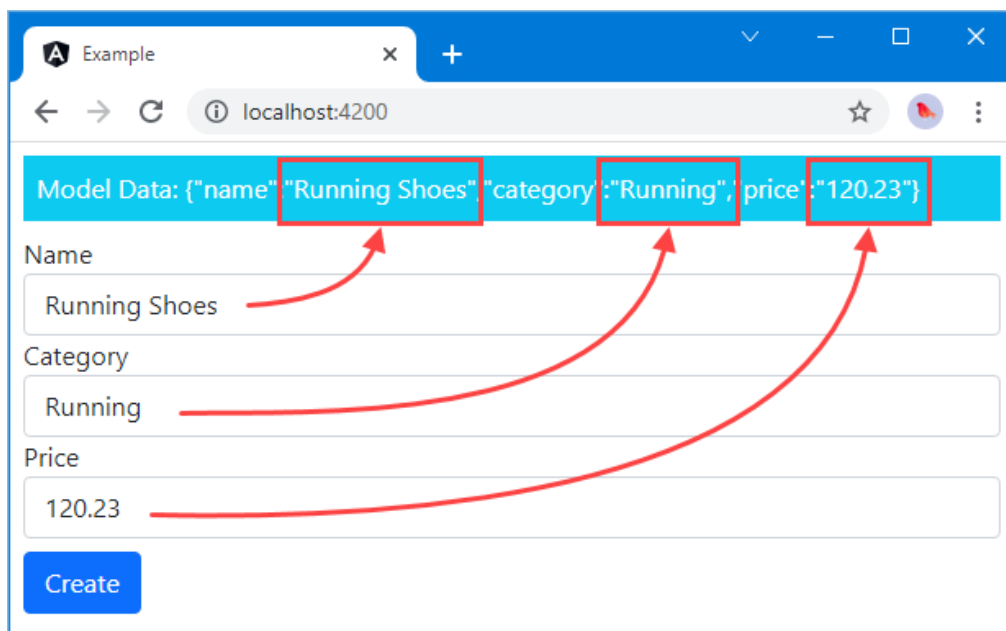


Figure 13.8. Using the form elements to create a new object in the data model

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"}
```

13.4.2 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. Table 13.4 lists the attributes that you can add to `input` elements, each of which defines a validation rule.

Table 13.4. The built-in Angular validation attributes

Attribute	Description
email	This attribute is used to specify a well-formatted email address.
required	This attribute is used to specify a value that must be provided.
minlength	This attribute is used to specify a minimum number of characters.
maxlength	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.
min	This attribute is used to specify a minimum value.
max	This attribute is used to specify a maximum value.
pattern	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 13.15 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 13.15. 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 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 13.15 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 13.5. 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 13.5. The Angular form validation classes

Name	Description
ng-untouched ng-touched	An element is assigned to the <code>ng-untouched</code> 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 <code>ng-touched</code> class.
ng-pristine ng-dirty	An element is assigned to the <code>ng-pristine</code> class if its contents have not been changed by the user and to the <code>ng-dirty</code> class otherwise. Once the contents have been edited, an element remains in the <code>ng-dirty</code> class, even if the user then returns to the previous contents.
ng-valid ng-invalid	An element is assigned to the <code>ng-valid</code> class if its contents meet the criteria defined by the validation rules that have been applied to it and to the <code>ng-invalid</code> class otherwise.
ng-pending	Elements are assigned to the <code>ng-pending</code> class when their contents are being validated asynchronously. See part 3 for details.

These classes can be used to style form elements to provide the user with validation feedback. Add the styles shown in listing 13.16 to the `styles.css` file in the `src` folder.

Listing 13.16. 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 13.9.

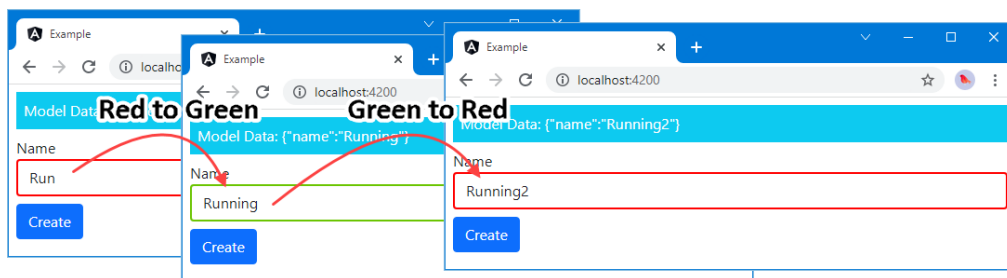


Figure 13.9. Providing validation feedback

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 13.9, 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 13.17 adds validation messages for each of the attributes applied to the `input` element using the support provided by the `ngModel` directive.

Listing 13.17. 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 the chapters 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 13.6. All of the properties described in the table are nullable.

Table 13.6. 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.

errors	This property returns a <code>ValidationErrors</code> object whose properties correspond to each attribute for which there is a validation error.
value	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 13.17 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 13.6, 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 13.7 describes the `error` properties provided for each attribute.

Table 13.7. The Angular form validation error description properties

Name	Description
email	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.
required	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.
minlength.requiredLength	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 13.10.

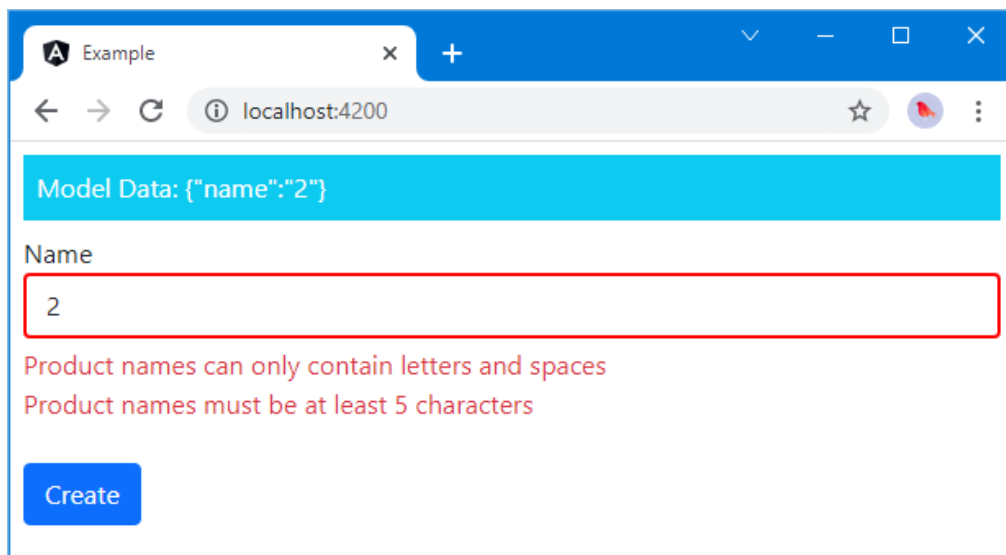


Figure 13.10. Displaying validation messages

USING THE COMPONENT TO DISPLAY VALIDATION MESSAGES

Defining 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 13.18 shows the addition of a component method that accepts the validation state for an `input` element and produces an array of validation messages.

Listing 13.18. Generating messages in the `component.ts` file in the `src/app` folder

```
import { Component, computed, signal } 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 {
  private model: Model = new Model();

  products = computed<Product[]>(() => this.model.Products());

  count = computed<number>(() => this.products().length);

  product(key: number): Product | undefined {
    return this.model.getProduct(key);
  }
}
```

```

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 13.6 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 13.19 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 13.19. 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="^[A-Za-z ]+$" />
      <ul class="text-danger list-unstyled mt-1"
        *ngIf="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>
```

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.

13.4.3 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 13.20 adds two new members to the component.

Listing 13.20. Enhancing the component in the component.ts file in the src/app folder

```
import { Component, computed, signal } 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 {
        private model: Model = new Model();

        // ...statements 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 13.21 shows the changes required to the template to take advantage of these new features and implement form-wide validation.

Listing 13.21. Performing 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 14. 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 13.21 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 13.11.

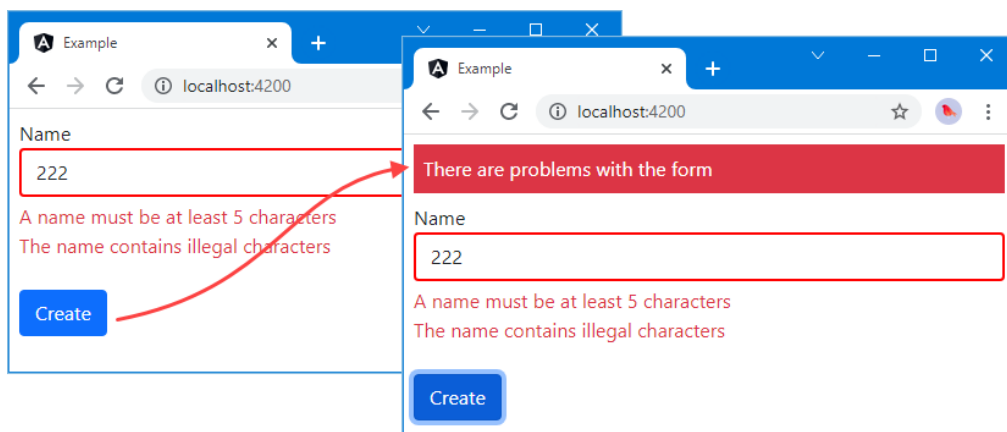


Figure 13.11. Displaying a validation summary message

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 13.22, 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 13.22. Validation messages in the component.ts file in the src/app folder

```
import { Component, computed, signal } 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 {
  private model: Model = new Model();

  // ...statements 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 13.23, 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 13.23. 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 13.12.

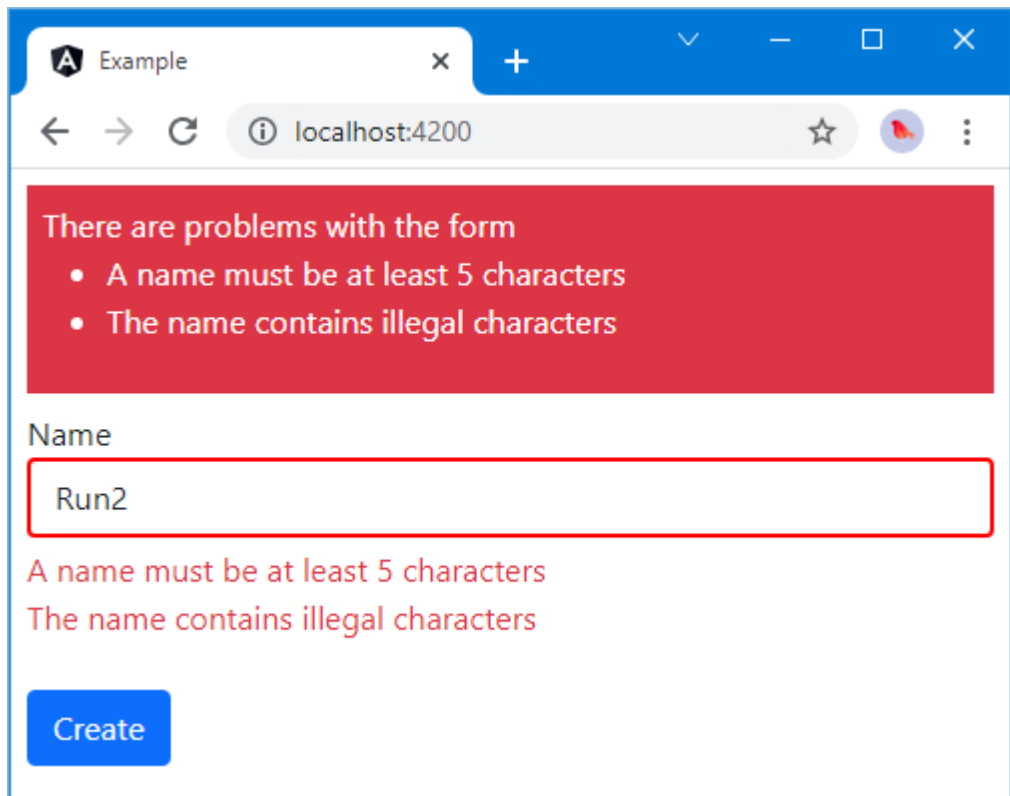


Figure 13.12. Displaying an overall validation summary

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 13.24, I used the property binding on the `button` element.

Listing 13.24. 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 `btn-secondary` class to the `button` element when the form has been submitted and has invalid data. This class applies a Bootstrap CSS style, as shown in figure 13.13.

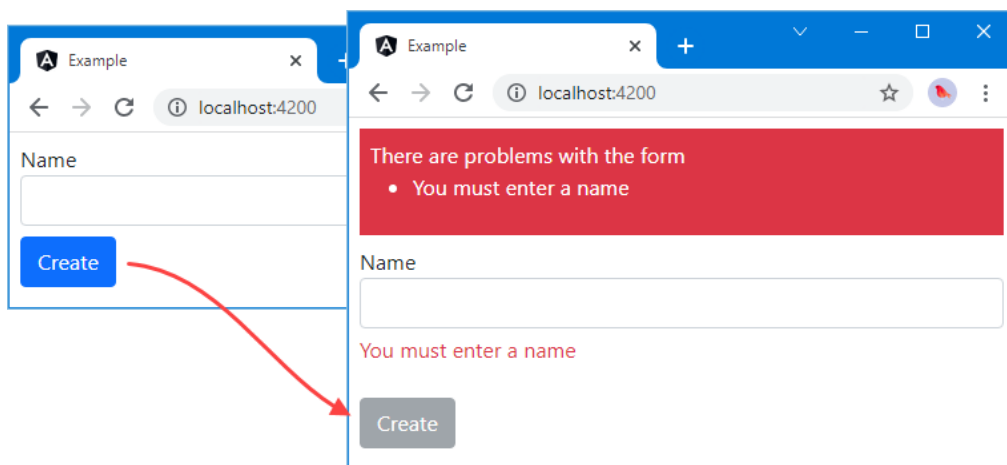


Figure 13.13. Disabling the submit button

13.4.4 Completing the form

Now that the validation features are done, I can complete the form. Listing 13.25 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 13.25. 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="^[A-Za-z ]+$" />
    </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 13.26.

Listing 13.26. 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 13.5, 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 13.14. As you address each validation error, the input elements will turn green, and you will be able to submit the form when no validation errors remain.

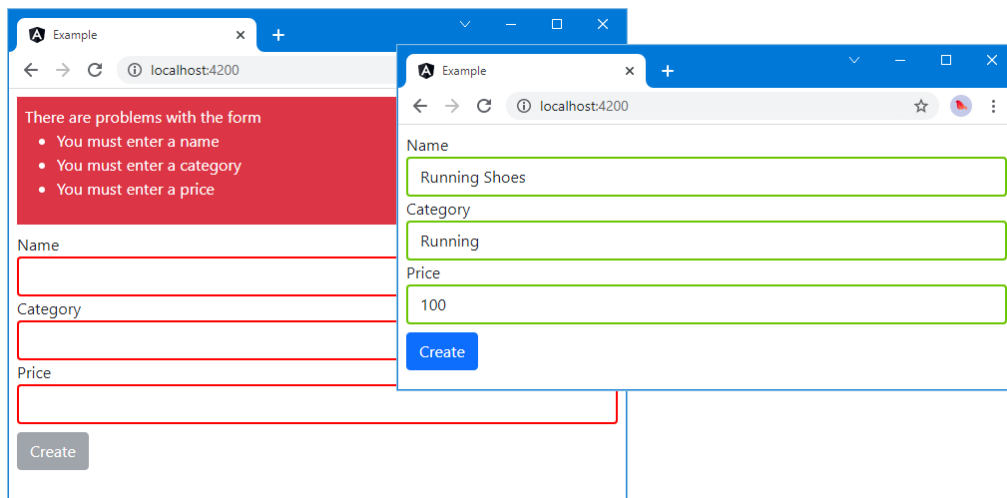


Figure 13.14. Finishing the form

13.5 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.

- Event bindings are used to handle events by evaluating expressions.
- Event binding expressions can invoke component methods or use variables with the template.
- Two-way bindings synchronize the contents of an HTML element with a component property.

- The content entered into form elements can be validated with rules defined in the template or the component.

In the next chapter, I explain how to create custom directives.

14

Creating attribute directives

This chapter covers

- Creating custom directives that modify a single HTML element
- Configuring and applying custom directives
- Receiving data in a custom directive
- Using the directive lifecycle API
- Generating custom events from directives
- Creating directives that support template variables

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 14.1 puts attribute directives into context.

Table 14.1. 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 14.2 summarizes the chapter.

Table 14.2. Chapter summary

Problem	Solution	Listing
Creating an attribute directive	Apply <code>@Directive</code> to a class	1–5
Accessing host element attribute values	Apply the <code>@Attribute</code> decorator to a constructor parameter	6–9
Creating a data-bound input property	Apply the <code>@Input</code> decorator to a class property	10–11, 13, 14
Receiving a notification when a data-bound input property value changes	Implement the <code>ngOnChanges</code> method	12
Defining an event	Apply the <code>@Output</code> decorator	15, 16
Creating a property binding or event binding on the host element	Apply the <code>@HostBinding</code> or <code>@HostListener</code> decorator	17–21
Exporting a directive's functionality for use in the template	Use the <code>exportAs</code> property of the <code>@Directive</code> decorator	22, 23

14.1 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 13, as shown in listing 14.1.

TIP You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/manningbooks/pro-angular-16>. See chapter 1 for how to get help if you have problems running the examples.

Listing 14.1. Replacing the contents of the `template.html` file in the `src/app` folder

```
<div class="container-fluid">
  <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 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 *ngFor="let item of products(); let i = index">
            <td>{{i + 1}}</td>
            <td>{{item.name}}</td>
            <td>{{item.category}}</td>
            <td>{{item.price}}</td>
          </tr>
        </tbody>
      </table>
    </div>
  </div>
</div>
```

```

        </table>
      </div>
    </div>
  </div>

```

This listing uses the Bootstrap grid layout to position the form and the table side by side. Listing 14.2 simplifies the component and updates the component's `addProduct` method so that it adds a new object to the data model.

Listing 14.2. Replacing the contents of the `component.ts` file in the `src/app` folder

```

import { Component, computed } from "@angular/core";
import { Model } from "../repository.model";
import { Product } from "../product.model";

@Component({
  selector: "app",
  templateUrl: "template.html"
})
export class ProductComponent {
  private model: Model = new Model();

  products = computed<Product[]>(() => this.model.Products());

  count = computed<number>(() => this.products().length);

  product(key: number): Product | undefined {
    return this.model.getProduct(key);
  }

  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 14.1. 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.

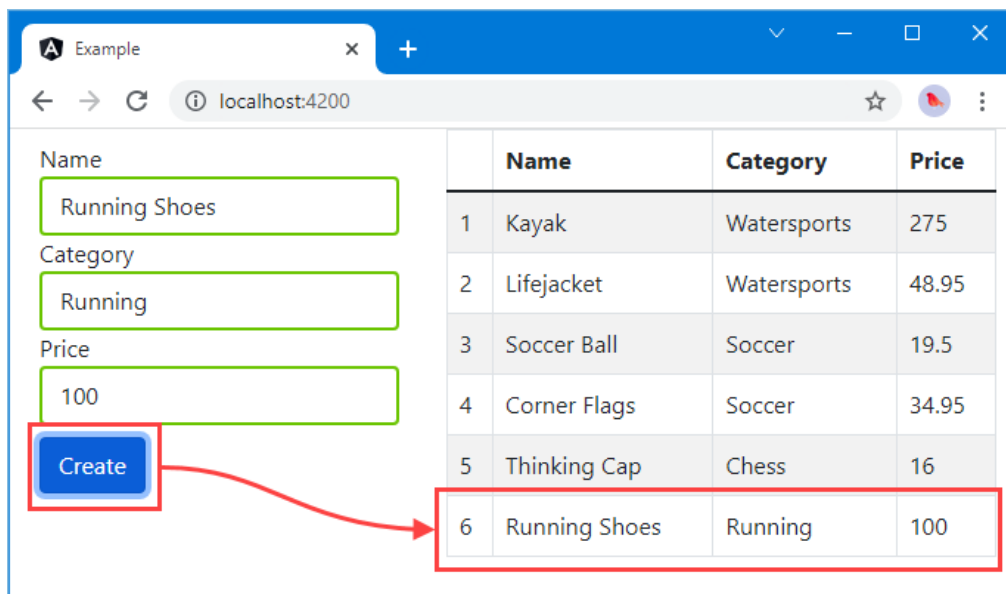


Figure 14.1. Running the example application

14.2 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 14.3. 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 14.3. 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 Document 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.

14.2.1 Applying a custom directive

There are two steps to applying 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 14.4.

Listing 14.4. Adding a Directive Attribute in the `template.html` File in the `src/app` Folder

```
...
<tbody>
  <tr *ngFor="let item of products(); let i = index" pa-attr>
    <td>{{i + 1}}</td>
    <td>{{item.name}}</td>
    <td>{{item.category}}</td>
    <td>{{item.price}}</td>
  </tr>
</tbody>
...
```

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 14.5.

Listing 14.5. 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 { FormsModule } from '@angular/forms';

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

import { ProductComponent } from './component';
import { PaAttrDirective } from './attr.directive';

@NgModule({
  declarations: [
    ProductComponent, PaAttrDirective
  ],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule
  ],
  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 16.

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 14.2. (You may have to restart the Angular development tools to see the change.)

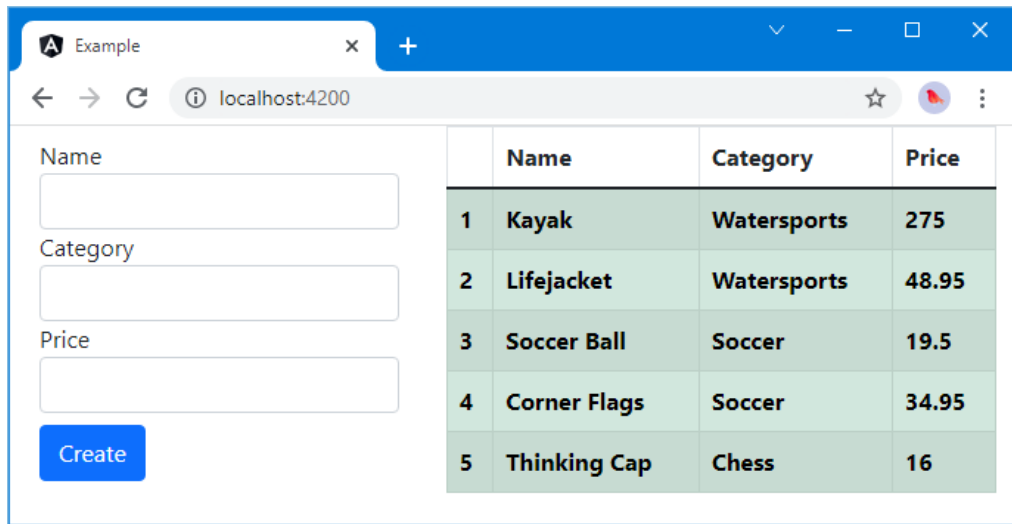


Figure 14.2. Applying a custom directive

14.3 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.

14.3.1 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 14.6 applies the directive to some of the `td` elements in the template table and adds an attribute that specifies the class to which the host element should be added. 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 14.6. Adding attributes in the `template.html` file in the `src/app` folder

```
...
<tbody>
  <tr *ngFor="let item of products(); 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 14.7 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 14.7. 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 14.7. 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 14.3.

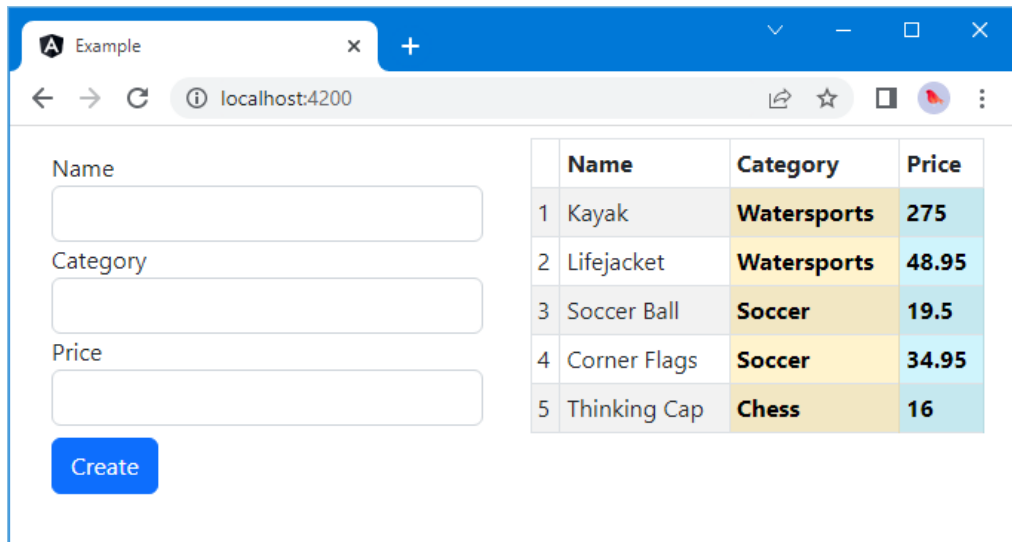


Figure 14.3. Configuring a directive using a host element attribute

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 14.8.

Listing 14.8. 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 14.9, I have updated the template to reflect the dual-purpose attribute.

Listing 14.9. Applying a directive in the `template.html` file in the `src/app` folder

```
...
<tbody>
  <tr *ngFor="let item of products(); let i = index">
    <td>{{i + 1}}</td>
```



```

        <td>{{item.name}}</td>
        <td pa-attr="table-warning">{{item.category}}</td>
        <td pa-attr="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.

14.3.2 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 14.10 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 14.10. Using expressions in the template.html file in the src/app folder

```

...
<tbody>
  <tr *ngFor="let item of products(); let i = index"
      [pa-attr]="products().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 products(); let i = index"
    [pa-attr]="products().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 12 and 13.

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 14.11.

Listing 14.11. 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. (The `@Input` decorator can also be configured with an object, as described in the "Requiring input property values" section).

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 14.3.

Table 14.3. The directive lifecycle hook methods

Name	Description
<code>ngOnInit</code>	This method is called after Angular has set the initial value for all the input properties that the directive has declared.
<code>ngOnChanges</code>	This method is called when the value of an input property has changed and also just before the <code>ngOnInit</code> method is called.
<code>ngDoCheck</code>	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.
<code>ngAfterContentInit</code>	This method is called when the directive's content has been initialized. See chapter 15 for an example that uses this method.
<code>ngAfterContentChecked</code>	This method is called after the directive's content has been inspected as part of the change detection process.
<code>ngOnDestroy</code>	This method is called immediately before Angular destroys a directive.

To set the class on the host element, the directive in listing 14.11 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 14.4.

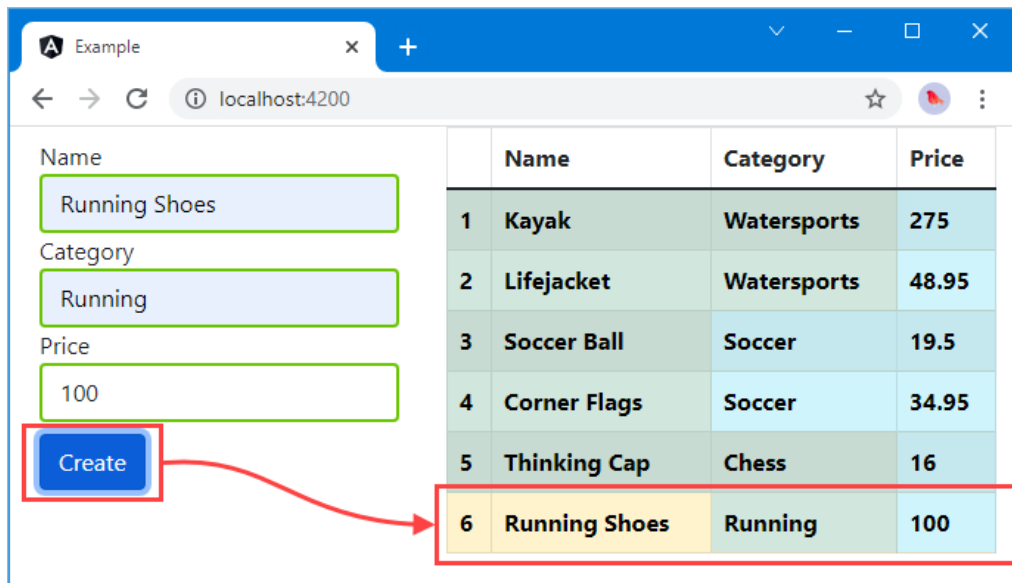


Figure 14.4. Using an input property in a custom directive

14.3.3 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 14.12.

Listing 14.12. 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.element.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 14.4.

Table 14.4. 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 <code>true</code> 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 14.5.

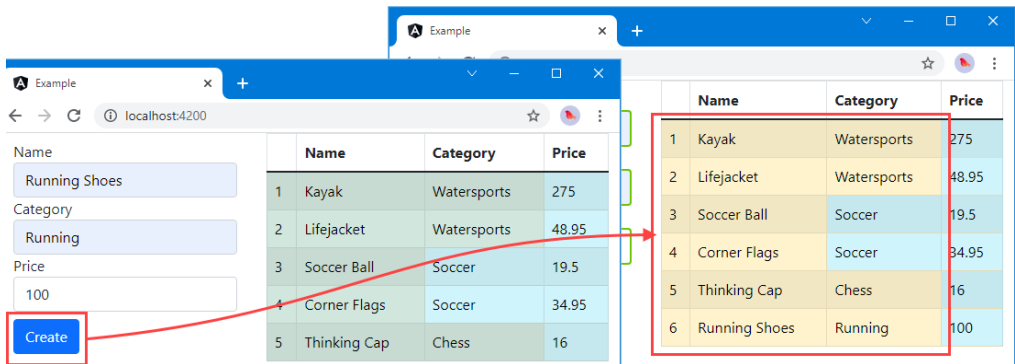


Figure 14.5. Responding to input property changes

14.3.4 Requiring input property values

By default, input properties are optional and can be omitted when a directive is applied in a template. This means that the directive has to be able to work when no value is supplied, typically by providing a fallback or default value.

An alternative approach is to configure input properties so that a value is required, as shown in listing 14.13.

Listing 14.13. Adding an input property 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 = "";

  @Input({ required: true, alias: "fg-class" })
  fgClass: string | undefined;

  ngOnChanges(changes: SimpleChanges) {
    ["bgClass", "fgClass"].forEach(c => {
      let change = changes[c];
      if (change) {
        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 `@Input` decorator is configured with an object containing that defines the properties described in table 14.5.

Table 14.5. The `@Input` decorator configuration properties

Name	Description
<code>alias</code>	This property is used to configure the name of the attribute associated with the property.
<code>required</code>	A value must be provided when this value of this property is <code>true</code> .

The configuration object in listing 14.13 uses the `alias` property to set the attribute name to `fg-class`, and sets the `required` property to `true`, which tells Angular that the directive can only be used when there is a value defined.

The Angular build tools only check for required input properties when processing templates, so you won't see an error until you also save the `template.html` file (although you may see errors displayed in your code editor). When the template is saved, the Angular build tools will produce the following error for each application of the directive:

```
Error: src/app/template.html:46:9 - error NG8008: Required input 'fg-class'
      from directive PaAttrDirective must be specified.
```

To correct the errors, each application of the directive requires a value for the input property, as shown in listing 14.14.

Listing 14.14. Adding property values in the `template.html` file in the `src/app` folder

```

...
<tbody>
  <tr *ngFor="let item of products(); let i = index"
      [pa-attr]="products().length < 6
        ? 'table-success' : 'table-warning'"
      [fg-class]="'text-dark'">
    <td>{{i + 1}}</td>
    <td>{{item.name}}</td>
    <td [pa-attr]="item.category == 'Soccer' ? 'table-info' : null"
      [fg-class]="'text-danger'">
      {{item.category}}
    </td>
    <td [pa-attr]="'table-info'" [fg-class]="'text-warning'">
      {{item.price}}
    </td>
  </tr>
</tbody>
...

```

Save the changes and you will see that the Bootstrap CSS classes assigned to the `fg-class` attribute are applied to the text in the table, as shown in figure 14.6.

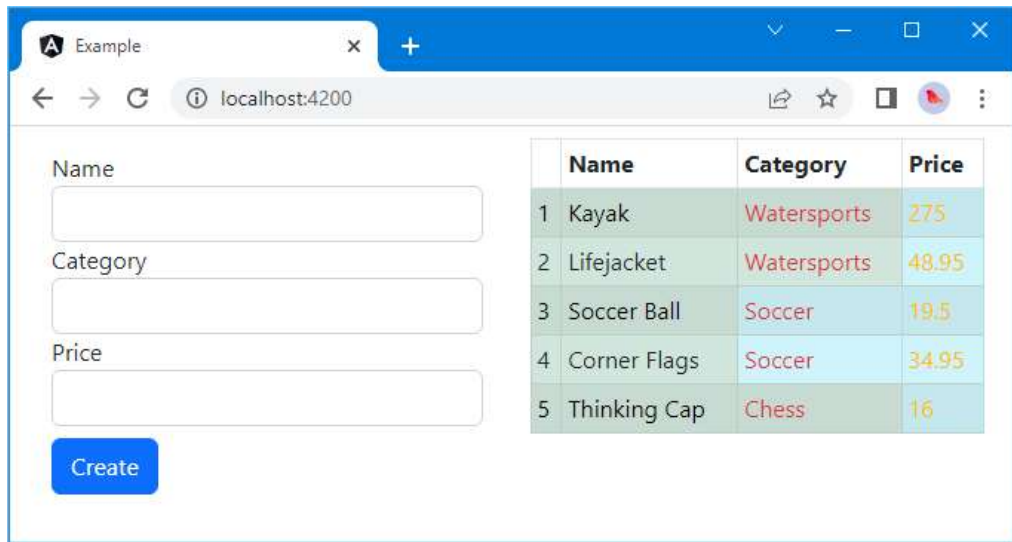


Figure 14.6. Using a required input property

14.4 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 14.15.

Listing 14.15. 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({ required: true, alias: "fg-class"})
fgClass: string | undefined;

@Input("pa-product")
product: Product = new Product();

@Output("pa-category")
click = new EventEmitter<string>();

ngOnChanges(changes: SimpleChanges) {
  ["bgClass", "fgClass"].forEach(c => {
    let change = changes[c];
    if (change) {
      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 14.6 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 14.6. The `EventEmitter` method

Name	Description
<code>emit (value)</code>	This method triggers the custom event associated with the <code>EventEmitter</code> , 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.

14.4.1 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 13. Listing 14.16 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 14.16. Binding to a custom event in the `template.html` file in the `src/app` folder

```
...
<tbody>
  <tr *ngFor="let item of products(); let i = index"
      [pa-attr]="products().length < 6
        ? 'table-success' : 'table-warning'"
      [pa-product]="item"
      (pa-category)="newProduct.category = $event"
      [fg-class]="'text-dark'">
    <td>{{i + 1}}</td>
    <td>{{item.name}}</td>
  </tr>
</tbody>
```

```

<td [pa-attr]="item.category == 'Soccer' ? 'table-info' : null"
    [fg-class]="'text-danger'">
    {{item.category}}
</td>
<td [pa-attr]="'table-info'" [fg-class]="'text-warning'">
    {{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 14.7.

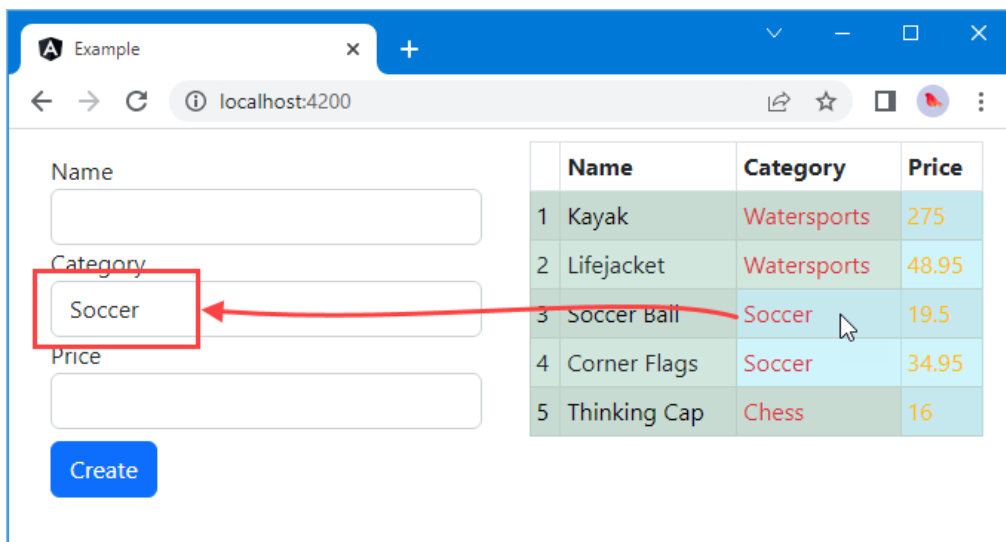


Figure 14.7. Defining and receiving a custom event using an output property

14.5 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 using 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 14.17.

Listing 14.17. 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({ required: true, alias: "fg-class" })
  fgClass: string | undefined;

  @Input("pa-product")
  product: Product = new Product();

  @Output("pa-category")
  click = new EventEmitter<string>();

  ngOnChanges(changes: SimpleChanges) {
    //["fgClass"].forEach(c => {
    let change = changes["fgClass"];
    if (change) {
      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.

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.

14.6 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 14.18 adds some new elements and bindings to the `template.html` file.

Listing 14.18. 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>
</div>

```

```

</div>

<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 *ngFor="let item of products(); let i = index"
      [pa-attr]="products().length < 6
        ? 'table-success' : 'table-warning'"
      [pa-product]="item"
      (pa-category)="newProduct.category = $event"
      [fg-class]="'text-dark'">
      <td>{{i + 1}}</td>
      <td>{{item.name}}</td>
      <td [pa-attr]="item.category == 'Soccer'
        ? 'table-info' : null"
        [fg-class]="'text-danger'">
        {{item.category}}
      </td>
      <td [pa-attr]="'table-info'"
        [fg-class]="'text-warning'">
        {{item.price}}
      </td>
    </tr>
  </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 14.19.

Listing 14.19. 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 14.20.

Listing 14.20. 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 { FormsModule } from '@angular/forms';

```

```

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

import { ProductComponent } from './component';
import { PaAttrDirective } from './attr.directive';
import { PaModel } from './twoway.directive';

@NgModule({
  declarations: [
    ProductComponent, PaAttrDirective, PaModel
  ],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule
  ],
  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 14.8.

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.

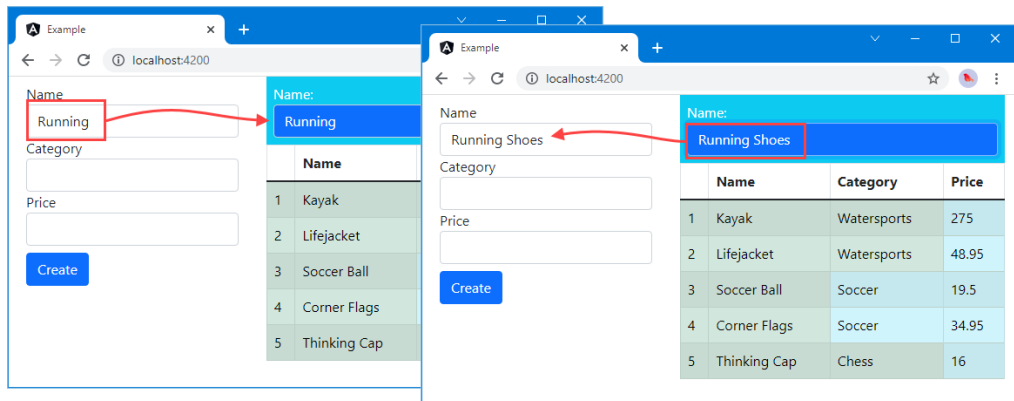


Figure 14.8. Testing the two-way flow of data

The final step is to simplify the bindings and apply the banana-in-a-box style of brackets, as shown in listing 14.21.

Listing 14.21. Simplifying the bindings 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" />
</div>
...
```

When Angular encounters the `[()]` brackets, it expands the binding to match the format used in listing 14.18, 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.

14.7 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 13:

```
...
<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 14.22 modifies the directive from the previous section so that it provides details of whether it has expanded the text in its host element.

Listing 14.22. 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 is 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 14.23 creates a template variable for the directive's exported functionality and uses it in a style binding.

Listing 14.23. Using exported 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 14.9.

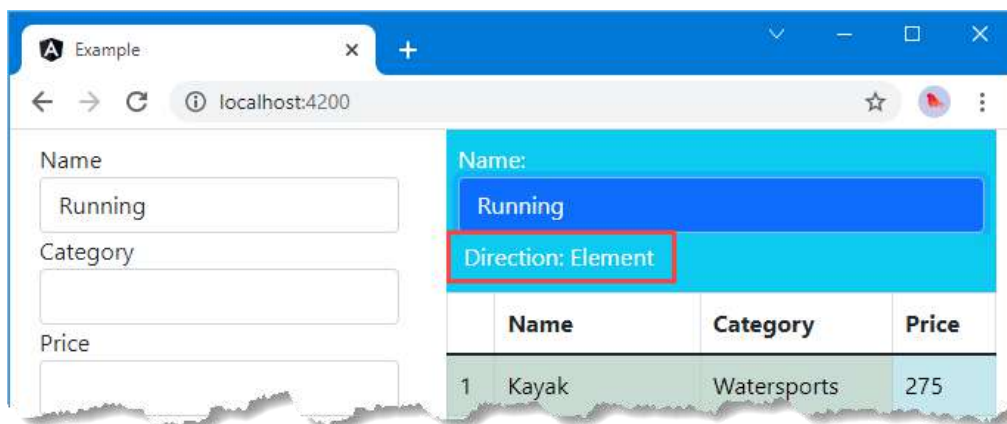


Figure 14.9. Exporting functionality from a directive

14.8 Summary

In this chapter, I described how to define and use attribute directives, including the use of input and output properties and host bindings.

- Attribute directives transform a single HTML element and are applied as attributes.
- Attribute directives are defined as classes and receive details of the element to which they have been applied through the constructor.
- Attribute directives can inspect the element they have been applied to and receive data values through input properties.
- Directives have a well-defined lifecycle, which is expressed through a series of methods that Angular invokes as the state of the application changes.
- Attribute directives can emit custom events to indicate changes that may be useful elsewhere in the application.
- Attribute directives can export data for use in template variables.

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.

15

Creating structural directives

This chapter covers

- Creating and applying structural directives
- Creating directives that iterate over data values
- Understanding the concise directive syntax
- Responding to changes in a structural directive
- Querying child content in a structural directive

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 14, 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 15.1 puts structural directives in context.

Table 15.1. 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 15.2 summarizes the chapter.

Table 15.2. 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	1–7
Creating an iterating structural directive	Define a <code>ForOf</code> input property in a structural directive class and iterate over its value	8–13
Handling data changes in a structural directive	Use a differ to detect changes in the <code>ngDoCheck</code> method	14–20
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	21–27

15.1 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, listing 15.1 removes the `fgClass` input property from the attribute directive created in chapter 14.

Listing 15.1. Removing a property 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({ required: true, alias: "fg-class"})
        // fgClass: string | undefined;

        @Input("pa-product")
        product: Product = new Product();

        @Output("pa-category")
        click = new EventEmitter<string>();

        // ngOnChanges(changes: SimpleChanges) {
        //     let change = changes["fgClass"];
        //     if (change) {
        //         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);
            }
        }
    }
}

```

In listing 15.2, I simplified the template to remove the form, leaving only the table. (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/manningbooks/pro-angular-16>. See chapter 1 for how to get help if you have problems running the examples.

Listing 15.2. Simplifying the template in the template.html file in the src/app folder

```

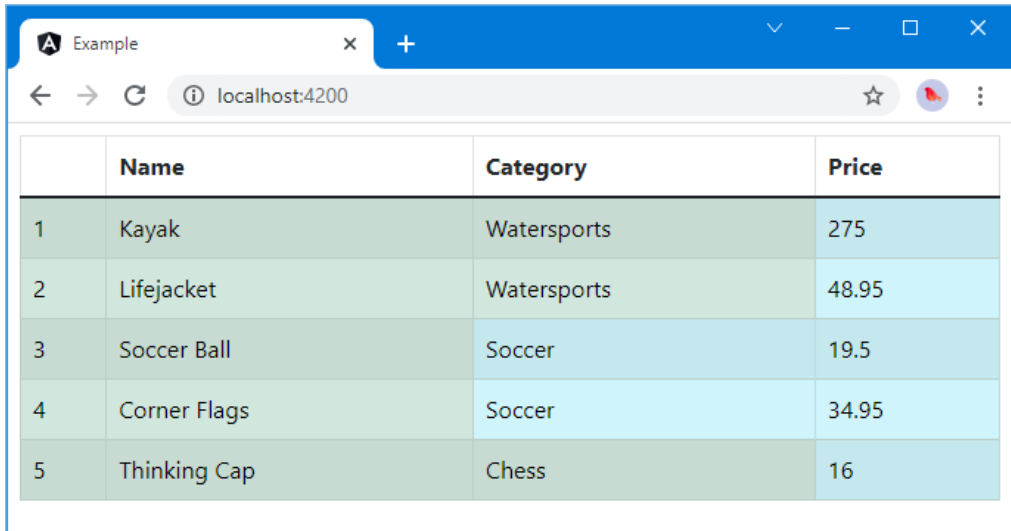
<div class="p-2">
  <table class="table table-bordered table-striped">
    <thead>
      <tr><th></th><th>Name</th><th>Category</th><th>Price</th></tr>
    </thead>
    <tbody>
      <tr *ngFor="let item of products(); let i = index"
        [pa-attr]="count() < 6
          ? 'table-success' : 'table-warning'"
        [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 15.1.



	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 15.1. Running the example application

15.2 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, 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 15.3 shows the template changes.

Listing 15.3. 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 products(); let i = index"
          [pa-attr]="count() < 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 15.4 adds the `showTable` property to the component.

Listing 15.4. Adding a property in the `component.ts` file in the `src/app` folder

```
import { Component, computed } from "@angular/core";
import { Model } from "../repository.model";
import { Product } from "../product.model";

@Component({
  selector: "app",
  templateUrl: "template.html"
})
export class ProductComponent {
  private model: Model = new Model();
  showTable: boolean = false;

  products = computed<Product[]>(() => this.model.Products());

  count = computed<number>(() => this.products().length);

  product(key: number): Product | undefined {
    return this.model.getProduct(key);
  }

  newProduct: Product = new Product();

  addProduct(p: Product) {
    this.model.saveProduct(p);
  }

  submitForm() {
    this.addProduct(this.newProduct);
  }
}
```

15.2.1 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 15.5.

Listing 15.5. 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 at the start of the chapter.

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 15.3.

Table 15.3. 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 15.3 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 14, 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.

15.2.2 Enabling the structural directive

The directive must be enabled in the Angular module before it can be used, as shown in listing 15.6.

Listing 15.6. 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 { FormsModule } from '@angular/forms';

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

import { ProductComponent } from './component';
import { PaAttrDirective } from './attr.directive';
import { PaModel } from './twoway.directive';
import { PaStructureDirective } from './structure.directive';

@NgModule({
  declarations: [
    ProductComponent, PaAttrDirective, PaModel, PaStructureDirective
  ],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule
  ],
  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 15.2. (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.

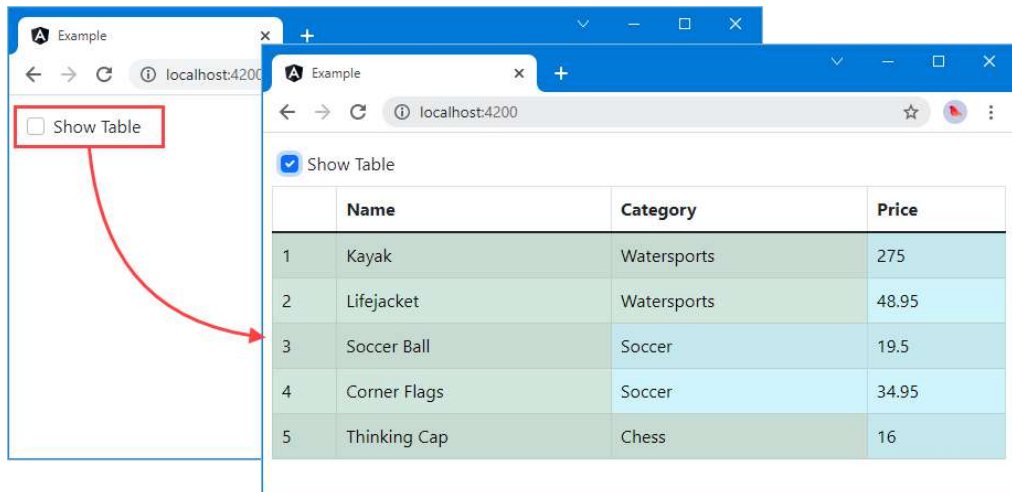


Figure 15.2. Creating a structural directive

15.2.3 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 15.7.

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 15.7. 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 *ngFor="let item of products(); let i = index"
        [pa-attr]="count() < 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.

15.3 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 15.8.

Listing 15.8. Preparing for a new 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>

  <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]="products()" 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]="products()" 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]="products()" 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 objects provided by component's `products` signal 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 15.9.

Listing 15.9. 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 15.10.

Listing 15.10. 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 { FormsModule } from '@angular/forms';

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

import { ProductComponent } from './component';
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
  ],
  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 15.3. 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.)

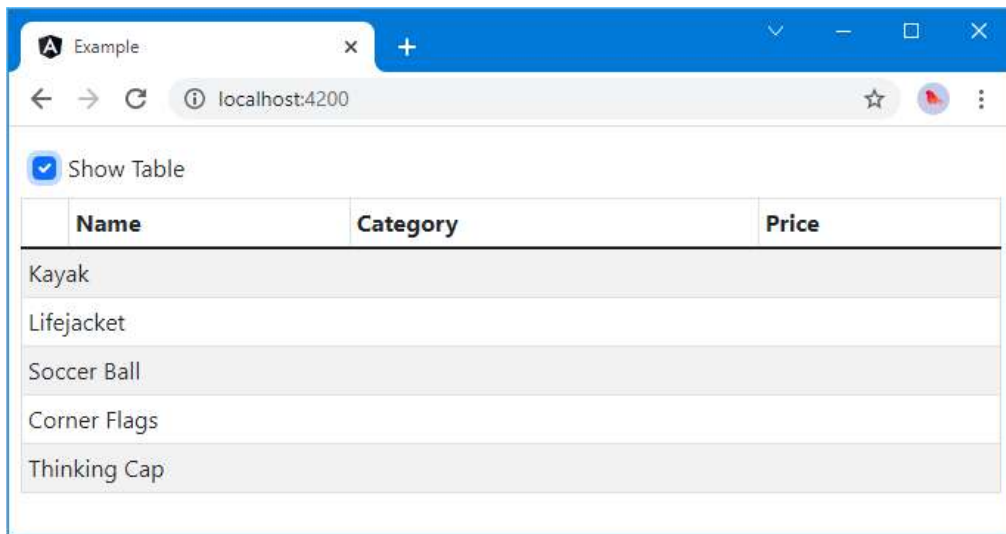


Figure 15.3. Creating an iterating structural directive

15.3.1 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 15.11, I have re-created the same set of values that `ngFor` provides.

Listing 15.11. 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 15.12.

Listing 15.12. Using directive context data in the `template.html` file in the `src/app` folder

```

<div class="p-2">

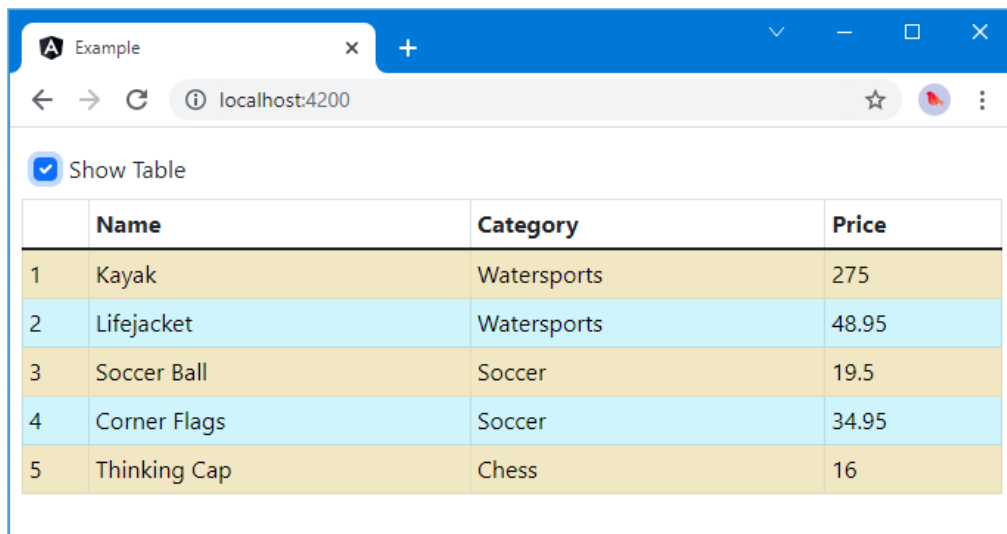
  <div class="form-check m-2">
    <input type="checkbox" class="form-check-input"
      [(ngModel)]="showTable" />
    <label class="form-check-label">Show table</label>
  </div>

  <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]="products()" 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 15.4. The listing also adds table cells to display all the `Product` properties.



	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 15.4. Using directive context data

15.3.2 Using the concise structure syntax

Iterating structural directives support the concise syntax and omit the `ng-template` element, as shown in listing 15.13.

Listing 15.13. 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 products(); 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]="products()" 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 products(); 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.

15.3.3 Dealing with property-level data changes

Two kinds of changes 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 15.14 I have used the Reactive Extensions library to periodically 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 15.14. Modifying objects in the `iterator.directive.ts` file in the `src/app` folder

```

import { Directive, ViewContainerRef, TemplateRef, Input }
  from "@angular/core";
import { interval } from "rxjs";

```

```

@Directive({
  selector: "[paForOf]"
})
export class PaIteratorDirective {

  // ...statements omitted for brevity...

}

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;

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

```

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 15.5.

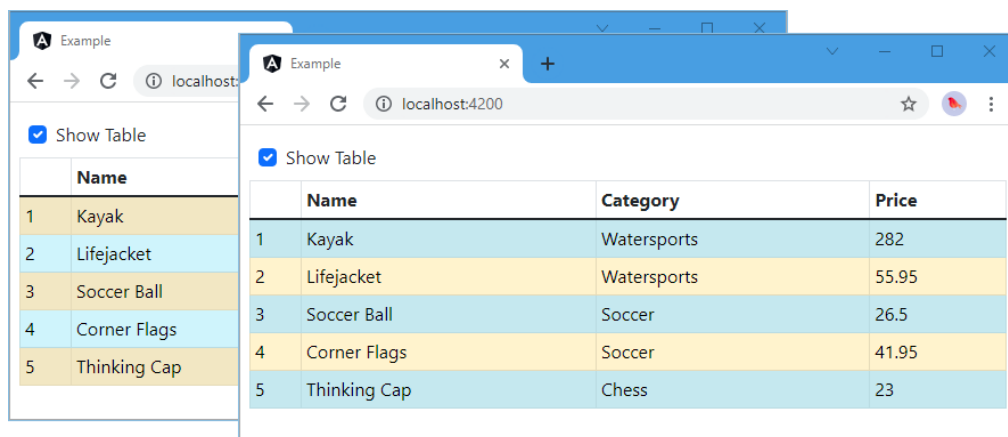


Figure 15.5. Automatic change detection for individual data source objects

15.3.4 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 15.15 implements the `ngDoCheck` method so that the directive updates the content it displays when there is a change.

Listing 15.15. Using `ngDoCheck` in the `iterator.directive.ts` File in the `src/app` folder

```
import { Directive, ViewContainerRef, TemplateRef, Input }
  from "@angular/core";
import { interval } from "rxjs";

@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;

        // interval(2000).subscribe(() => {
        //     this.odd = !this.odd; this.even = !this.even;
        //     this.$implicit.price++;
        // })
    }
}

```

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 statements that periodically change state.

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 15.16. I also removed the checkbox and removed the directive from the table so that it is always displayed.

Listing 15.16. 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 products(); 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>
</div>

```

When you save the changes to the template, the HTML form will be displayed alongside the table of products, as shown in figure 15.6.

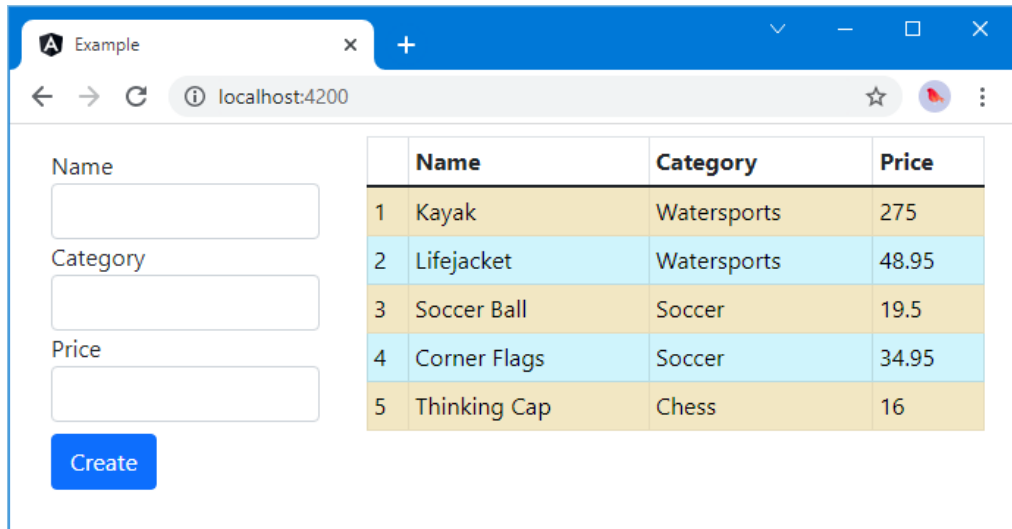


Figure 15.6. Restoring the table in the template

Angular doesn't make any assumptions about how the data relates to the directive's content and so the `ngDoCheck` method is that it is invoked every time Angular detects a change anywhere in the application.

NOTE This behavior may change when signals are fully integrated into Angular, but for the moment, the conventional change detection is applied to directives even when the source of the data is a value read from a signal.

This is a variation of the problem I described in chapter 10 and it requires directives to be written with efficiency in mind. To give a sense of how an inefficient directive behaves, I added a call to the `console.log` method within the directive's `ngDoCheck` method in listing 15.15 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, 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.

MINIMIZING UPDATES

Fortunately, Angular provides some tools for managing updates more efficiently and updating content only when it is required, as shown in listing 15.17. This is not related to the signals feature and is a long-standing capability that allows directives to minimize HTML changes.

Listing 15.17. Minimizing changes in the `iterator.directive.ts` file in the `src/app` folder

```
import { Directive, ViewContainerRef, TemplateRef, Input,
  IterableDiffer, IterableDiffers, IterableChangeRecord }
  from "@angular/core";
import { interval } from "rxjs";

@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 be processed.

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 an `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 15.4 for processing changes.

Table 15.4. 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 15.4 will receive an `IterableChangeRecord` object that describes an item and how it has changed, using the properties shown in table 15.5.

Table 15.5. 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 15.17 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 15.5 to create new views in the view container.

The changes in listing 15.17 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 15.18 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 15.18. Adding a delete method in the component.ts file in the src/app folder

```

import { Component, computed } from "@angular/core";
import { Model } from "../repository.model";
import { Product } from "../product.model";

@Component({
  selector: "app",
  templateUrl: "template.html"
})
export class ProductComponent {
  private model: Model = new Model();
  showTable: boolean = false;

  products = computed<Product[]>(() => this.model.Products());

  count = computed<number>(() => this.products().length);

  product(key: number): Product | undefined {
    return this.model.getProduct(key);
  }

  newProduct: Product = new Product();

  addProduct(p: Product) {
    this.model.saveProduct(p);
  }

  deleteProduct(key: number) {
    this.model.deleteProduct(key);
  }

  submitForm() {
    this.addProduct(this.newProduct);
  }
}

```

Listing 15.19 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 15.19. 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 products(); 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>

```



```

        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 15.7.

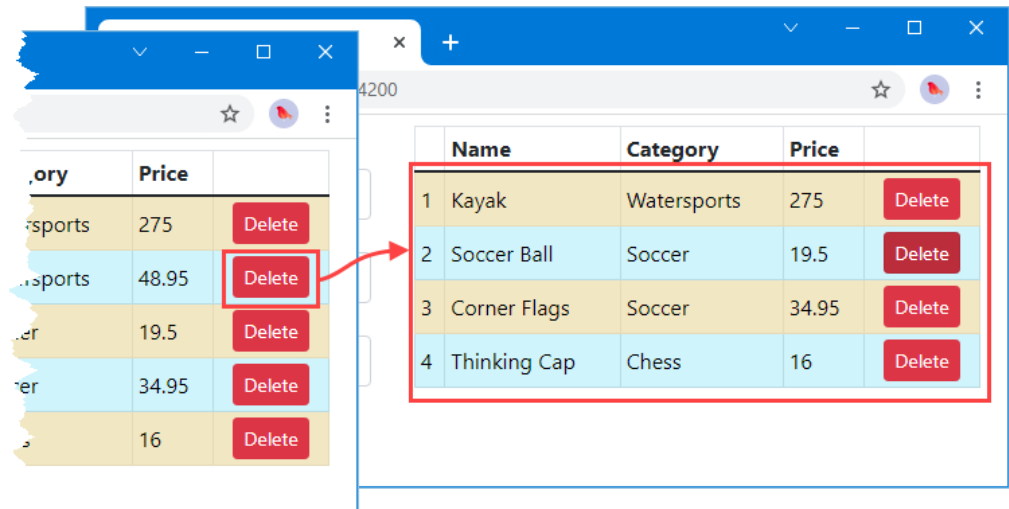


Figure 15.7. Removing objects from the data model

15.4 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 18.

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 15.21.

Listing 15.21. 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 15.22.

Listing 15.22. 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 15.23, 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 15.23. 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 products(); 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 15.24 adds the `darkColor` property to the component.

Listing 15.24. Defining a property in the `component.ts` file in the `src/app` folder

```

import { Component, computed } from "@angular/core";
import { Model } from "../repository.model";
import { Product } from "../product.model";

@Component({
  selector: "app",
  templateUrl: "template.html"
})
export class ProductComponent {
  private model: Model = new Model();
  showTable: boolean = false;
  darkColor: boolean = false;

  products = computed<Product[]>(() => this.model.Products());

  count = computed<number>(() => this.products().length);

  product(key: number): Product | undefined {
    return this.model.getProduct(key);
  }

  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 15.25.

Listing 15.25. 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 { FormsModule } from '@angular/forms';

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

import { ProductComponent } from './component';
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
  ],
  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 15.8. (If you don't see the color change, then restart the Angular development tools and reload the browser.)

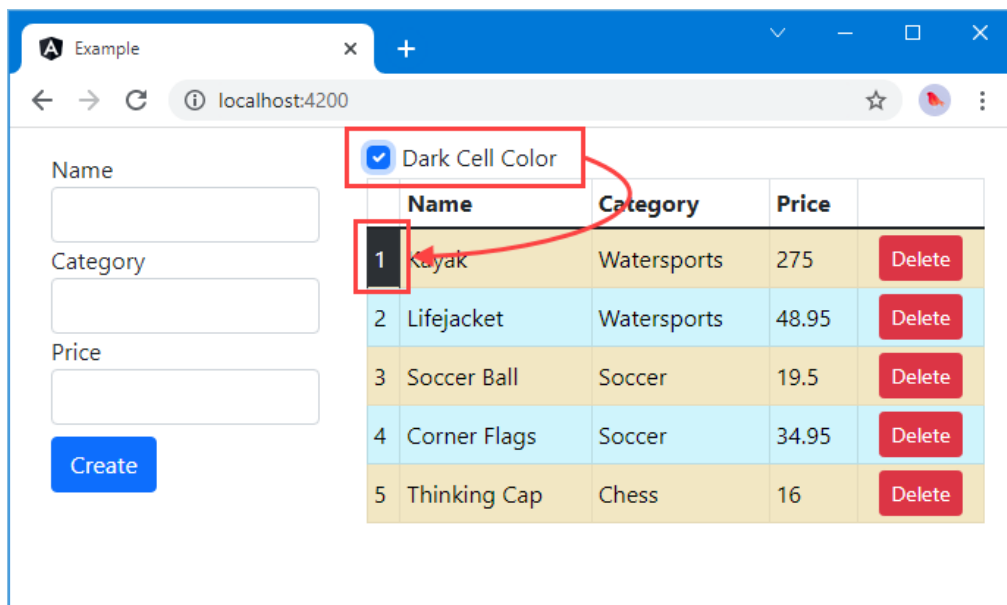


Figure 15.8. Operating on a content child

15.4.1 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 15.26.

Listing 15.26. Querying 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) => {
      if (dark) {
        child.setColor(index % 2 ? dark : !dark);
      } else {
        child.setColor(false);
      }
    });
  }
}

```

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 15.6. The `descendants` configuration property is used to select descendant elements, and without this value, only direct children are selected.

Table 15.6. 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 <code>true</code> , 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 15.9 shows the effect when the checkbox is selected.

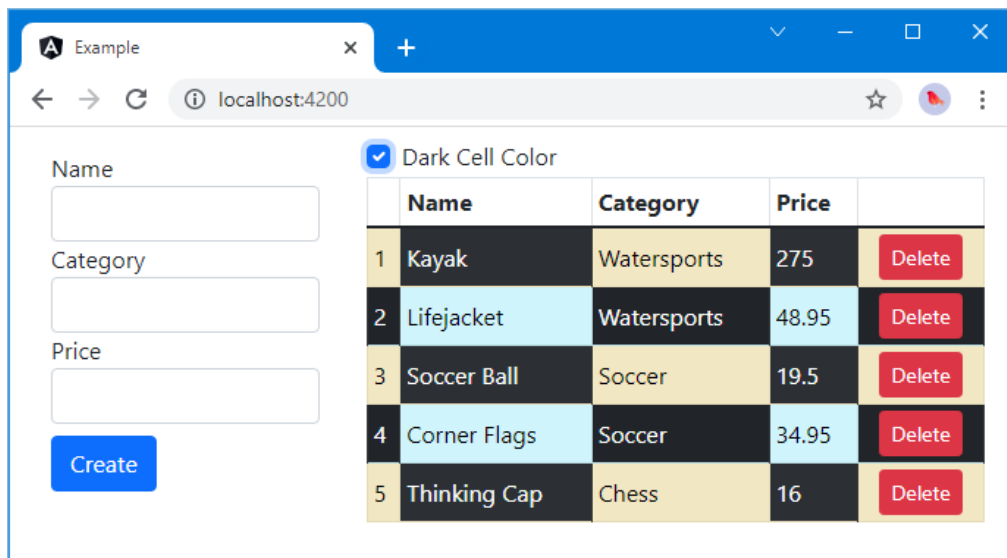


Figure 15.9. Operating on multiple content children

15.4.2 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.

In listing 15.27, I have updated the `PaCellColorSwitcher` directive so that it receives notifications when the set of content children in the `QueryList` changes.

Listing 15.27. Notifications in the `cellColorSwitcher.directive.ts` file in the `src/app` folder

```
import { Directive, Input, SimpleChanges, ContentChildren, QueryList,
  ChangeDetectorRef } from "@angular/core";
import { PaCellColor } from "../cellColor.directive";

@Directive({
  selector: "table"
})
export class PaCellColorSwitcher {

  constructor(private changeRef: ChangeDetectorRef) {}

  @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) => {
      if (dark) {
        child.setColor(index % 2 ? dark : !dark);
      } else {
        child.setColor(false);
      }
    });
    this.changeRef.detectChanges();
  }
}

```

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 problem this presents is that the notifications are received asynchronously, which means that any subsequent alterations cause Angular to think that the code directive is making unstable changes and report an error. To avoid this issue, the constructor receives a `ChangeDetectorRef` object, which provides access to the Angular change detection functionality, whose most useful method is described in table 15.7. (The `ChangeDetectorRef` object is created as part of the Angular services feature, which I describe in chapter 18).

Table 15.7. The `ChangeDetectorRef` method

Name	Description
<code>detectChanges()</code>	This method triggers the change detection process.

Calling the `detectChanges` method causes Angular to perform the change detection process, which ensures that the changes made following the notifications don't produce an error.

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 15.10.

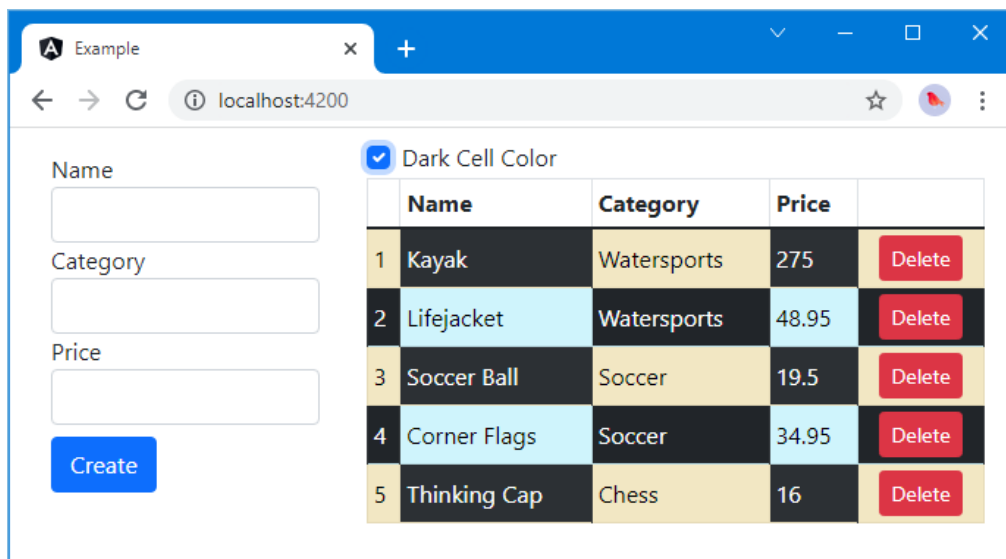


Figure 15.10. Acting on content query change notifications

15.5 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.

- Structural directives add and remove HTML elements.
- Structural directives are created with the `@Directive` decorator.
- The concise syntax is used to apply a directive directly to the element that will be transformed.
- Iterating directives generate content for each value in a sequence.
- Structural directives can query the children of the element to which they are applied.

In the next chapter, I introduce components and explain how they differ from directives.

16

Understanding components

This chapter covers

- Defining component classes and templates
- Binding templates to component classes
- Using input and output properties in components
- Styling content generated by components
- Querying content generated by components

Components are directives that have their own templates, rather than relying on content provided from elsewhere. Components have access to all the directive features described in earlier chapters and still have a host element, can still define input and output properties, and so on. But they also define their own content using templates.

It can be easy to underestimate the importance of the template, but attribute and structural directives have limitations. Directives can do useful and powerful work, but they don't have much insight into the elements they are applied to. Directives are most useful when they are general-purpose tools, such the `ngModel` directive, which can be applied to any data model property and any form element, without regard to what the data or the element is being used for.

Components, by contrast, are closely tied to the contents of their templates. Components provide the data and logic that will be used by the data bindings that are applied to the HTML elements in the template, which provide the context used to evaluate data binding expressions and act as the glue between the directives and the rest of the application. Components are also a useful tool in allowing large Angular projects to be broken up into manageable chunks.

In this chapter, I explain how components work and explain how to restructure an application by introducing some additional components. Table 16.1 puts components in context.

Table 16.1. Putting components in context

Question	Answer
What are they?	Components are directives that define their own HTML content and, optionally, CSS styles.
Why are they useful?	Components make it possible to define self-contained blocks of functionality, which makes projects more manageable and allows for functionality to be more readily reused.
How are they used?	The <code>@Component</code> decorator is applied to a class, which is registered in the application's Angular module.
Are there any pitfalls or limitations?	No. Components provide all the functionality of directives, with the addition of providing their own templates.
Are there any alternatives?	An Angular application must contain at least one component, which is used in the bootstrap process. Aside from this, you don't have to add additional components, although the resulting application becomes unwieldy and difficult to manage.

Table 16.2 summarizes the chapter.

Table 16.2. Chapter summary

Problem	Solution	Listing
Creating a component	Apply the <code>@Component</code> directive to a class	1–5
Defining the content displayed by a component	Create an inline or external template	6–8
Including data in a template	Use a data binding in the component's template	9
Coordinating between components	Use input or output properties	10–17
Displaying content in an element to which a component has been applied	Project the host element's content	18–21
Styling component content	Create component styles	22–25
Querying the content in the component's template	Use the <code>@ViewChildren</code> decorator	26

16.1 Preparing the example project

In this chapter, I continue using the example project that I created in chapter 9 and have been modifying since. No changes are required to prepare for this chapter.

TIP You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/manningbooks/pro-angular-16>. See chapter 1 for how to get help if you have problems running the examples.

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 in figure 16.1.

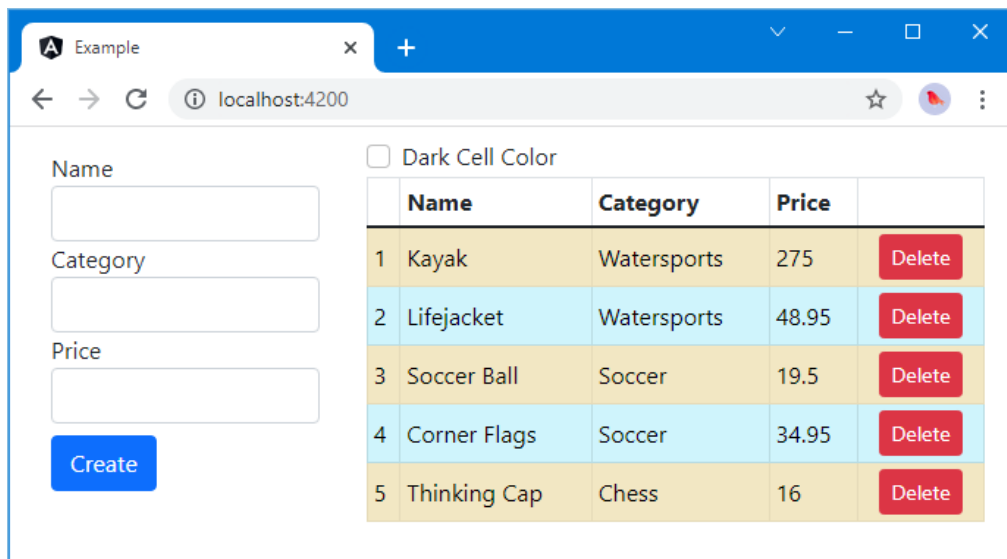


Figure 16.1. Running the example project

16.2 Structuring an application with components

At the moment, the example project contains only one component and one template. Angular applications require at least one component, known as the *root component*, which is the entry point specified in the Angular module.

The problem with having only one component is that it ends up containing the logic required for all the application's features, with its template containing all the markup required to expose those features to the user. The result is that a single component and its template are responsible for handling a lot of tasks. The component in the example application is responsible for the following:

- Providing Angular with an entry point into the application, as the root component
- Providing access to the application's data model so that it can be used in data bindings
- Defining the HTML form used to create new products
- Defining the HTML table used to display products
- Defining the layout that contains the form and the table
- Maintaining state information used to prevent invalid data from being used to create data
- Maintaining state information about whether the table should be displayed

A lot is going on for such a simple application, and not all of these tasks are related. This effect tends to creep up gradually as development proceeds, but it means that the application is harder to test because individual features can't be isolated effectively, and it is harder to enhance and maintain because the code and markup become increasingly complex.

Adding components to the application allows features to be separated into building blocks that can be used repeatedly in different parts of the application and tested in isolation. In the sections that follow, I create components that break up the functionality contained in the example application into manageable, reusable, and self-contained units. Along the way, I'll explain the different features that components provide beyond those available to directives. To prepare for these changes, I have simplified the existing component's template, as shown in listing 16.1.

Listing 16.1. Simplifying the content of the template.html file in the src/app folder

```
<div class="container-fluid">
  <div class="row p-2">
    <div class="col-4 p-2 bg-success text-white">
      Form will go here
    </div>
    <div class="col p-2 bg-primary text-white">
      table will go here
    </div>
  </div>
</div>
```

When you save the changes to the template, you will see the content in figure 16.2. The placeholders will be replaced with application functionality as I develop the new components and add them to the application.

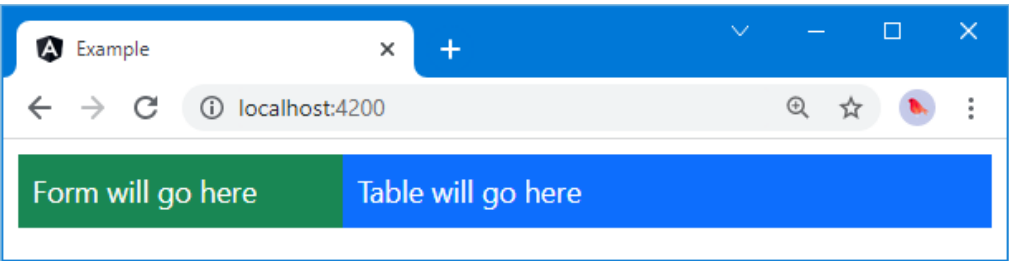


Figure 16.2. Simplifying the existing template

16.2.1 Creating new components

To create a new component, I added a file called `productTable.component.ts` to the `src/app` folder and used it to define the component shown in listing 16.2.

Listing 16.2. The contents of the `productTable.component.ts` file in the `src/app` folder

```
import { Component } from "@angular/core";

@Component({
  selector: "paProductTable",
  template: "<div>This is the table component</div>"
})
export class ProductTableComponent {
}
```

A component is a class to which the `@Component` decorator has been applied. This is as simple as a component can get, and it provides just enough functionality to count as a component without doing anything useful.

The naming convention for the files that define components is to use a descriptive name that suggests the purpose of the component, followed by a period and then `component.ts`. For this component, which will be used to generate the table of products, the filename is `productTable.component.ts`. The name of the class should be equally descriptive. This component's class is named `ProductTableComponent`.

The `@Component` decorator describes and configures the component. The most useful decorator properties are described in table 16.3, which also includes details of where they are described (not all of them are covered in this chapter).

Table 16.3. Useful component decorator properties

Name	Description
<code>selector</code>	This property is used to specify the CSS selector used to match host elements, as described after the table.

styles	This property is used to define CSS styles that are applied only to the component's template. The styles are defined inline, as part of the TypeScript file. See the "Using Component Styles" section for details.
styleUrls	This property is used to define CSS styles that are applied only to the component's template. The styles are defined in separate CSS files. See the "Using Component Styles" section for details.
template	This property is used to specify an inline template, as described in the "Defining Templates" section.
templateUrl	This property is used to specify an external template, as described in the "Defining Templates" section.

For the second component, I created a file called `productForm.component.ts` in the `src/app` folder and added the code shown in listing 16.3.

Listing 16.3. The contents of the `productForm.component.ts` file in the `src/app` folder

```
import { Component } from "@angular/core";

@Component({
  selector: "paProductForm",
  template: "<div>This is the form component</div>"
})
export class ProductFormComponent {

}
```

This component is equally simple and is just a placeholder for the moment. Later in the chapter, I'll add some more useful features. To enable the components, they must be declared in the application's Angular module, as shown in listing 16.4.

Listing 16.4. Enabling new components in the `app.module.ts` file in the `src/app` folder

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

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

import { ProductComponent } from './component';
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';
import { ProductTableComponent } from './productTable.component';
import { ProductFormComponent } from './productForm.component';

@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    ProductComponent,
    PaAttrDirective,
    PaModel,
    PaStructureDirective,
    PaIteratorDirective,
    PaCellColor,
    PaCellColorSwitcher,
    ProductTableComponent,
    ProductFormComponent
  ],
  declarations: [
    AppComponent
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

```

    declarations: [
      ProductComponent, PaAttrDirective, PaModel, PaStructureDirective,
      PaIteratorDirective, PaCellColor, PaCellColorSwitcher,
      ProductTableComponent, ProductFormComponent
    ],
    imports: [
      BrowserModule,
      BrowserAnimationsModule,
      FormsModule
    ],
    providers: [],
    bootstrap: [ProductComponent]
  })
  export class AppModule { }

```

The component class is brought into scope using an `import` statement and is added to the `NgModule` decorator's `declarations` array. The final step is to add an HTML element that matches the component's selector property, as shown in listing 16.5, which will provide the component with its host element.

Listing 16.5. Adding a host element in the `template.html` file in the `src/app` folder

```

<div class="container-fluid">
  <div class="row p-2">
    <div class="col-4 p-2 bg-success text-white">
      <paProductForm></paProductForm>
    </div>
    <div class="col p-2 bg-primary text-white">
      <paProductTable></paProductTable>
    </div>
  </div>
</div>

```

When all the changes have been saved, the browser will display the content shown in figure 16.3, which shows that parts of the HTML document are now under the management of the new components.

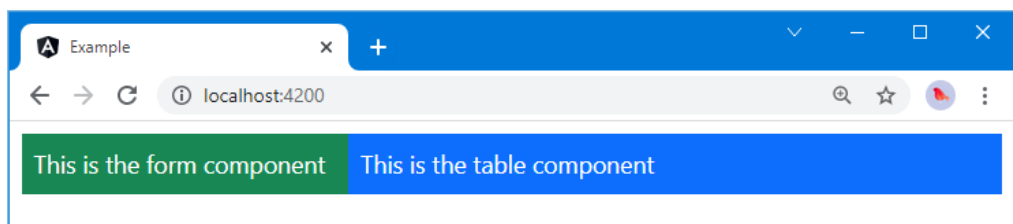


Figure 16.3. Adding new components

UNDERSTANDING THE NEW APPLICATION STRUCTURE

The new components have changed the structure of the application. Previously, the root component was responsible for all the HTML content displayed by the application. Now,

however, there are three components and responsibility for some of the HTML content has been delegated to the new additions, as illustrated in figure 16.4.

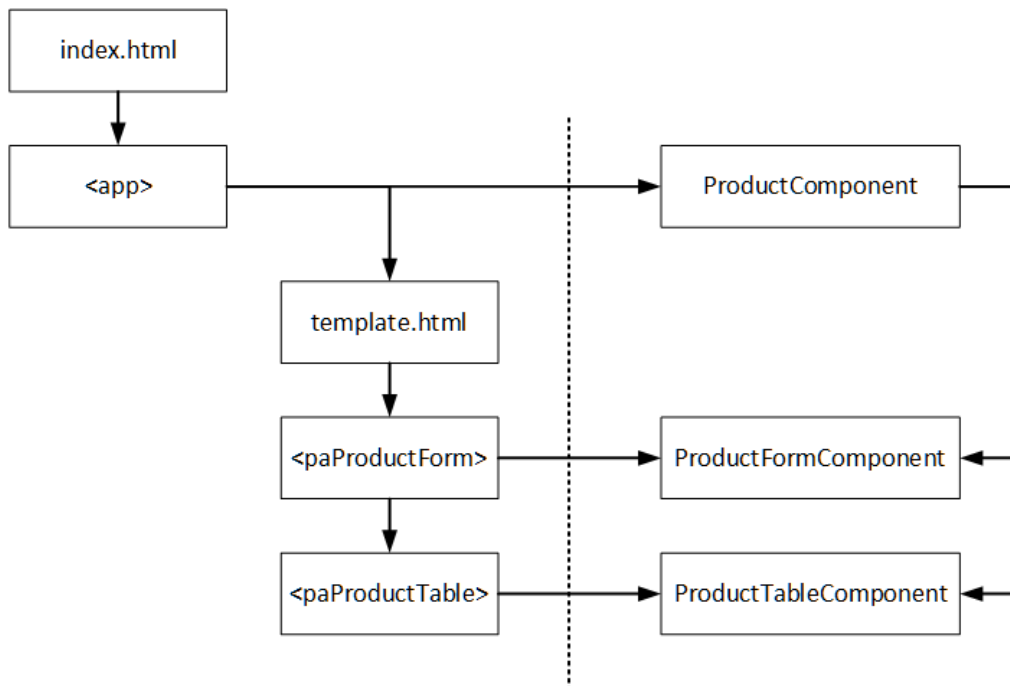


Figure 16.4. The new application structure

When the browser loads the `index.html` file, the Angular bootstrap process starts, and Angular processes the application's module, which provides a list of the components that the application requires. Angular inspects the decorator for each component in its configuration, including the value of the `selector` property, which is used to identify which elements will be hosts.

Angular then begins processing the body of the `index.html` file and finds the `app` element, which is specified by the `selector` property of the `ProductComponent` component. Angular populates the `app` element with the component's template, which is contained in the `template.html` file. Angular inspects the contents of the `template.html` file and finds the `paProductForm` and `paProductTable` elements, which match the `selector` properties of the newly added components. Angular populates these elements with each component's template, producing the placeholder content shown in figure 16.3.

There are some important new relationships to understand. First, the HTML content that is displayed in the browser window is now composed of several templates, each of which is managed by a component. Second, the `ProductComponent` is now the parent component to

the `ProductFormComponent` and `ProductTableComponent` objects, a relationship that is formed by the fact that the host elements for the new components are defined in the `template.html` file, which is the `ProductComponent` template. Equally, the new components are children of the `ProductComponent`. The parent-child relationship is an important one when it comes to Angular components, as you will see as I describe how components work in later sections.

16.2.2 Defining templates

Although there are new components in the application, they don't have much impact at the moment because they display only placeholder content. Each component has a template, which defines the content that will be used to replace its host element in the HTML document. There are two different ways to define templates: inline within the `@Component` decorator or externally in an HTML file.

The new components that I added use templates, where a fragment of HTML is assigned to the `template` property of the `@Component` decorator, like this:

```
...
template: "<div>This is the form component</div>"
...
```

The advantage of this approach is simplicity: the component and the template are defined in a single file, and there is no way that the relationship between them can be confused. The drawback of inline templates is that they can get out of control and be hard to read if they contain more than a few HTML elements.

NOTE Another problem is that editors that highlight syntax errors as you type usually rely on the file extension to figure out what type of checking should be performed and won't realize that the value of the `template` property is HTML and will simply treat it as a string.

If you are using TypeScript, then you can use multiline strings to make inline templates more readable. Multiline strings are denoted with the backtick character (the ``` character, which is also known as the *grave accent*), and they allow strings to spread over multiple lines, as shown in listing 16.6.

Listing 16.6. A multiline string in the `productTable.component.ts` file in the `src/app` folder

```
import { Component } from "@angular/core";

@Component({
  selector: "paProductTable",
  template: `
```

Multiline strings allow the structure of the HTML elements in a template to be preserved, which makes it easier to read and increases the size of the template that can be practically included inline before it becomes too unwieldy to manage. Figure 16.5 shows the effect of the template in listing 16.6.

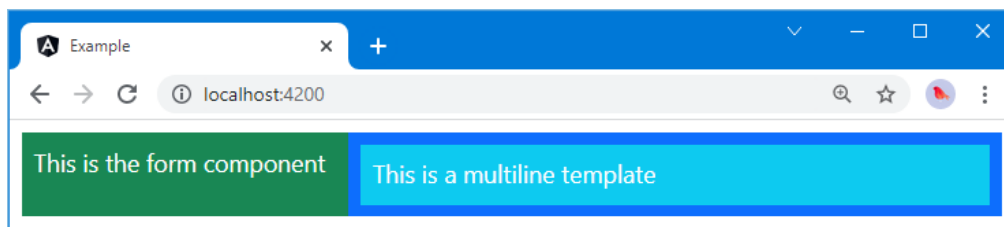


Figure 16.5. Using a multiline inline template

My advice is to use external templates (explained in the next section) for any template that contains more than two or three simple elements, largely to take advantage of the HTML editing and syntax highlighting features provided by modern editors, which can go a long way to reduce the number of errors you discover when running the application.

DEFINING EXTERNAL TEMPLATES

External templates are defined in a different file from the rest of the component. The advantage of this approach is that the code and HTML are not mixed together, which makes both easier to read and unit test, and it also means that code editors will know they are working with HTML content when you are working on a template file, which can help reduce coding-time errors by highlighting errors.

The drawback of external templates is that you have to manage more files in the project and ensure that each component is associated with the correct template file. The best way to do this is to follow a consistent file naming strategy so that it is immediately obvious that a file contains a template for a given component. The convention for Angular is to create pairs of files using the convention `<componentname>.component.<type>` so that when you see a file called `productTable.component.ts`, you know it contains a component called `Products` written in TypeScript, and when you see a file called `productTable.component.html`, you know that it contains an external template for the `Products` component.

TIP The syntax and features for both types of template are the same, and the only difference is where the content is stored, either in the same file as the component code or in a separate file.

To define an external template using the naming convention, I created a file called `productTable.component.html` in the `src/app` folder and added the markup shown in listing 16.7.

Listing 16.7. The productTable.component.html file in the src/app folder

```
<div class="bg-info p-2">
  This is an external template
</div>
```

To specify an external template, the `templateUrl` property is used in the `@Component` decorator, as shown in listing 16.8.

Listing 16.8. The productTable.component.ts file in the src/app folder

```
import { Component } from "@angular/core";

@Component({
  selector: "paProductTable",
  // template: `<div class='bg-info p-2'>
  //           This is a multiline template
  //           </div>`
  templateUrl: "productTable.component.html"
})
export class ProductTableComponent {

}
```

Notice that different properties are used: `template` is for inline templates, and `templateUrl` is for external templates. Figure 16.6 shows the effect of using an external template.

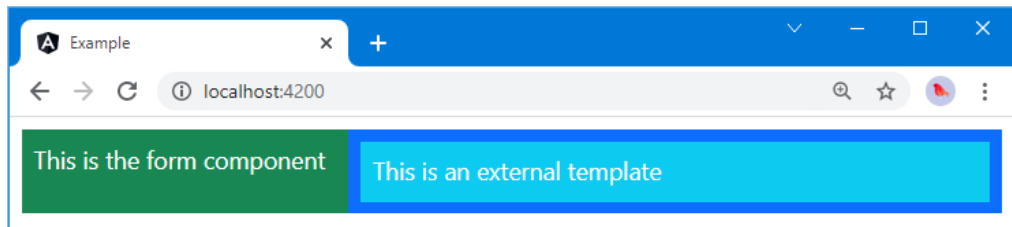


Figure 16.6. Using an external template

USING DATA BINDINGS IN COMPONENT TEMPLATES

A component's template can contain the full range of data bindings and target any of the built-in directives or custom directives that have been registered in the application's Angular module. Each component class provides the context for evaluating the data binding expressions in its template, and by default, each component is isolated from the others. This means the component doesn't have to worry about using the same property and method names that other components use and can rely on Angular to keep everything separate. As an example, listing 16.9 shows the addition of a property called `model` to the form child component, which would conflict with the property of the same name in the root component were they not kept separate.

Listing 16.9. Adding a property in the productForm.component.ts file in the src/app folder

```
import { Component } from "@angular/core";

@Component({
  selector: "paProductForm",
  template: "<div>{{model}}</div>"
})
export class ProductFormComponent {

  model: string = "This is the model";
}
```

The component class uses the `model` property to store a message that is displayed in the template using a string interpolation binding. Figure 16.7 shows the result.

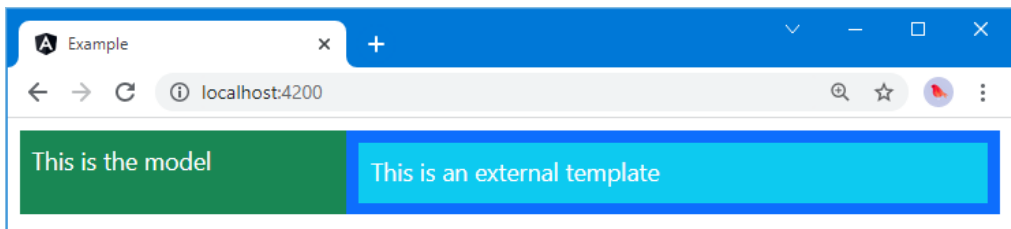


Figure 16.7. Using a data binding in a child component

USING INPUT PROPERTIES TO COORDINATE BETWEEN COMPONENTS

Few components exist in isolation and need to share data with other parts of the application. Components can define input properties to receive the value of data binding expressions on their host elements. The expression will be evaluated in the context of the parent component, but the result will be passed to the child component's property.

To demonstrate, listing 16.10 adds an input property to the table component, which it will use to receive the model data that it should display.

Listing 16.10. Defining an input property in the `productTable.component.ts` file in the `src/app` folder

```
import { Component, Input, Signal } from "@angular/core";
import { Model } from "../repository.model";
import { Product } from "../product.model";

@Component({
  selector: "paProductTable",
  templateUrl: "productTable.component.html"
})
export class ProductTableComponent {

  @Input({ alias: "model", required: true })
  dataModel!: Model;

  get Products(): Signal<Product[]> {
    return this.dataModel.Products;
  }
}
```

```

    getProduct(key: number): Product | undefined {
        return this.dataModel?.getProduct(key);
    }

    deleteProduct(key: number) {
        this.dataModel.deleteProduct(key);
    }
}

```

The component now defines a required input property that will be assigned the value expression assigned to the `model` attribute on the host element. The `Products` property, and the `getProducts` and `deleteProduct` methods use the input property to provide access to the data model to bindings in the component's template, which is modified in listing 16.11.

Listing 16.11. Adding a data binding in the `productTable.component.html` file in the `src/app` folder

There are `{{Products().length}}` items in the model

Providing the child component with the data that it requires means adding a binding to its host element, which is defined in the template of the parent component, as shown in listing 16.12.

Listing 16.12. Adding a data binding in the `template.html` file in the `src/app` folder

```

<div class="container-fluid">
  <div class="row p-2">
    <div class="col-4 p-2 bg-success text-white">
      <paProductForm></paProductForm>
    </div>
    <div class="col p-2 bg-primary text-white">
      <paProductTable [model]="model"></paProductTable>
    </div>
  </div>
</div>

```

The access level for the `model` property must be changed so that it can be accessed in the template, as shown in listing 16.13.

Listing 16.13. Exposing a property in the `component.ts` file in the `src/app` folder

```

...
@Component({
  selector: "app",
  templateUrl: "template.html"
})
export class ProductComponent {
  model: Model = new Model();
  showTable: boolean = false;
  darkColor: boolean = false;

  products = computed<Product[]>(() => this.model.Products());
  ...
}

```

The effect is to provide the child component with access to the parent component's `model` property. This can be a confusing feature because it relies on the fact that the host element is

defined in the parent component's template but that the input property is defined by the child component, as illustrated by figure 16.8.

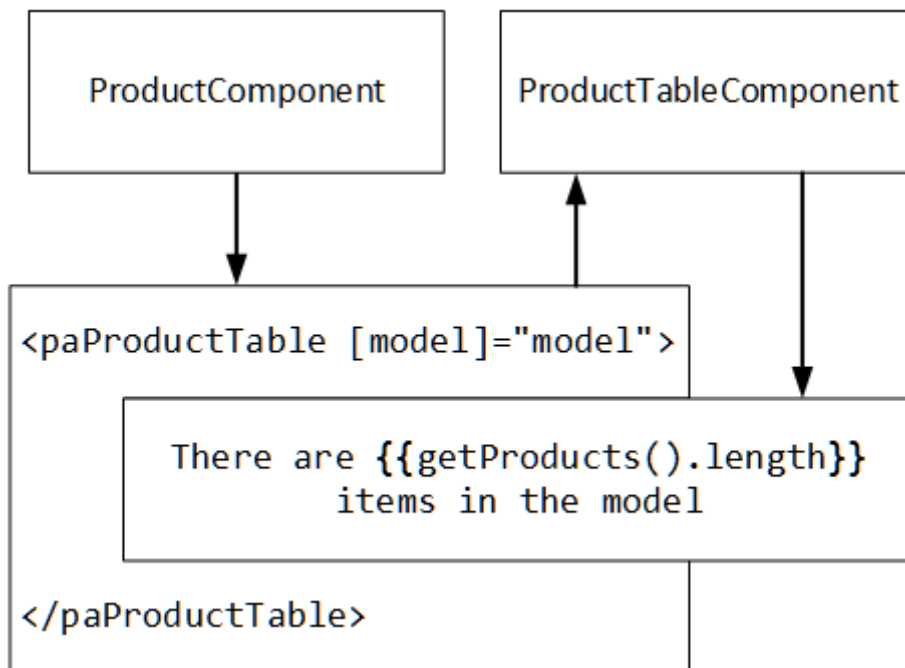


Figure 16.8. Sharing data between parent and child components

The child component's host element acts as the bridge between the parent and child components, and the input property allows the component to provide the child with the data it needs, producing the result shown in figure 16.9.

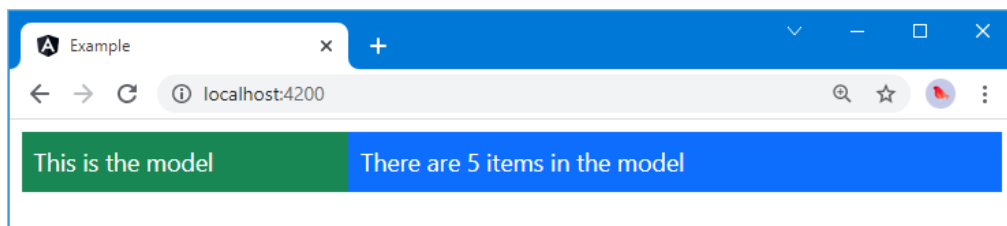


Figure 16.9. Sharing data from a parent to a child component

USING DIRECTIVES IN A CHILD COMPONENT TEMPLATE

Once the input property has been defined, the child component can use the full range of data bindings and directives, either by using the data provided through the parent component or by defining its own. In listing 16.14, I have restored the original table functionality from earlier chapters that displays a list of the `Product` objects in the data model, along with a checkbox that determines whether the table is displayed. This functionality was previously managed by the root component and its template.

Listing 16.14. Restoring the table in the `productTable.component.html` file in the `src/app` folder

```
<table class="table table-sm table-bordered table-striped">
  <thead class="table-light">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th>
    <th></th></tr>
  </thead>
  <tbody>
    <tr *paFor="let item of Products(); 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 same HTML elements, data bindings, and directives (including custom directives like `paIf` and `paFor`) are used, producing the result shown in figure 16.10. The key difference is not in the appearance of the table but in the way that it is now managed by a dedicated component.

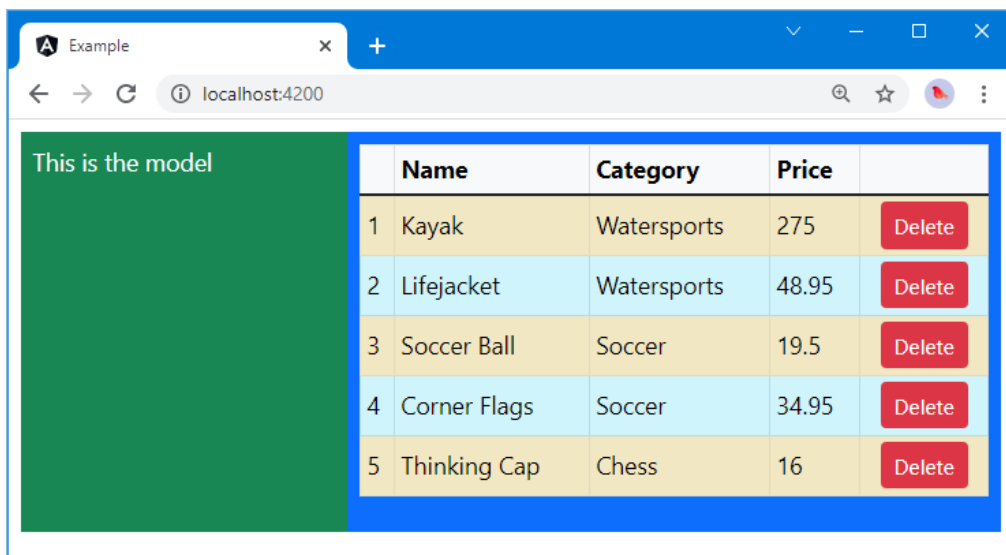


Figure 16.10. Restoring the table display

USING OUTPUT PROPERTIES TO COORDINATE COMPONENTS

Child components can use output properties that define custom events that signal important changes and that allow the parent component to respond when they occur. Listing 16.15 changes the form component, adding an external template and an output property that will be triggered when the user creates a new `Product` object when invoking the `submitForm` method.

Listing 16.15. Defining an output property in the `productForm.component.ts` file in the `src/app` folder

```
import { Component, Output, EventEmitter } from "@angular/core";
import { Product } from "../product.model";

@Component({
  selector: "paProductForm",
  templateUrl: "productForm.component.html"
})
export class ProductFormComponent {
  newProduct: Product = new Product();

  @Output("paNewProduct")
  newProductEvent = new EventEmitter<Product>();

  submitForm(form: any) {
    this.newProductEvent.emit(this.newProduct);
    this.newProduct = new Product();
    form.resetForm();
  }
}
```

```
}
```

The output property is called `newProductEvent`, and the component triggers it when the `submitForm` method is called. Aside from the output property, the additions in the listing are based on the logic in the root controller, which previously managed the form. I also removed the inline template and created a file called `productForm.component.html` in the `src/app` folder, with the content shown in listing 16.16.

Listing 16.16. The `productForm.component.html` file in the `src/app` folder

```
<form #form="ngForm" (ngSubmit)="submitForm(form)">
  <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="name" [(ngModel)]="newProduct.price" />
  </div>
  <button class="btn btn-primary mt-2" type="submit">
    Create
  </button>
</form>
```

The form contains standard elements, configured using two-way bindings. The child component's host element acts as the bridge to the parent component, which can register interest in the custom event, as shown in listing 16.17.

Listing 16.17. Using the event in the `template.html` file in the `src/app` folder

```
<div class="container-fluid">
  <div class="row p-2">
    <div class="col-4 p-2 text-dark">
      <paProductForm (paNewProduct)="addProduct($event)">
    </paProductForm>
    </div>
    <div class="col p-2">
      <paProductTable [model]="model"></paProductTable>
    </div>
  </div>
</div>
```

The new binding handles the custom event by passing the event object to the `addProduct` method. The child component is responsible for managing the form elements and validating their contents. When the data passes validation, the custom event is triggered, and the data binding expression is evaluated in the context of the parent component, whose `addProduct` method adds the new object to the model. Since the model has been shared with the table child component through its input property, the new data is displayed to the user, as shown in

figure 16.11. (You may need to restart the Angular development tools to include the new template file in the build process.)

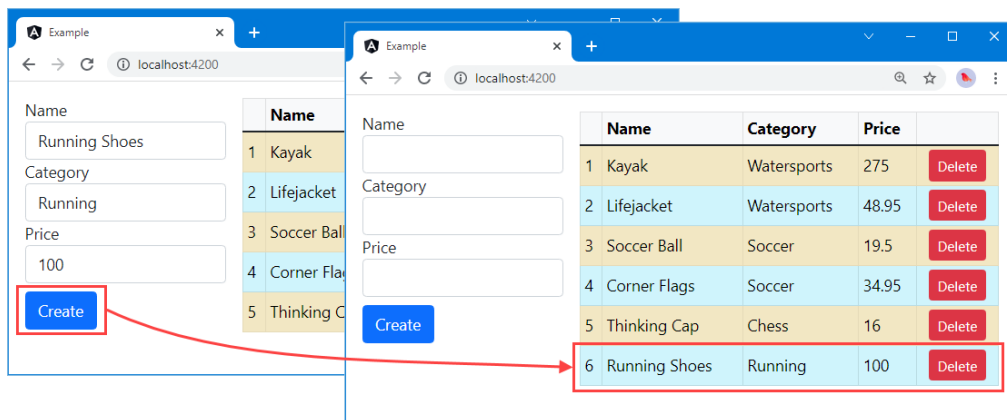


Figure 16.11. Using a custom event in a child component

PROJECTING HOST ELEMENT CONTENT

If the host element for a component contains content, it can be included in the template using the special `ng-content` element. This is known as *content projection*, and it allows components to be created that combine the content in their template with the content in the host element. To demonstrate, I added a file called `toggleView.component.ts` to the `src/app` folder and used it to define the component shown in listing 16.18.

Listing 16.18. The contents of the `toggleView.component.ts` file in the `src/app` folder

```
import { Component } from "@angular/core";

@Component({
  selector: "paToggleView",
  templateUrl: "toggleView.component.html"
})
export class PaToggleView {
  showContent: boolean = true;
}
```

This component defines a `showContent` property that will be used to determine whether the host element's content will be displayed within the template. To provide the template, I added a file called `toggleView.component.html` to the `src/app` folder and added the elements shown in listing 16.19.

Listing 16.19. The contents of the `toggleView.component.html` file in the `src/app` folder

```
<div class="form-check">
  <label class="form-check-label">Show Content</label>
```

```

        <input class="form-check-input" type="checkbox"
          [(ngModel)]="showContent" />
      </div>
      <ng-content *ngIf="showContent"></ng-content>

```

The important element is `ng-content`, which Angular will replace with the content of the host element. The `ngIf` directive has been applied to the `ng-content` element so that it will be visible only if the checkbox in the template is checked. Listing 16.20 registers the component with the Angular module.

Listing 16.20. Registering 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 { FormsModule } from '@angular/forms';

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

import { ProductComponent } from './component';
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';
import { ProductTableComponent } from './productTable.component';
import { ProductFormComponent } from './productForm.component';
import { PaToggleView } from './toggleView.component';

@NgModule({
  declarations: [
    ProductComponent, PaAttrDirective, PaModel, PaStructureDirective,
    PaIteratorDirective, PaCellColor, PaCellColorSwitcher,
    ProductTableComponent, ProductFormComponent, PaToggleView
  ],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule
  ],
  providers: [],
  bootstrap: [ProductComponent]
})
export class AppModule { }

```

The final step is to apply the new component to a host element that contains content, as shown in listing 16.21.

Listing 16.21. Adding a host element with Content in the `template.html` file in the `src/app` folder

```

<div class="container-fluid">
  <div class="row p-2">
    <div class="col-4 p-2 text-dark">
      <paProductForm (paNewProduct)="addProduct($event)">

```

```

        </paProductForm>
      </div>
      <div class="col p-2">
        <paToggleView>
          <paProductTable [model]="model"></paProductTable>
        </paToggleView>
      </div>
    </div>
  </div>

```

The `paToggleView` element is the host for the new component, and it contains the `paProductTable` element, which applies the component that creates the product table. The result is that there is a checkbox that controls the visibility of the table, as shown in figure 16.12. The new component has no knowledge of the content of its host element, and its inclusion in the template is possible only through the `ng-content` element.

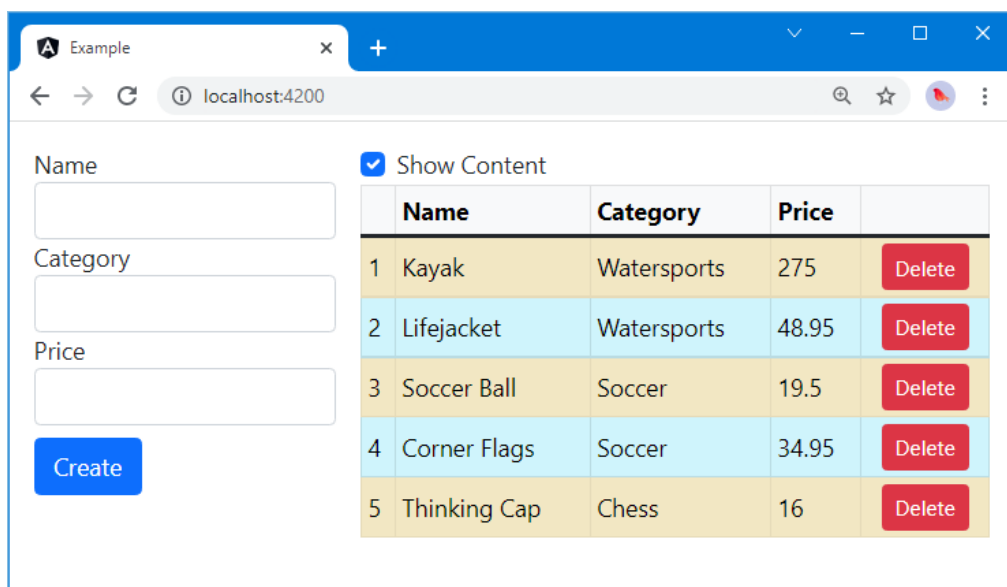


Figure 16.12. Including host element content in the template

Selecting components dynamically

The `ViewContainerRef` class, introduced in chapter 15, defines the `createComponent` method, which can be used to select a component programmatically, without having to specify a fixed element in a template. I have not demonstrated this feature because it has serious limitations and causes more problems than it addresses, at least in my experience. If you want to explore this feature, see the Angular documentation at <https://angular.io/guide/dynamic-component-loader>, but proceed with caution.

16.2.3 Completing the component restructure

The functionality that was previously contained in the root component has been distributed to the new child components. All that remains is to tidy up the root component to remove the code that is no longer required, as shown in listing 16.22.

Listing 16.22. Removing obsolete code in the component.ts file in the src/app folder

```
import { Component, computed } 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;

  // products = computed<Product[]>(() => this.model.Products());

  // count = computed<number>(() => this.products().length);

  // product(key: number): Product | undefined {
  //   return this.model.getProduct(key);
  // }

  // newProduct: Product = new Product();

  addProduct(p: Product) {
    this.model.saveProduct(p);
  }

  // deleteProduct(key: number) {
  //   this.model.deleteProduct(key);
  // }

  // submitForm() {
  //   this.addProduct(this.newProduct);
  // }
}
```

Many of the responsibilities of the root component have been moved elsewhere in the application. Of the original list from the start of the chapter, only the following remain the responsibility of the root component:

- Providing Angular with an entry point into the application, as the root component
- Providing access to the application's data model so that it can be used in data bindings

The child components have assumed the rest of the responsibilities, providing self-contained blocks of functionality that are simpler, easier to develop, and easier to maintain and that can be reused as required.

16.3 Using component styles

Components can define styles that apply only to the content in their templates, which allows content to be styled by a component without it being affected by the styles defined by its parents or other antecedents and without affecting the content in its child and other descendant components. Styles can be defined inline using the `styles` property of the `@Component` decorator, as shown in listing 16.23.

Listing 16.23. Defining inline styles in the `productForm.component.ts` file in the `src/app` folder

```
import { Component, Output, EventEmitter } from "@angular/core";
import { Product } from "../product.model";

@Component({
  selector: "paProductForm",
  templateUrl: "productForm.component.html",
  styles: ["div { background-color: lightgreen }"]
})
export class ProductFormComponent {
  newProduct: Product = new Product();

  @Output("paNewProduct")
  newProductEvent = new EventEmitter<Product>();

  submitForm(form: any) {
    this.newProductEvent.emit(this.newProduct);
    this.newProduct = new Product();
    form.resetForm();
  }
}
```

The `styles` property is set to an array, where each item contains a CSS selector and one or more properties. In the listing, I have specified styles that set the background color of `div` elements to `lightgreen`. Even though there are `div` elements throughout the combined HTML document, this style will affect only the elements in the template of the component that defines them, which is the form component in this case, as shown in figure 16.13.

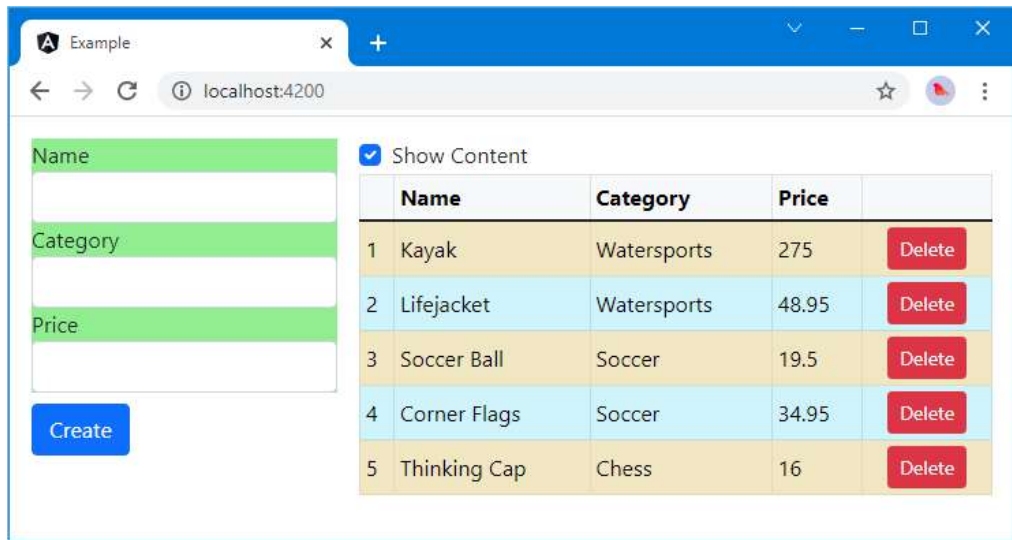


Figure 16.13. Defining inline component styles

TIP The styles included in the bundles created by the development tools are still applied, which is why the elements are still styled using Bootstrap.

16.3.1 Defining external component styles

Inline styles offer the same benefits and drawbacks as inline templates: they are simple and keep everything in one file, but they can be hard to read, can be hard to manage, and can confuse code editors.

The alternative is to define styles in a separate file and associate them with a component using the `styleUrls` property in its decorator. External style files follow the same naming convention as templates and code files. I added a file called `productForm.component.css` to the `src/app` folder and used it to define the styles shown in listing 16.24.

Listing 16.24. The `productForm.component.css` file in the `src/app` folder

```
div {
  background-color: lightgray;
  font-weight: bold;
  font-size: larger;
}
```

In listing 16.25, the component's decorator has been updated to specify the styles file.

Listing 16.25. Using external styles in the `productForm.component.ts` file in the `src/app` folder

```
import { Component, Output, EventEmitter } from "@angular/core";
import { Product } from "../product.model";
```

```

@Component({
  selector: "paProductForm",
  templateUrl: "productForm.component.html",
  //styles: ["div { background-color: lightgreen }"]
  styleUrls: ["productForm.component.css"]
})
export class ProductFormComponent {
  newProduct: Product = new Product();

  @Output("paNewProduct")
  newProductEvent = new EventEmitter<Product>();

  submitForm(form: any) {
    this.newProductEvent.emit(this.newProduct);
    this.newProduct = new Product();
    form.resetForm();
  }
}

```

The `styleUrls` property is set to an array of strings, each of which specifies a CSS file. Figure 16.14 shows the effect of adding the external styles file.

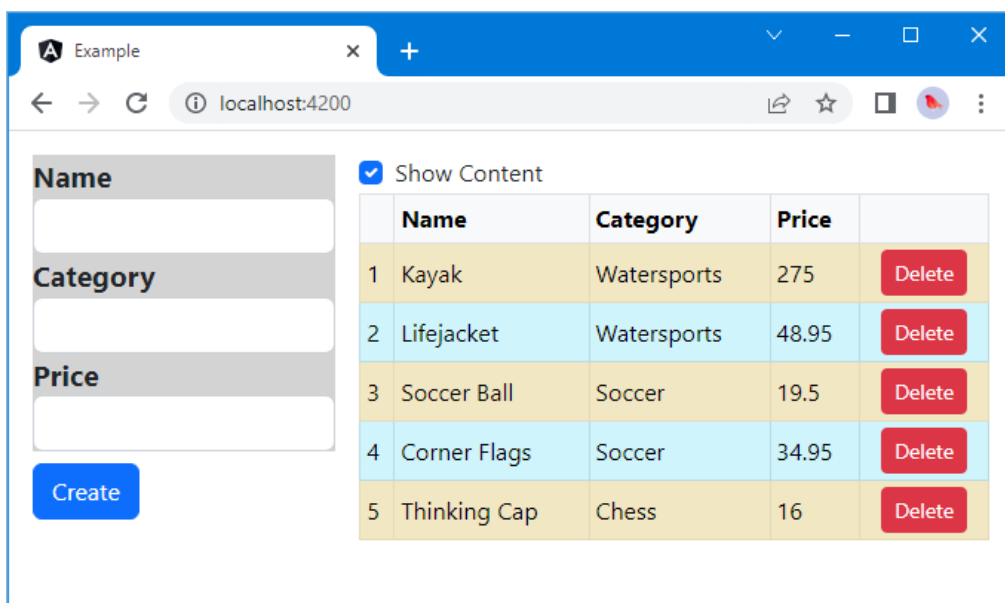


Figure 16.14. Defining external component styles

16.4 Querying template content

Components can query the content of their templates to locate instances of directives or components, which are known as *view children*. These are similar to the directive content children queries that were described in chapter 15 but with some important differences.

In listing 16.26, I have added some code to the component that manages the table that queries for the `PaCellColor` directive that was created to demonstrate directive content queries. This directive is still registered in the Angular module and selects `td` elements, so Angular will have applied it to the cells in the table component's content.

Listing 16.26. Selecting view children in the `productTable.component.ts` file in the `src/app` folder

```
import { Component, Input, Signal, QueryList, ViewChildren,
        ChangeDetectorRef } from "@angular/core";
import { Model } from "../repository.model";
import { Product } from "../product.model";
import { PaCellColor } from "../cellColor.directive";

@Component({
  selector: "paProductTable",
  templateUrl: "productTable.component.html"
})
export class ProductTableComponent {

  constructor(private changeRef: ChangeDetectorRef) {}

  @Input({ alias: "model", required: true })
  dataModel!: Model;

  get Products(): Signal<Product[]> {
    return this.dataModel.Products;
  }

  getProduct(key: number): Product | undefined {
    return this.dataModel?.getProduct(key);
  }

  deleteProduct(key: number) {
    this.dataModel.deleteProduct(key);
  }

  @ViewChildren(PaCellColor)
  viewChildren: QueryList<PaCellColor> | undefined;

  ngAfterViewInit() {
    this.viewChildren?.changes.subscribe(() => {
      this.updateViewChildren();
    });
    this.updateViewChildren();
  }

  private updateViewChildren() {
    this.viewChildren?.forEach((child, index) => {
      child.setColor(index % 2 ? true : false);
    });
  }
}
```

```

    });
    this.changeRef.detectChanges();
  }
}

```

Two property decorators are used to query for directives or components defined in the template, as described in table 16.6.

Table 16.6. The view children query property decorators

Name	Description
@ViewChild(class)	This decorator tells Angular to query for the first directive or component object of the specified type and assign it to the property. The class name can be replaced with a template variable. Multiple classes or variable names can be separated by commas.
@ViewChildren(class)	This decorator assigns all the directive and component objects of the specified type to the property. Template variables can be used instead of classes, and multiple values can be separated by commas. The results are provided in a <code>QueryList</code> object, described in chapter 15.

In the listing, I used the `@ViewChildren` decorator to select all the `PaCellColor` objects from the component's template. Aside from the different property decorators, components have two different lifecycle methods that are used to provide information about how the template has been processed, as described in table 16.7.

Table 16.7. The additional component lifecycle methods

Name	Description
<code>ngAfterViewInit</code>	This method is called when the component's view has been initialized. The results of the view queries are set before this method is invoked.
<code>ngAfterViewChecked</code>	This method is called after the component's view has been checked as part of the change detection process.

In the listing, I implement the `ngAfterViewInit` method to ensure that Angular has processed the component's template and set the result of the query. Within the method, I perform the initial call to the `updateViewChildren` method, which operates on the `PaCellColor` objects, and I set up the function that will be called when the query results change, using the `QueryList.changes` property, as described in chapter 15. The result is that the color of every second table cell is changed, as shown in figure 16.15.

TIP You may need to combine view child and content child queries if you have used the `ng-content` element. The content defined in the template is queried using the technique shown in listing 16.26, but the project content—which replaces the `ng-content` element—is queried using the child queries described in chapter 15.

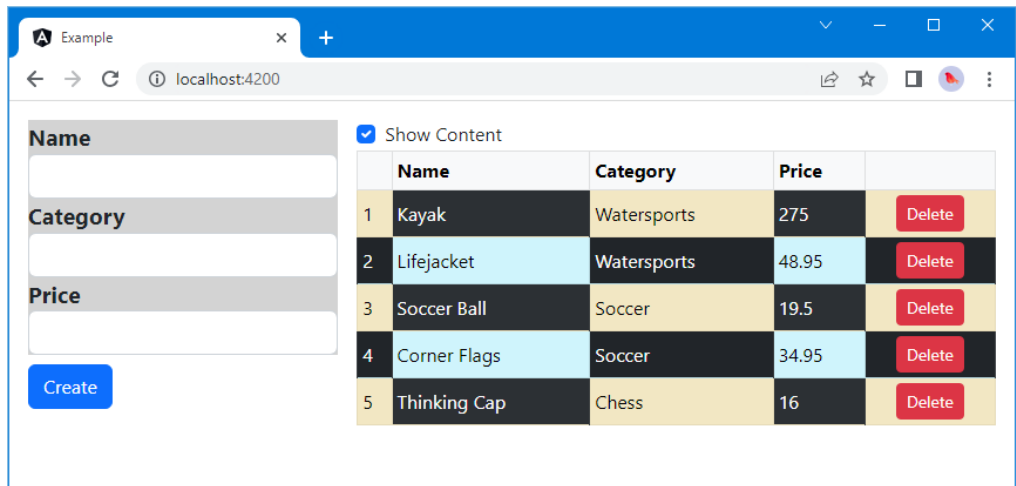


Figure 16.15. Querying for view children

16.5 Summary

In this chapter, I revisited the topic of components and explained how to combine all the features of directives with the ability to provide their own templates. I explained how to structure an application to create small module components and how components can coordinate between themselves using input and output properties. I also showed how components can define CSS styles that are applied only to their templates and no other parts of the application.

- Components are the key Angular feature for presenting HTML content to user.
- Components are defined as classes with templates.
- Component templates contain bindings to properties and methods defined by the class.
- Templates can be defined within the component class or, preferably, using a separate file.
- Components can define input and output properties, allowing integration with other application features.
- Components can define CSS styles that are applied to the content they produce and query the content for specific elements or Angular features.

In the next chapter, I introduce pipes, which are used to prepare data for display in templates.

17

Using and creating pipes

This chapter covers

- Creating and applying pipes
- Using pure and impure pipes to update HTML content
- Formatting data values using the built-in Angular pipes

Pipes are small fragments of code that transform data values so they can be displayed to the user in templates. Pipes allow transformation logic to be defined in self-contained classes so that it can be applied consistently throughout an application. Table 17.1 puts pipes in context.

Table 17.1. Putting pipes in context

Question	Answer
What are they?	Pipes are classes that are used to prepare data for display to the user.
Why are they useful?	Pipes allow preparation logic to be defined in a single class that can be used throughout an application, ensuring that data is presented consistently.
How are they used?	The <code>@Pipe</code> decorator is applied to a class and used to specify a name by which the pipe can be used in a template.
Are there any pitfalls or limitations?	Pipes should be simple and focused on preparing data. It can be tempting to let the functionality creep into areas that are the responsibility of other building blocks, such as directives or components.
Are there any alternatives?	You can implement data preparation code in components or directives, but that makes it harder to reuse in other parts of the application.

Table 17.2 summarizes the chapter.

Table 17.2. Chapter summary

Problem	Solution	Listing
Formatting a data value for inclusion in a template	Use a pipe in a data binding expression	1–5
Creating a custom pipe	Apply the <code>@Pipe</code> decorator to a class	6–8
Formatting a data value using multiple pipes	Chain the pipe names together using the bar character	9
Specifying when Angular should reevaluate the output from a pipe	Use the <code>pure</code> property of the <code>@Pipe</code> decorator	10–13
Formatting numerical values	Use the <code>number</code> pipe	14, 15
Formatting currency values	Use the <code>currency</code> pipe	16, 17
Formatting percentage values	Use the <code>percent</code> pipe	18
Formatting dates	Use the <code>date</code> pipe	19–21
Changing the case of strings	Use the <code>uppercase</code> or <code>lowercase</code> pipe	22, 23
Serializing objects into the JSON format	Use the <code>json</code> pipe	24
Selecting elements from an array	Use the <code>slice</code> pipe	25
Formatting an object or map as key-value pairs	Use the <code>keyvalue</code> pipe	26
Selecting a value to display for a string or number value	Use the <code>i18nSelect</code> or <code>i18nPlural</code> pipe	27–30
Display events from an observable	Use the <code>async</code> pipe	31–33

17.1 Preparing the example project

I am going to continue working with the example project that was first created in chapter 9 and that has been expanded and modified in the chapters since. In the final examples in the previous chapter, component styles and view children queries left the application with a strikingly garish appearance that I am going to tone down for this chapter. In listing 17.1, I have disabled the inline component styles applied to the form elements.

TIP You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/manningbooks/pro-angular-16>. See chapter 1 for how to get help if you have problems running the examples.

Listing 17.1. Disabling CSS styles in the `productForm.component.ts` file in the `src/app` folder

```
import { Component, Output, EventEmitter } from "@angular/core";
import { Product } from "../product.model";

@Component({
  selector: "paProductForm",
  templateUrl: "productForm.component.html",
  //styles: ["div { background-color: lightgreen }"]
  // styleUrls: ["productForm.component.css"]
})
export class ProductFormComponent {
  newProduct: Product = new Product();

  @Output("paNewProduct")
  newProductEvent = new EventEmitter<Product>();

  submitForm(form: any) {
    this.newProductEvent.emit(this.newProduct);
    this.newProduct = new Product();
    form.resetForm();
  }
}
```

To disable the checkerboard coloring of the table cells, I changed the selector for the `PaCellColor` directive so that it matches an attribute that is not currently applied to the HTML elements, as shown in listing 17.2.

Listing 17.2. Changing the selector in the `cellColor.directive.ts` file in the `src/app` folder

```
import { Directive, HostBinding } from "@angular/core";

@Directive({
  selector: "td[paApplyColor]"
})
export class PaCellColor {

  @HostBinding("class")
  bgClass: string = "";

  setColor(dark: Boolean) {
    this.bgClass = dark ? "table-dark" : "";
  }
}
```

The next change is to simplify the `ProductTableComponent` class to remove methods and properties that are no longer required and add new properties that will be used in later examples, as shown in listing 17.3.

Listing 17.3. Simplifying the `productTable.component.ts` file in the `src/app` folder

```

import { Component, Input, Signal, QueryList, ViewChildren,
  ChangeDetectorRef } from "@angular/core";
import { Model } from "../repository.model";
import { Product } from "../product.model";
//import { PaCellColor } from "../cellColor.directive";

@Component({
  selector: "paProductTable",
  templateUrl: "productTable.component.html"
})
export class ProductTableComponent {

  constructor(private changeRef: ChangeDetectorRef) {}

  @Input({ alias: "model", required: true})
  dataModel!: Model;

  get Products(): Signal<Product[]> {
    return this.dataModel.Products;
  }

  getProduct(key: number): Product | undefined {
    return this.dataModel?.getProduct(key);
  }

  deleteProduct(key: number) {
    this.dataModel.deleteProduct(key);
  }

  // @ViewChildren(PaCellColor)
  // viewChildren: QueryList<PaCellColor> | undefined;

  // ngAfterViewInit() {
  //   this.viewChildren?.changes.subscribe(() => {
  //     this.updateViewChildren();
  //   });
  //   this.updateViewChildren();
  // }

  // private updateViewChildren() {
  //   this.viewChildren?.forEach((child, index) => {
  //     child.setColor(index % 2 ? true : false);
  //   });
  //   this.changeRef.detectChanges();
  // }

  taxRate: number = 0;
  categoryFilter: string | undefined;
  itemCount: number = 3;
}

```

Finally, I have removed one of the component elements from the root component's template to disable the checkbox that shows and hides the table, as shown in listing 17.4.

Listing 17.4. Simplifying the elements in the template.html file in the src/app folder

```
<div class="container-fluid">
```

```

<div class="row p-2">
  <div class="col-4 p-2 text-dark">
    <paProductForm (paNewProduct)="addProduct($event)">
    </paProductForm>
  </div>
  <div class="col p-2">
    <!-- <paToggleView> -->
    <paProductTable [model]="model"></paProductTable>
    <!-- </paToggleView> -->
  </div>
</div>
</div>

```

Run the following command in the `example` folder to start the Angular development tools:

```
ng serve
```

Open a new browser tab and navigate to <http://localhost:4200> to see the content shown in figure 17.1.

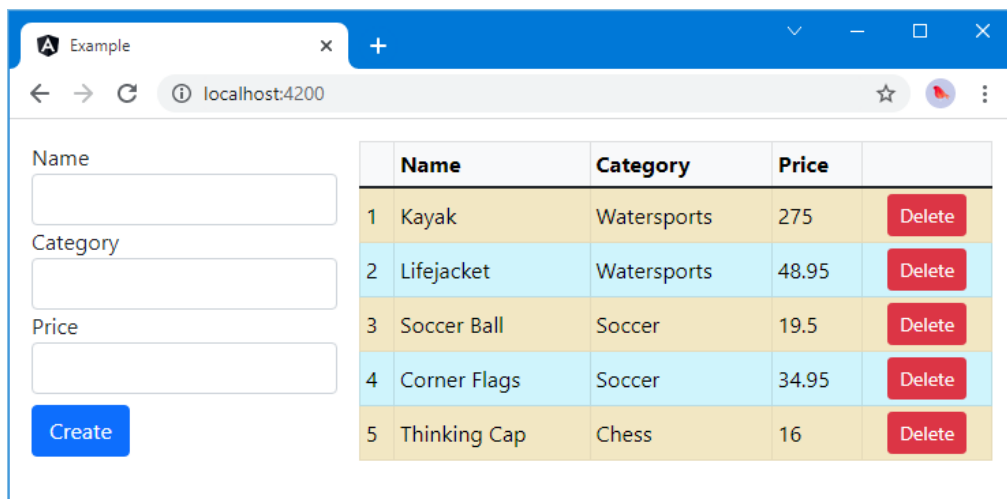


Figure 17.1. Running the example application

17.2 Understanding pipes

Pipes are classes that transform data before it is received by a directive or component. That may not sound like an important job, but pipes can be used to perform some of the most commonly required development tasks easily and consistently.

As a quick example to demonstrate how pipes are used, listing 17.5 applies one of the built-in pipes to transform the values in the `Price` column of the table displayed by the application.

Listing 17.5. Using a pipe in the `productTable.component.html` file in the `src/app` folder

```

<table class="table table-sm table-bordered table-striped">
  <thead class="table-light">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th>

```

```

        <th></th></tr>
    </thead>
    <tbody>
        <tr *paFor="let item of Products(); 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 | currency:"USD":"symbol" }}</td>
            <td class="text-center">
                <button class="btn btn-danger btn-sm"
                    (click)="deleteProduct(item.id)">
                    Delete
                </button>
            </td>
        </tr>
    </tbody>
</table>

```

The syntax for applying a pipe is similar to the style used by command prompts, where a value is “piped” for transformation using the vertical bar symbol (the | character). Figure 17.2 shows the structure of the data binding that contains the pipe.

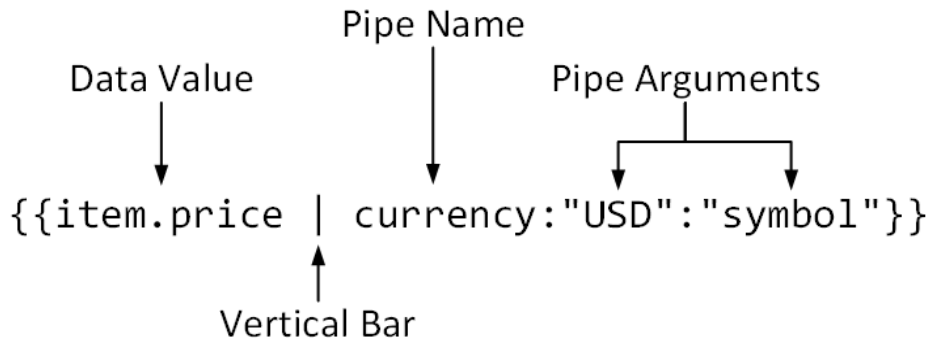


Figure 17.2. The anatomy of data binding with a pipe

The name of the pipe used in listing 17.5 is `currency`, and it formats numbers into currency values. Arguments to the pipe are separated by colons (the `:` character). The first pipe argument specifies the currency code that should be used, which is `USD` in this case, representing U.S. dollars. The second pipe argument, which is `symbol`, specifies whether the currency symbol, rather than its code, should be displayed.

When Angular processes the expression, it obtains the data value and passes it to the pipe for transformation. The result produced by the pipe is then used as the expression result for

the data binding. In the example, the bindings are string interpolations, and figure 17.3 shows the results.

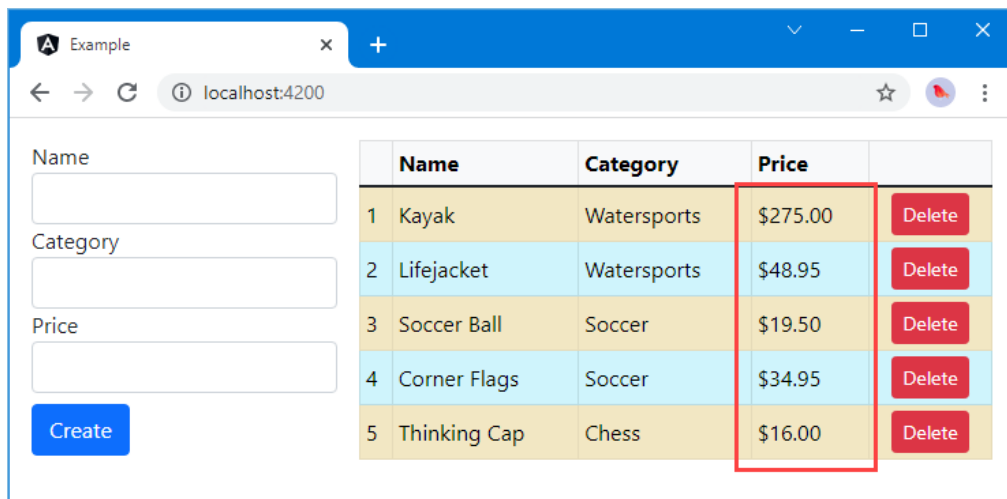


Figure 17.3. The effect of using the currency pipe

17.3 Creating a custom pipe

I will return to the built-in pipes that Angular provides later in the chapter, but the best way to understand how pipes work and what they are capable of is to create a custom pipe. I added a file called `addTax.pipe.ts` in the `src/app` folder and defined the class shown in listing 17.6.

Listing 17.6. The contents of the `addTax.pipe.ts` file in the `src/app` folder

```
import { Pipe } from "@angular/core";

@Pipe({
  name: "addTax"
})
export class PaAddTaxPipe {

  defaultRate: number = 10;

  transform(value: any, rate?: any): number {
    let valueNumber = Number.parseFloat(value);
    let rateNumber = rate == undefined ?
      this.defaultRate : Number.parseInt(rate);
    return valueNumber + (valueNumber * (rateNumber / 100));
  }
}
```

Pipes are classes to which the `Pipe` decorator has been applied and that implement a method called `transform`. The `Pipe` decorator defines two properties, which are used to configure pipes, as described in table 17.3.

Table 17.3. The pipe decorator properties

Name	Description
<code>name</code>	This property specifies the name by which the pipe is applied in templates.
<code>pure</code>	When <code>true</code> , this pipe is reevaluated only when its input value or its arguments are changed. This is the default value. See the “Creating impure pipes” section for details.

The example pipe is defined in a class called `PaAddTaxPipe`, and its decorator `name` property specifies that the pipe will be applied using `addTax` in templates. The `transform` method must accept at least one argument, which Angular uses to provide the data value that the pipe formats. The pipe does its work in the `transform` method, and its result is used by Angular in the binding expression. In this example, the `transform` method accepts a number value, and its result is the received value plus sales tax.

The `transform` method can also define additional arguments that are used to configure the pipe. In the example, the optional `rate` argument can be used to specify the sales tax rate, which defaults to 10 percent.

CAUTION Be careful when dealing with the arguments received by the `transform` method and make sure that you parse or convert them to the types you need. The TypeScript type annotations are not enforced at runtime, and Angular will pass you whatever data values it is working with.

17.3.1 Registering a custom pipe

Pipes are registered using the `declarations` property of the Angular module, as shown in listing 17.7.

Listing 17.7. Registering a custom pipe in the `app.module.ts` file in the `src/app` folder

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

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

import { ProductComponent } from './component';
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';
```

```

import { ProductTableComponent } from "../productTable.component";
import { ProductFormComponent } from "../productForm.component";
import { PaToggleView } from "../toggleView.component";
import { PaAddTaxPipe } from '../addTax.pipe';

@NgModule({
  declarations: [
    ProductComponent, PaAttrDirective, PaModel, PaStructureDirective,
    PaIteratorDirective, PaCellColor, PaCellColorSwitcher,
    ProductTableComponent, ProductFormComponent, PaToggleView,
    PaAddTaxPipe
  ],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule
  ],
  providers: [],
  bootstrap: [ProductComponent]
})
export class AppModule { }

```

17.3.2 Applying a custom pipe

Once a custom pipe has been registered, it can be used in data binding expressions. In listing 17.8, I have applied the pipe to the `price` value in the tables and added a `select` element that allows the tax rate to be specified.

Listing 17.8. Applying the pipe in the `productTable.component.html` file in the `src/app` folder

```

<div class="my-2">
  <label>Tax Rate:</label>
  <select class="form-select" [value]="taxRate || 0"
    (change)="taxRate=$any($event).target.value">
    <option value="0">None</option>
    <option value="10">10%</option>
    <option value="20">20%</option>
    <option value="50">50%</option>
  </select>
</div>

<table class="table table-sm table-bordered table-striped">
  <thead class="table-light">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th>
    <th></th></tr>
  </thead>
  <tbody>
    <tr *paFor="let item of Products(); 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 | addTax:(taxRate || 0) }}</td>
<td class="text-center">
  <button class="btn btn-danger btn-sm"
    (click)="deleteProduct(item.id)">
    Delete
  </button>
</td>
</tr>
</tbody>
</table>

```

Just for variety, I defined the tax rate entirely within the template. The `select` element has a binding that sets its `value` property to a component variable called `taxRate` or defaults to `0` if the property has not been defined. The event binding handles the `change` event and sets the value of the `taxRate` property. You cannot specify a fallback value when using the `ngModel` directive, which is why I have split up the bindings.

In applying the custom pipe, I used the vertical bar character, followed by the value specified by the `name` property in the pipe's decorator. The name of the pipe is followed by a colon, which is followed by an expression that is evaluated to provide the pipe with its argument. In this case, the `taxRate` property will be used if it has been defined, with a fallback value of zero.

Pipes are part of the dynamic nature of Angular data bindings, and the pipe's `transform` method will be called to get an updated value if the underlying data value changes or if the expression used for the arguments changes. The dynamic nature of pipes can be seen by changing the value displayed by the `select` element, which will define or change the `taxRate` property, which will, in turn, update the amount added to the `price` property by the custom pipe, as shown in figure 17.4.

	Name	Category	Price	
1	Kayak	Watersports	330	Delete
2	Lifejacket	Watersports	58.74	Delete
3	Soccer Ball	Soccer	23.4	Delete
4	Corner Flags	Soccer	41.940000000000005	Delete
5	Thinking Cap	Chess	19.2	Delete

Figure 17.4. Using a custom pipe

17.3.3 Combining pipes

The `addTax` pipe is applying the tax rate, but the fractional amounts that are produced by the calculation are unsightly—and unhelpful since few tax authorities insist on accuracy to 15 fractional digits.

I could fix this by adding support to the custom pipe to format the number values as currencies, but that would require duplicating the functionality of the built-in `currency` pipe that I used earlier in the chapter. A better approach is to combine the functionality of both pipes so that the output from the custom `addTax` pipe is fed into the built-in `currency` pipe, which is then used to produce the value displayed to the user.

Pipes are chained together in this way using the vertical bar character, and the names of the pipes are specified in the order that data should flow, as shown in listing 17.9.

Listing 17.9. Combining pipes in the `productTable.component.html` file in the `src/app` folder

```
...
<td>{{item.price | addTax:(taxRate || 0) | currency:"USD":"symbol" }}</td>
...
```

The value of the `item.price` property is passed to the `addTax` pipe, which adds the sales tax, and then to the `currency` pipe, which formats the number value into a currency amount, as shown in figure 17.5.

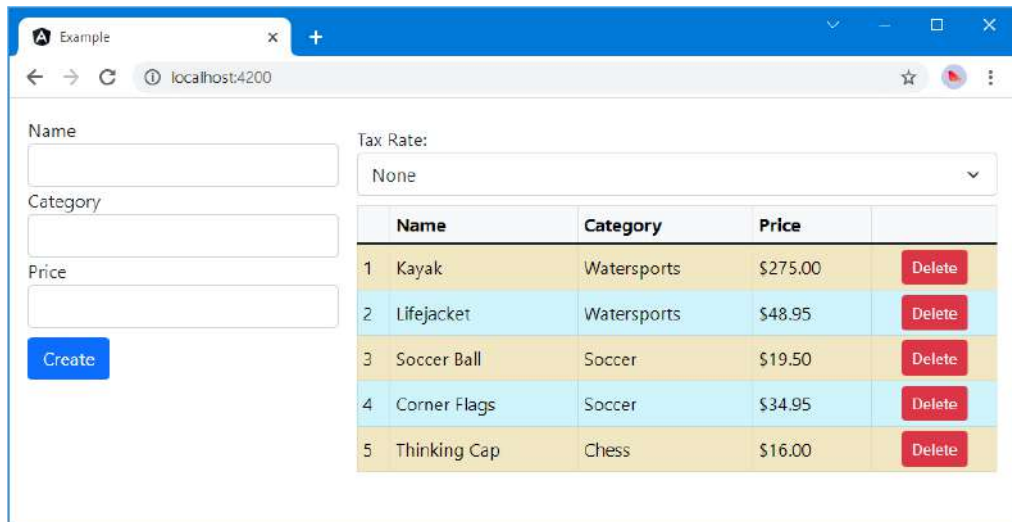


Figure 17.5. Combining the functionality of pipes

17.3.4 Creating impure pipes

The `pure` decorator property is used to tell Angular when to call the pipe's `transform` method. The default value for the `pure` property is `true`, which tells Angular that the pipe's `transform` method will generate a new value only if the input data value—the data value before the vertical bar character in the template—changes or when one or more of its arguments is modified. This is known as a *pure* pipe because it has no independent internal state and all its dependencies can be managed using the Angular change detection process.

Setting the `pure` decorator property to `false` creates an *impure pipe* and tells Angular that the pipe has its own state data or that it depends on data that may not be picked up in the change detection process when there is a new value.

When Angular performs its change detection process, it treats impure pipes as sources of data values in their own right and invokes the `transform` methods even when there has been no data value or argument changes.

The most common need for impure pipes is when they process the contents of arrays and the elements in the array change. Angular doesn't automatically detect changes that occur within arrays and won't invoke a pure pipe's `transform` method when an array element is added, edited, or deleted because it just sees the same array object being used as the input data value.

CAUTION Impure pipes should be used sparingly because Angular has to call the `transform` method whenever there is any data change or user interaction in the application, just in case it might result in a different result from the pipe. If you do create

an impure pipe, then keep it as simple as possible. Performing complex operations, such as sorting an array, can devastate the performance of an Angular application.

As a demonstration, I added a file called `categoryFilter.pipe.ts` in the `src/app` folder and used it to define the pipe shown in listing 17.10.

Listing 17.10. The contents of the `categoryFilter.pipe.ts` file in the `src/app` folder

```
import { Pipe } from "@angular/core";
import { Product } from "../product.model";

@Pipe({
  name: "filter",
  pure: true
})
export class PaCategoryFilterPipe {
  transform(products: Product[] | undefined,
    category: string | undefined): Product[] {
    if (products == undefined) {
      return [];
    }
    return category == undefined ?
      products : products.filter(p => p.category == category);
  }
}
```

This is a pure filter that receives an array of `Product` objects and returns only the ones whose `category` property matches the `category` argument. Listing 17.11 shows the new pipe registered in the Angular module.

Listing 17.11. Registering a pipe in the `app.module.ts` file in the `src/app` folder

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

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

import { ProductComponent } from './component';
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';
import { ProductTableComponent } from './productTable.component';
import { ProductFormComponent } from './productForm.component';
import { PaToggleView } from './toggleView.component';
import { PaAddTaxPipe } from './addTax.pipe';
import { PaCategoryFilterPipe } from './categoryFilter.pipe';

@NgModule({
  imports: [
    BrowserModule,
    FormsModule,
    BrowserAnimationsModule,
    ProductComponent,
    PaAttrDirective,
    PaModel,
    PaStructureDirective,
    PaIteratorDirective,
    PaCellColor,
    PaCellColorSwitcher,
    ProductTableComponent,
    ProductFormComponent,
    PaToggleView,
    PaAddTaxPipe,
    PaCategoryFilterPipe
  ],
  declarations: [
    AppComponent
  ],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

```

    declarations: [
      ProductComponent, PaAttrDirective, PaModel, PaStructureDirective,
      PaIteratorDirective, PaCellColor, PaCellColorSwitcher,
      ProductTableComponent, ProductFormComponent, PaToggleView,
      PaAddTaxPipe, PaCategoryFilterPipe
    ],
    imports: [
      BrowserModule,
      BrowserAnimationsModule,
      FormsModule
    ],
    providers: [],
    bootstrap: [ProductComponent]
  })
  export class AppModule { }

```

Listing 17.12 shows the application of the new pipe to the binding expression that targets the `ngFor` directive as well as a new `select` element that allows the filter category to be selected.

Listing 17.12. Applying a pipe in the `productTable.component.html` file in the `src/app` folder

```

<div class="my-2">
  <label>Tax Rate:</label>
  <select class="form-select" [value]="taxRate || 0"
    (change)="taxRate=$any($event).target.value">
    <option value="0">None</option>
    <option value="10">10%</option>
    <option value="20">20%</option>
    <option value="50">50%</option>
  </select>
</div>

<div class="my-2">
  <label>Category Filter:</label>
  <select class="form-select" [(ngModel)]="categoryFilter">
    <option>Watersports</option>
    <option>Soccer</option>
    <option>Chess</option>
  </select>
</div>

<table class="table table-sm table-bordered table-striped">
  <thead class="table-light">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th>
    <th></th></tr>
  </thead>
  <tbody>
    <tr *paFor="let item of Products() | filter:categoryFilter;"
      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 | addTax:(taxRate || 0)
    | currency:"USD":"symbol" }}
</td>
<td class="text-center">
  <button class="btn btn-danger btn-sm"
    (click)="deleteProduct(item.id) ">
    Delete
  </button>
</td>
</tr>
</tbody>
</table>

```

To see the problem, use the `select` element to filter the products in the table so that only those in the `Soccer` category are shown. Then use the form elements to create a new product in that category. Clicking the `Create` button will add the product to the data model, but the new product won't be shown in the table, as illustrated in figure 17.6.

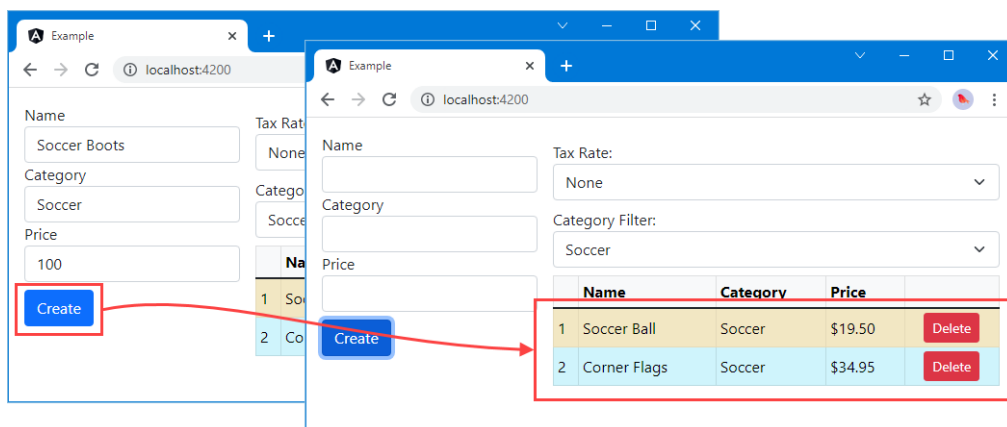


Figure 17.6. A problem caused by a pure pipe

The table isn't updated because, as far as Angular is concerned, none of the inputs to the filter pipe has changed. The component's `getProducts` method returns the same array object, and the `categoryFilter` property is still set to `Soccer`. The fact that there is a new object inside the array returned by the `getProducts` method isn't recognized by Angular.

The solution is to set the pipe's `pure` property to `false`, as shown in listing 17.13.

Listing 17.13. An impure pipe in the `categoryFilter.pipe.ts` file in the `src/app` folder

```

import { Pipe } from "@angular/core";
import { Product } from "../product.model";

@Pipe({
  name: "filter",
  pure: false
})

```

```
    })
    export class PaCategoryFilterPipe {

      transform(products: Product[] | undefined,
                category: string | undefined): Product[] {
        if (products == undefined) {
          return [];
        }
        return category == undefined ?
          products : products.filter(p => p.category == category);
      }
    }
  }
```

If you repeat the test, you will see that the new product is now correctly displayed in the table, as shown in figure 17.7.

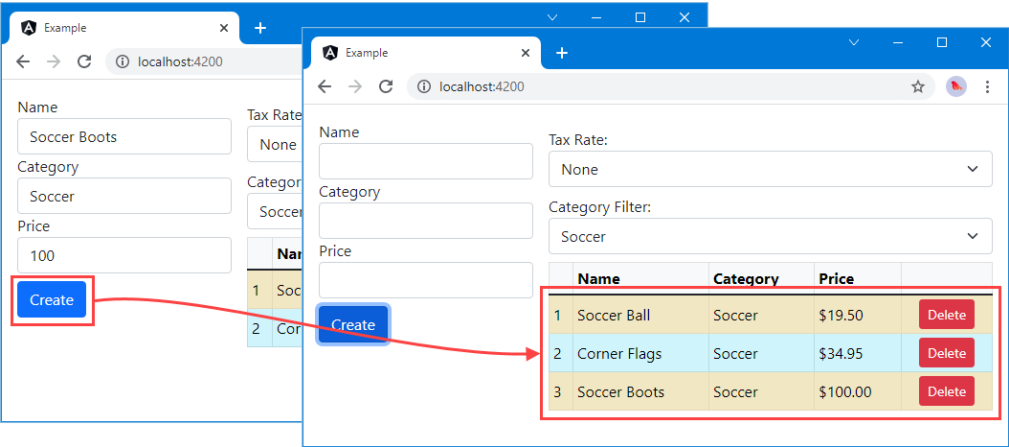


Figure 17.7. Using an impure pipe

17.4 Using the built-in pipes

Angular includes a set of built-in pipes that perform commonly required tasks. These pipes are described in table 17.4 and demonstrated in the sections that follow.

Table 17.4. The built-in pipes

Name	Description
number	This pipe performs location-sensitive formatting of number values. See the “Formatting numbers” section for details.
currency	This pipe performs location-sensitive formatting of currency amounts. See the “Formatting currency values” section for details.

percent	This pipe performs location-sensitive formatting of percentage values. See the “Formatting percentages” section for details.
date	This pipe performs location-sensitive formatting of dates. See the “Formatting dates” section for details.
uppercase	This pipe transforms all the characters in a string to uppercase. See the “Changing string case” section for details.
Lowercase	This pipe transforms all the characters in a string to lowercase. See the “Changing string case” section for details.
titlecase	This pipe transforms all the characters in a string to title case. See the “Changing string case” section for details.
json	This pipe transforms an object into a JSON string. See the “Serializing data as JSON” section for details.
slice	This pipe selects items from an array or characters from a string, as described in the “Slicing data arrays” section.
keyvalue	This pipe transforms an object or map into a series of key-value pairs, as described in the “Formatting key-value pairs” section.
i18nSelect	This pipe selects a text value to display for a set of values, as described in the “Selecting values” section.
i18nPlural	This pipe selects a pluralized string for a value, as described in the “Pluralizing values” section.
async	This pipe subscribes to an observable or a promise and displays the most recent value it produces.

17.4.1 Formatting numbers

The `number` pipe formats `number` values using locale-sensitive rules. Listing 17.14 shows the use of the `number` pipe, along with the argument that specifies the formatting that will be used. I have removed the custom pipes and the associated `select` elements from the template.

Listing 17.14. Using the `number` pipe in the `productTable.component.html` file in the `src/app` folder

```

<!-- <div class="my-2">
  <label>Tax Rate:</label>
  <select class="form-select" [value]="taxRate || 0"
    (change)="taxRate=$any($event).target.value">
    <option value="0">None</option>
    <option value="10">10%</option>
    <option value="20">20%</option>
    <option value="50">50%</option>
  </select>
</div>

```

```

    </select>
  </div>

  <div class="my-2">
    <label>Category Filter:</label>
    <select class="form-select" [(ngModel)]="categoryFilter">
      <option>Watersports</option>
      <option>Soccer</option>
      <option>Chess</option>
    </select>
  </div> -->

  <table class="table table-sm table-bordered table-striped">
    <thead class="table-light">
      <tr><th></th><th>Name</th><th>Category</th><th>Price</th>
      <th></th></tr>
    </thead>
    <tbody>
      <tr *paFor="let item of Products() | filter:categoryFilter;
        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 | number:"3.2-2" }}</td>
        <td class="text-center">
          <button class="btn btn-danger btn-sm"
            (click)="deleteProduct(item.id)">
            Delete
          </button>
        </td>
      </tr>
    </tbody>
  </table>

```

The number pipe accepts a single argument that specifies the number of digits that are included in the formatted result. The argument is in the following format (note the period and hyphen that separate the values and that the entire argument is quoted as a string):

```
"<minIntegerDigits>.<minFactionDigits>-<maxFractionDigits>"
```

Table 17.5 describes each element of the formatting argument.

Table 17.5. The elements of the number pipe argument

Name	Description
minIntegerDigits	This value specifies the minimum number of digits. The default value is 1.
minFractionDigits	This value specifies the minimum number of fractional digits. The default value is 0.

maxFractionDigits	This value specifies the maximum number of fractional digits. The default value is 3.
-------------------	---

The argument used in the listing is "3.2-2", which specifies that at least three digits should be used to display the integer portion of the number and that two fractional digits should always be used. This produces the result shown in figure 17.8.

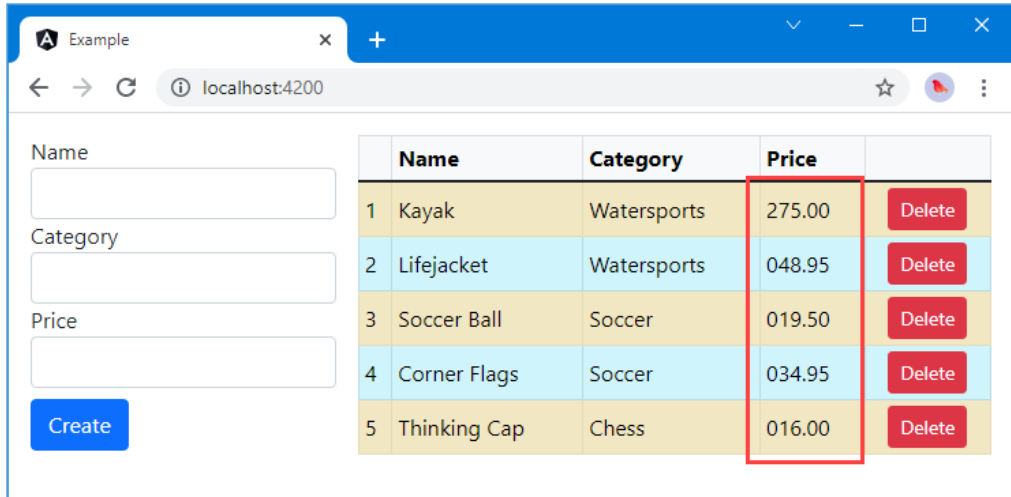


Figure 17.8. Formatting number values

The `number` pipe is location-sensitive, which means that the same format argument will produce differently formatted results based on the user's locale setting. Angular applications default to the `en-US` locale by default and require other locales to be loaded explicitly, as shown in listing 17.15.

Listing 17.15. Setting the locale 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";
```

```

import { ProductTableComponent } from "../productTable.component";
import { ProductFormComponent } from "../productForm.component";
import { PaToggleView } from "../toggleView.component";
import { PaAddTaxPipe } from "../addTax.pipe";
import { PaCategoryFilterPipe } from "../categoryFilter.pipe";

import { LOCALE_ID } from "@angular/core";
import localeFr from '@angular/common/locales/fr';
import { registerLocaleData } from '@angular/common';

registerLocaleData(localeFr);

@NgModule({
  declarations: [ProductComponent, PaAttrDirective, PaModel,
    PaStructureDirective, PaIteratorDirective,
    PaCellColor, PaCellColorSwitcher, ProductTableComponent,
    ProductFormComponent, PaToggleView, PaAddTaxPipe,
    PaCategoryFilterPipe],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule, ReactiveFormsModule
  ],
  providers: [{ provide: LOCALE_ID, useValue: "fr-FR" }],
  bootstrap: [ProductComponent]
})
export class AppModule { }

```

Setting the locale consists of importing the locale you require from the modules that contain each region's data and registering it by calling the `registerLocaleData` function, which is imported from the `@angular/common` module. In the listing, I have imported the `fr-FR` locale, which is for French as it is spoken in France. The final step is to configure the `providers` property, which I describe in chapter 18, but the effect of the configuration in listing 17.15 is to enable the `fr-FR` locale, which changes the formatting of the numerical values, as shown in figure 17.9.

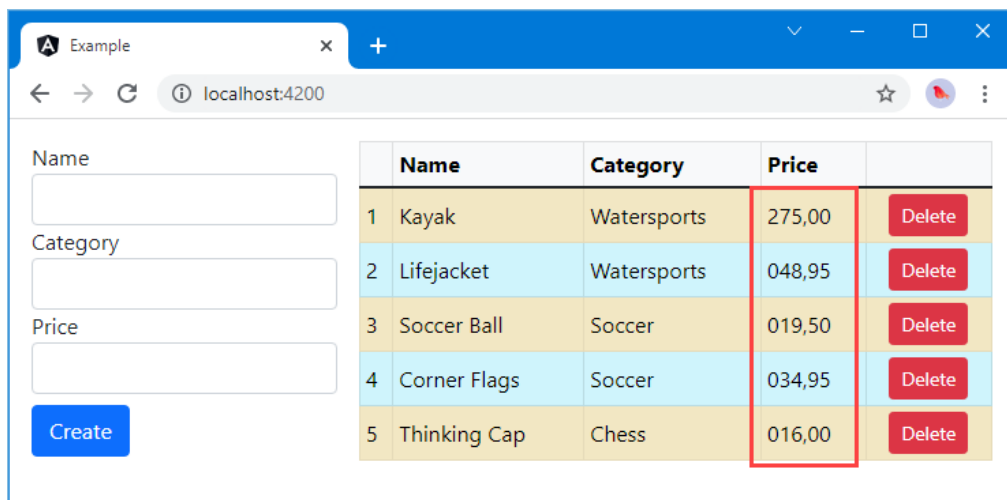


Figure 17.9. Locale-sensitive formatting

You can override the application's locale setting by specifying a locale as a configuration option for the pipe, like this:

```
...
<td>{{item.price | number:"3.2-2": "en-US" }}</td>
...
```

17.4.2 Formatting currency values

The currency pipe formats number values that represent monetary amounts. Listing 17.5 used this pipe to introduce the topic, and listing 17.16 shows another application of the same pipe but with the addition of number format specifiers.

Listing 17.16. Using the currency pipe in the productTable.component.html file in the src/app folder

```
<table class="table table-sm table-bordered table-striped">
  <thead class="table-light">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th>
    <th></th></tr>
  </thead>
  <tbody>
    <tr *paFor="let item of Products() | filter:categoryFilter;
      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 | currency:"USD": "symbol": "2.2-2" }}</td>
```

```

        <td class="text-center">
            <button class="btn btn-danger btn-sm"
                (click)="deleteProduct(item.id) ">
                Delete
            </button>
        </td>
    </tr>
</tbody>
</table>

```

The currency pipe can be configured using four arguments, which are described in table 17.6.

Table 17.6. The arguments for the currency pipe

Name	Description
currencyCode	This string argument specifies the currency using an ISO 4217 code. The default value is <code>USD</code> if this argument is omitted. You can see a list of currency codes at http://en.wikipedia.org/wiki/ISO_4217 .
display	This string indicates whether the currency symbol or code should be displayed. The supported values are <code>code</code> (use the currency code), <code>symbol</code> (use the currency symbol), and <code>symbol-narrow</code> (which shows the concise form when a currency has narrow and wide symbols). You can also specify a string to use. The default value is <code>symbol</code> .
digitInfo	This string argument specifies the formatting for the number, using the same formatting instructions supported by the <code>number</code> pipe, as described in the “Formatting Numbers” section.
locale	This string argument specifies the locale for the currency. This defaults to the <code>LOCALE_ID</code> value, the configuration of which is shown in listing 17.15.

The arguments specified in listing 17.16 tell the pipe to use the U.S. dollar as the currency (which has the ISO code `USD`), to display the symbol rather than the code in the output, and to format the number so that it has at least two integer digits and exactly two fraction digits.

This pipe relies on the Internationalization API to get details of the currency—especially its symbol—but doesn’t select the currency automatically to reflect the user’s locale setting.

This means that the formatting of the number and the position of the currency symbol are affected by the application’s locale setting, regardless of the currency that has been specified by the pipe. The example application is still configured to use the `fr-FR` locale, which produces the results shown in figure 17.10.

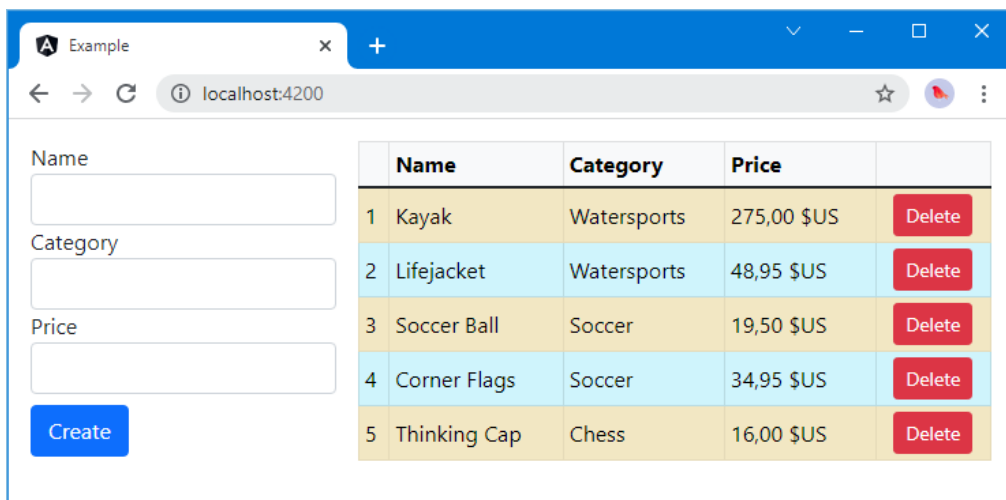


Figure 17.10. Location-sensitive currency formatting

To revert to the default locale, listing 17.17 removes the `fr-FR` setting from the application's root module.

Listing 17.17. Removing the locale setting in the `app.module.ts` file in the `src/app` folder

```
...
@NgModule({
  declarations: [
    ProductComponent, PaAttrDirective, PaModel, PaStructureDirective,
    PaIteratorDirective, PaCellColor, PaCellColorSwitcher,
    ProductTableComponent, ProductFormComponent, PaToggleView,
    PaAddTaxPipe, PaCategoryFilterPipe
  ],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule
  ],
  //providers: [{ provide: LOCALE_ID, useValue: "fr-FR" }],
  bootstrap: [ProductComponent]
})
export class AppModule { }
...
```

Figure 17.11 shows the result.

	Name	Category	Price	
1	Kayak	Watersports	\$275.00	Delete
2	Lifejacket	Watersports	\$48.95	Delete
3	Soccer Ball	Soccer	\$19.50	Delete
4	Corner Flags	Soccer	\$34.95	Delete
5	Thinking Cap	Chess	\$16.00	Delete

Figure 17.11. Formatting currency values

17.4.3 Formatting percentages

The `percent` pipe formats `number` values as percentages, where values between 0 and 1 are formatted to represent 0 to 100 percent. This pipe has optional arguments that are used to specify the number formatting options, using the same format as the `number` pipe, and override the default locale. Listing 17.18 re-introduces the custom sales tax filter and populates the associated `select` element with `option` elements whose content is formatted with the `percent` filter.

Listing 17.18. Formatting percentages in the `productTable.component.html` file in the `src/app` folder

```
<div class="my-2">
  <label>Tax Rate:</label>
  <select class="form-select" [value]="taxRate || 0"
    (change)="taxRate=$any($event).target.value">
    <option value="0">None</option>
    <option value="10">{{ 0.1 | percent }}</option>
    <option value="20">{{ 0.2 | percent }}</option>
    <option value="50">{{ 0.5 | percent }}</option>
    <option value="150">{{ 1.5 | percent }}</option>
  </select>
</div>

<table class="table table-sm table-bordered table-striped">
  <thead class="table-light">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th>
      <th></th></tr>
  </thead>
  <tbody>
    <tr *paFor="let item of Products() | filter:categoryFilter;"
```

```

        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 | addTax:(taxRate || 0)
    | currency:"USD": "symbol": "2.2-2" }}
</td>
<td class="text-center">
    <button class="btn btn-danger btn-sm"
        (click)="deleteProduct(item.id)">
        Delete
    </button>
</td>
</tr>
</tbody>
</table>

```

Values that are greater than 1 are formatted into percentages greater than 100 percent. You can see this in the last item shown in figure 17.12, where the value 1.5 produces a formatted value of 150 percent.

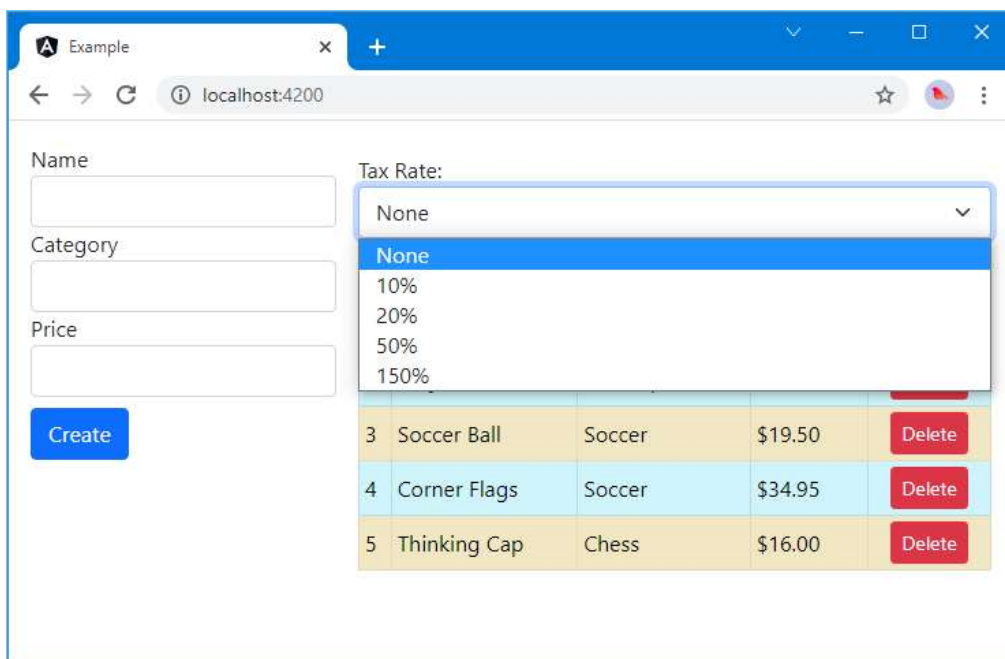


Figure 17.12. Formatting percentage values

The formatting of percentage values is location-sensitive, although the differences between locales can be subtle. As an example, while the `en-US` locale produces a result such as 10 percent, with the numerals and the percent sign next to one another, many locales, including `fr-FR`, will produce a result such as 10 %, with a space between the numerals and the percent sign.

17.4.4 Formatting dates

The `date` pipe performs location-sensitive formatting of dates. Dates can be expressed using JavaScript `Date` objects, as a `number` value representing milliseconds since the beginning of 1970 or as a well-formatted string. Listing 17.19 adds three properties to the `ProductTableComponent` class, each of which encodes a date in one of the formats supported by the `date` pipe.

Listing 17.19. Defining dates in the `productTable.component.ts` file in the `src/app` folder

```
import { Component, Input, Signal, QueryList, ViewChildren,
        ChangeDetectorRef } from "@angular/core";
import { Model } from "../repository.model";
import { Product } from "../product.model";

@Component({
  selector: "paProductTable",
  templateUrl: "productTable.component.html"
})
export class ProductTableComponent {

  constructor(private changeRef: ChangeDetectorRef) {}

  @Input({ alias: "model", required: true })
  dataModel!: Model;

  get Products(): Signal<Product[]> {
    return this.dataModel.Products;
  }

  getProduct(key: number): Product | undefined {
    return this.dataModel?.getProduct(key);
  }

  deleteProduct(key: number) {
    this.dataModel.deleteProduct(key);
  }

  taxRate: number = 0;
  categoryFilter: string | undefined;
  itemCount: number = 3;

  dateObject: Date = new Date(2020, 1, 20);
  dateString: string = "2020-02-20T00:00:00.000Z";
  dateNumber: number = 1582156800000;
}
```


All three properties describe the same date, which is February 20, 2020. No time has been specified, which means that these values will represent midnight, with no time specified. In listing 17.20, I have used the `date` pipe to format all three properties.

Listing 17.20. Formatting dates in the `productTable.component.html` file in the `src/app` folder

```
<div class="bg-info p-2 text-white">
  <div>Date formatted from object: {{ dateObject | date }}</div>
  <div>Date formatted from string: {{ dateString | date }}</div>
  <div>Date formatted from number: {{ dateNumber | date }}</div>
</div>

<table class="table table-sm table-bordered table-striped">
  <thead class="table-light">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th>
      <th></th></tr>
  </thead>
  <tbody>
    <tr *paFor="let item of Products() | filter:categoryFilter;
      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 | addTax: (taxRate || 0)
        | currency:"USD": "symbol": "2.2-2" }}
      </td>
      <td class="text-center">
        <button class="btn btn-danger btn-sm"
          (click)="deleteProduct(item.id)">
          Delete
        </button>
      </td>
    </tr>
  </tbody>
</table>
```

The pipe works out which data type it is working with, parses the value to get a date, and then formats it, as shown in figure 17.13.

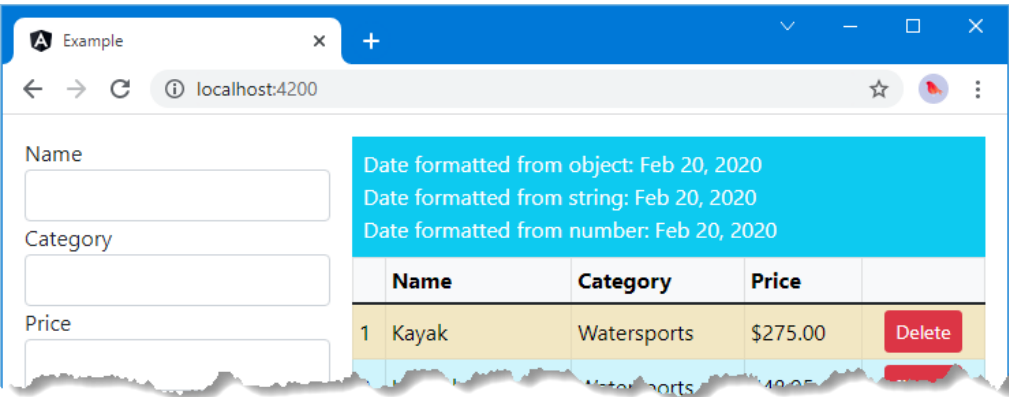


Figure 17.13. Formatting dates

If you are in a time zone that is to the left of GMT, then you will see Feb 19, 2020, for two of the dates. The first date is expressed relative to the application’s time zone, but the others are expressed in the UTC time zone, which means that the dates will be adjusted, as shown in figure 17.14.

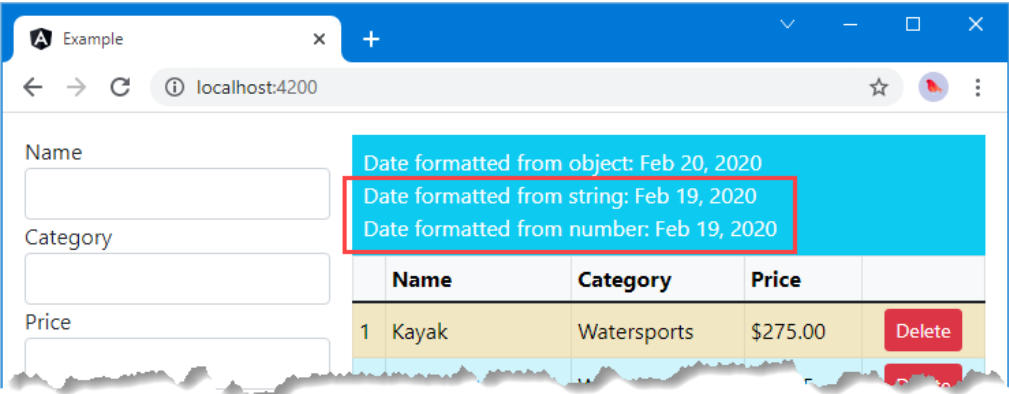


Figure 17.14. The effect of a different time zone

The `date` pipe accepts an argument that specifies the date format that should be used. Individual date components can be selected for the output using the symbols described in table 17.7. A complete set of supported symbols can be found in the Angular API documentation at <https://angular.io/api/common/DatePipe>.

Table 17.7. Useful date pipe format symbols

Name	Description
Y, YY, YYYY	These symbols select the year.
M, MMM, MMMM	These symbols select the month.
d, dd	These symbols select the day (as a number).
E, EE, EEE, EEEE, EEEEE	These symbols select the day (as a name).
h, hh, H, HH	These symbols select the hour in 12- and 24-hour forms.
m, mm	These symbols select the minutes.
s, ss	These symbols select the seconds.
Z	This symbol selects the time zone.

The symbols in table 17.7 provide access to the date components in differing levels of brevity so that `M` will return `2` if the month is February, `MM` will return `02`, `MMM` will return `Feb`, and `MMMM` will return `February`, assuming that you are using the `en-US` locale. The date pipe also supports predefined date formats for commonly used combinations, the most useful of which are described in table 17.8.

Table 17.8. Useful predefined date pipe formats

Name	Description
<code>short</code>	This format is equivalent to the component string <code>M/d/yy, h:mm a</code> . It presents the date in a concise format, including the time component.
<code>medium</code>	This format is equivalent to the component string <code>MMM d, y, h:mm:ss a</code> . It presents the date as a more expansive format, including the time component.
<code>shortDate</code>	This format is equivalent to the component string <code>M/d/yy</code> . It presents the date in a concise format and excludes the time component.
<code>mediumDate</code>	This format is equivalent to the component string <code>MMM d, y</code> . It presents the date in a more expansive format and excludes the time component.
<code>longDate</code>	This format is equivalent to the component string <code>MMMM d, y</code> . It presents the date and excludes the time component.
<code>fullDate</code>	This format is equivalent to the component string <code>EEEE, MMMM d, y</code> . It presents the date fully and excludes the date format.
<code>shortTime</code>	This format is equivalent to the component string <code>h:mm a</code> .
<code>mediumTime</code>	This format is equivalent to the component string <code>h:mm:ss a</code> .

The `date` pipe also accepts arguments that specify a time zone and a locale. Listing 17.21 shows the use of the predefined formats as arguments to the `date` pipe, rendering the same date in different ways and with different locale settings.

TIP The time zone argument has to be specified in order to set the locale. Use the empty string ("") as the time zone if you want to use the application's default time zone.

Listing 17.21. Formatting dates in the `productTable.component.html` file in the `src/app` folder

```
<div class="bg-info p-2 text-white">
  <div>
    Date formatted from object: {{ dateObject | date:"shortDate" }}
  </div>
  <div>
    Date formatted from string: {{ dateString | date:"mediumDate" }}
  </div>
  <div>
    Date formatted from number: {{ dateNumber | date:"longDate" }}
  </div>
</div>

<div class="bg-info p-2 text-white">
  <div>
    Date formatted from object:
      {{ dateObject | date:"shortDate":"UTC":"fr-FR" }}
  </div>
  <div>
    Date formatted from string:
      {{ dateString | date:"mediumDate":"UTC":"fr-FR" }}
  </div>
  <div>
    Date formatted from number:
      {{ dateNumber | date:"longDate":"UTC":"fr-FR" }}
  </div>
</div>

<table class="table table-sm table-bordered table-striped">
  <thead class="table-light">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th>
    <th></th></tr>
  </thead>
  <tbody>
    <tr *paFor="let item of Products() | filter:categoryFilter;
      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 | addTax:(taxRate || 0)
        | currency:"USD":"symbol":"2.2-2" }}
      </td>
```

```
 <button class="btn btn-danger btn-sm"         (click)="deleteProduct(item.id)">         Delete       </button>     </td>   </tr> </tbody> </table> |
```

Formatting arguments are specified as literal strings. Take care to capitalize the format string correctly because `shortDate` will be interpreted as one of the predefined formats from table 17.8, but `shortdate` (with a lowercase letter `d`) will be interpreted as a series of characters from table 17.7 and produce nonsensical output.

CAUTION Date parsing/formatting is a complex and time-consuming process. As a consequence, the `pure` property for the date pipe is `true`; as a result, changes to individual components of a `Date` object won't trigger an update. If you need to reflect changes in the way that a date is displayed, then you must change the reference to the `Date` object that the binding containing the date pipe refers to.

Figure 17.15 shows the formatted dates, in the `en-US` and `fr-FR` locales.

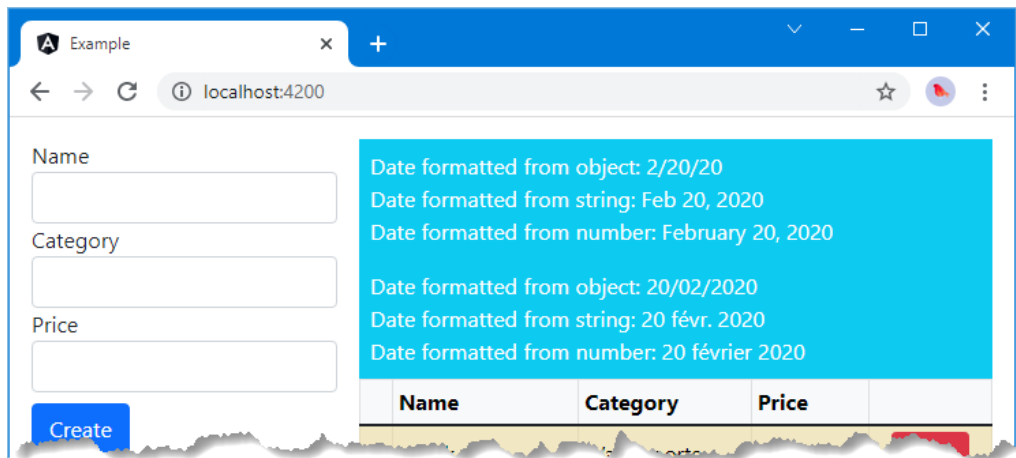


Figure 17.15. Location-sensitive date formatting

Understanding the impact of lazy localization

Localizing a product takes time, effort, and resources, and it needs to be done by someone who understands the linguistic, cultural, and monetary conventions of the target country or region. If you don't localize properly, then the result can be worse than not localizing at all.

It is for this reason that I don't describe localization features in detail in this book—or any of my books. Describing features outside of the context in which they will be used feels like setting up readers for a self-inflicted disaster. At least if a product isn't localized, the user knows where they stand and doesn't have to try to figure out whether you just forgot to change the currency code or whether those prices are really in U.S. dollars. (This is an issue that I see all the time living in the United Kingdom.)

You *should* localize your products. Your users *should* be able to do business or perform other operations in a way that makes sense to them. But you *must* take it seriously and allocate the time and effort required to do it properly.

17.4.5 Changing string case

The `uppercase`, `lowercase`, and `titlecase` pipes convert all the characters in a string to uppercase or lowercase, respectively. Listing 17.22 shows the first two pipes applied to cells in the product table. This listing also removes the dates used in the previous section.

Listing 17.22. Changing character case in the `productTable.component.html` file in the `src/app` folder

```
<!-- <div class="bg-info p-2 text-white">
  <div>
    Date formatted from object: {{ dateObject | date:"shortDate" }}
  </div>
  <div>
    Date formatted from string: {{ dateString | date:"mediumDate" }}
  </div>
  <div>
    Date formatted from number: {{ dateNumber | date:"longDate" }}
  </div>
</div>

<div class="bg-info p-2 text-white">
  <div>
    Date formatted from object:
      {{ dateObject | date:"shortDate":"UTC":"fr-FR" }}
  </div>
  <div>
    Date formatted from string:
      {{ dateString | date:"mediumDate":"UTC":"fr-FR" }}
  </div>
  <div>
    Date formatted from number:
      {{ dateNumber | date:"longDate":"UTC":"fr-FR" }}
  </div>
</div> -->

<table class="table table-sm table-bordered table-striped">
  <thead class="table-light">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th>
      <th></th></tr>
  </thead>
```

```

<tbody>
  <tr *paFor="let item of Products() | filter:categoryFilter;
    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 | uppercase }}</td>
    <td>{{item.category | lowercase }}</td>
    <td>{{item.price | addTax:(taxRate || 0)
      | currency:"USD":"symbol":"2.2-2" }}
    </td>
    <td class="text-center">
      <button class="btn btn-danger btn-sm"
        (click)="deleteProduct(item.id)">
        Delete
      </button>
    </td>
  </tr>
</tbody>
</table>

```

These pipes use the standard JavaScript string methods `toUpperCase` and `toLowerCase`, which are not sensitive to locale settings, as shown in figure 17.16.

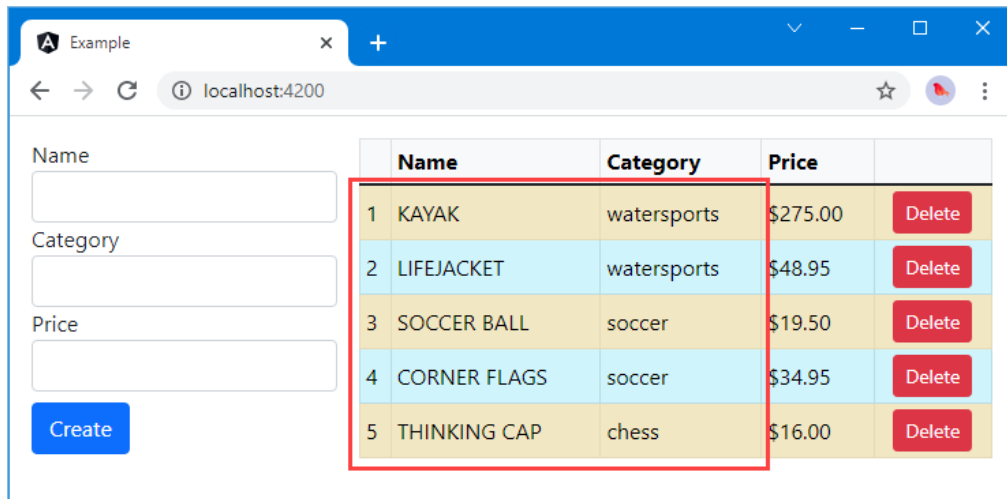


Figure 17.16. Changing character case

The `titlecase` pipe capitalizes the first character of each word and uses lowercase for the remaining characters. Listing 17.23 applies the `titlecase` pipe to the table cells.

Listing 17.23. Applying the pipe in the productTable.component.html file in the src/app folder

```

<table class="table table-sm table-bordered table-striped">
  <thead class="table-light">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th>
    <th></th></tr>
  </thead>
  <tbody>
    <tr *paFor="let item of Products() | filter:categoryFilter;
      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 | titlecase }}</td>
      <td>{{item.category | lowercase }}</td>
      <td>{{item.price | addTax:(taxRate || 0)
        | currency:"USD":"symbol":"2.2-2" }}
      </td>
      <td class="text-center">
        <button class="btn btn-danger btn-sm"
          (click)="deleteProduct(item.id)">
          Delete
        </button>
      </td>
    </tr>
  </tbody>
</table>

```

Figure 17.17 shows the effect of the pipe.

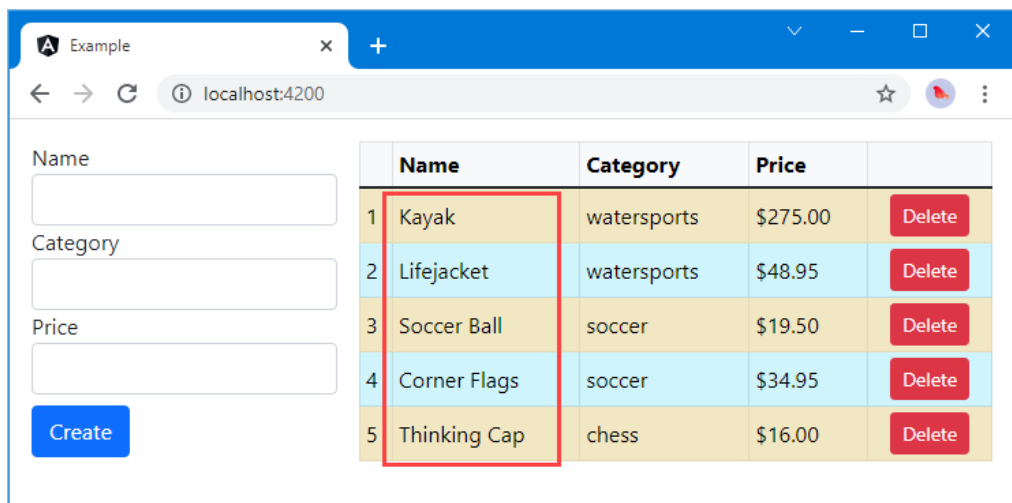


Figure 17.17. Using the titlecase pipe

17.4.6 Serializing data as JSON

The `json` pipe creates a JSON representation of a data value. No arguments are accepted by this pipe, which uses the browser's `JSON.stringify` method to create the JSON string. Listing 17.24 applies this pipe to create a JSON representation of the objects in the data model.

Listing 17.24. Creating a JSON string in the `productTable.component.html` file in the `src/app` folder

```
<div class="bg-info p-2 text-white">
  <div>{{ Products() | json }}</div>
</div>

<table class="table table-sm table-bordered table-striped">
  <thead class="table-light">
    <tr><th></th><th>Name</th><th>Category</th><th>Price</th>
      <th></th></tr>
  </thead>
  <tbody>
    <tr *paFor="let item of Products() | filter:categoryFilter;
      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 | titlecase }}</td>
      <td>{{ item.category | lowercase }}</td>
      <td>{{ item.price | addTax:(taxRate || 0)
        | currency:"USD":"symbol":"2.2-2" }}
      </td>
      <td class="text-center">
        <button class="btn btn-danger btn-sm"
          (click)="deleteProduct(item.id) ">
          Delete
        </button>
      </td>
    </tr>
  </tbody>
</table>
```

This pipe is useful during debugging, and its decorator's `pure` property is `false` so that any change in the application will cause the pipe's `transform` method to be invoked, ensuring that even collection-level changes are shown. Figure 17.18 shows the JSON generated from the objects in the example application's data model.

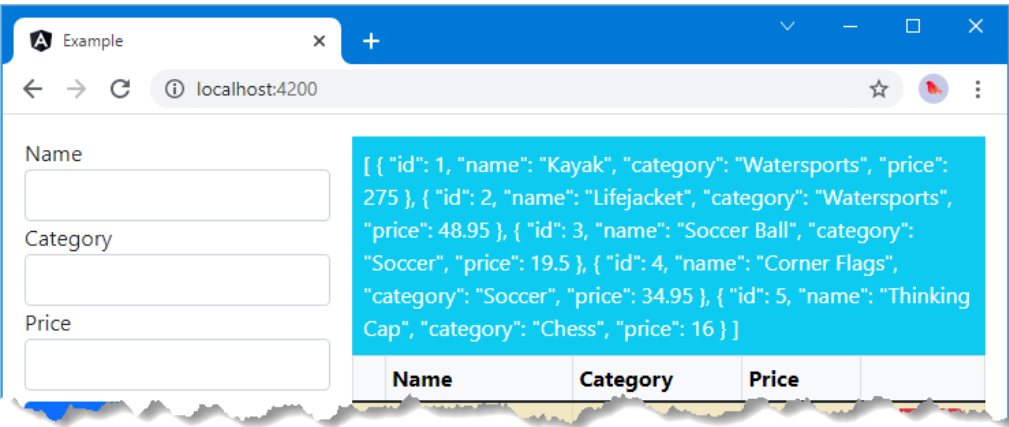


Figure 17.18. Generating JSON strings for debugging

17.4.7 Slicing data arrays

The `slice` pipe operates on an array or string and returns a subset of the elements or characters it contains. This is an impure pipe, which means it will reflect any changes that occur within the data object it is operating on but also means that the slice operation will be performed after any change in the application, even if that change was not related to the source data.

The objects or characters selected by the `slice` pipe are specified using two arguments, which are described in table 17.9.

Table 17.9. The slice pipe arguments

Name	Description
start	This argument must be specified. If the value is positive, the start index for items to be included in the result counts from the first position in the array. If the value is negative, then the pipe counts back from the end of the array.
end	This optional argument is used to specify how many items from the <code>start</code> index should be included in the result. If this value is omitted, all the items after the <code>start</code> index (or before in the case of negative values) will be included.

Listing 17.25 demonstrates the use of the `slice` pipe in combination with a `select` element that specifies how many items should be displayed in the product table.

Listing 17.25. Using the slice pipe in the productTable.component.html file in the src/app folder

```
<div class="form-group my-2">
  <label>Number of items:</label>
  <select class="form-select" [value]="itemCount || 1"
```

```

        (change)="itemCount=$any($event).target.value">
        <option *ngFor="let item of Products(); let i = index"
            [value]="i + 1" [selected]="(i + 1) === itemCount">
            {{i + 1}}
        </option>
    </select>
</div>

<table class="table table-sm table-bordered table-striped">
    <thead class="table-light">
        <tr><th></th><th>Name</th><th>Category</th><th>Price</th>
        <th></th></tr>
    </thead>
    <tbody>
        <tr *paFor="let item of Products() | slice:0:(itemCount ?? 1);
            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 | titlecase }}</td>
            <td>{{item.category | lowercase }}</td>
            <td>{{item.price | addTax:(taxRate || 0)
                | currency:"USD":"symbol":"2.2-2" }}
            </td>
            <td class="text-center">
                <button class="btn btn-danger btn-sm"
                    (click)="deleteProduct(item.id)">
                    Delete
                </button>
            </td>
        </tr>
    </tbody>
</table>

```

The `select` element is populated with `option` elements created with the `ngFor` directive. This directive doesn't directly support iterating a specific number of times, so I have used the `index` variable to generate the values that are required. The `select` element sets a property called `itemCount`, which is used as the second argument of the `slice` pipe, like this:

```

...
<tr *paFor="let item of getProducts() | slice:0:(itemCount ?? 1);
    let i = index; let odd = odd;
    let even = even" [class.table-info]="odd" [class.table-warning]="even"
    class="align-middle">
...

```

The effect is that changing the value displayed by the `select` element changes the number of items displayed in the product table, as shown in figure 17.19.

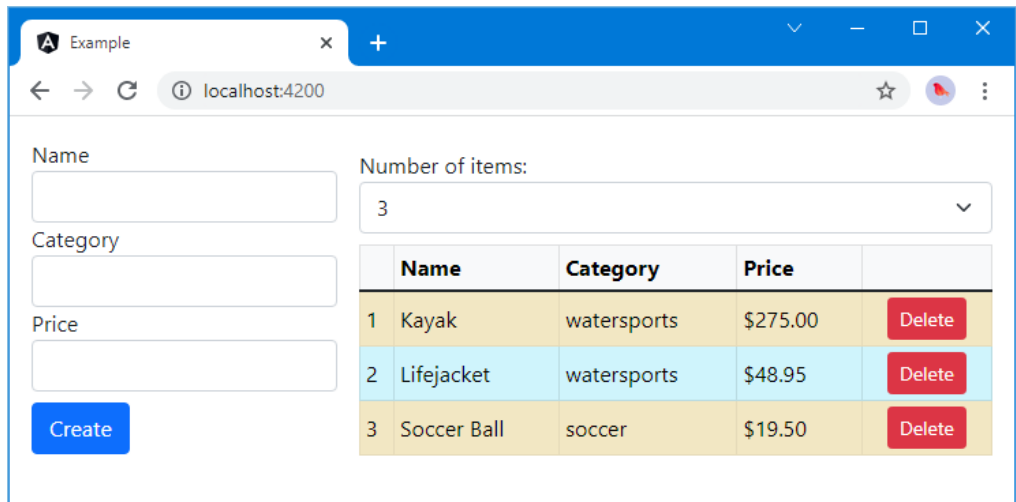


Figure 17.19. Using the slice pipe

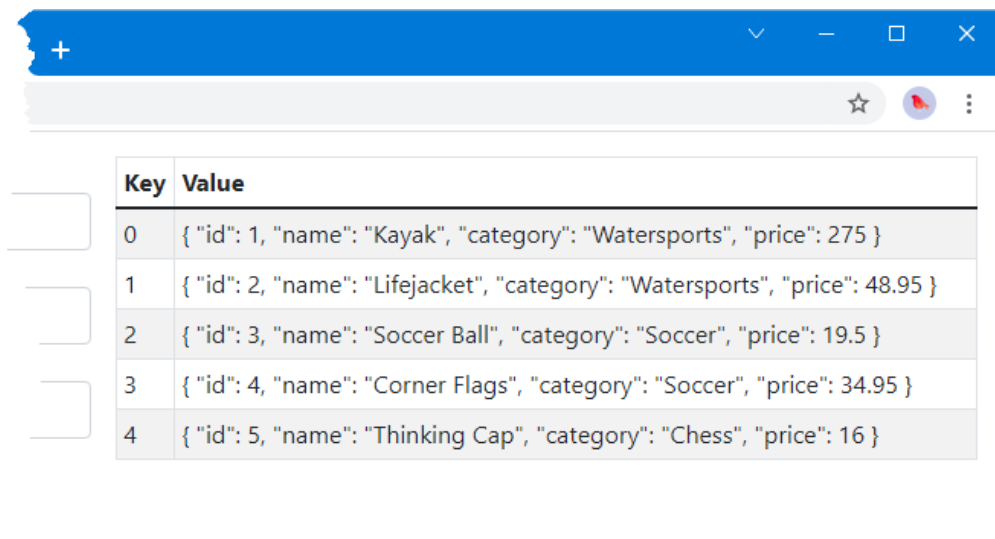
17.4.8 Formatting key-value pairs

The `keyvalue` pipe operates on an object or a map and returns a sequence of key-value pairs. Each object in the sequence is represented as an object with `key` and `value` properties, and listing 17.26 replaces the contents of the `productTable.component.html` file to demonstrate the use of the pipe to enumerate the contents of the array returned by the `getProducts` method.

Listing 17.26. Using the `keyvalue` pipe in the `productTable.component.html` file in the `src/app` folder

```
<table class="table table-sm table-bordered table-striped">
  <thead><tr><th>Key</th><th>Value</th></tr></thead>
  <tbody>
    <tr *paFor="let item of Products() | keyvalue">
      <td>{{ item.key }}</td>
      <td>{{ item.value | json }}</td>
    </tr>
  </tbody>
</table>
```

When used on an array, the keys are the array indexes, and the values are the objects in the array. The objects in the array are formatted using the `json` filter, producing the results shown in figure 17.20.



Key	Value
0	{ "id": 1, "name": "Kayak", "category": "Watersports", "price": 275 }
1	{ "id": 2, "name": "Lifejacket", "category": "Watersports", "price": 48.95 }
2	{ "id": 3, "name": "Soccer Ball", "category": "Soccer", "price": 19.5 }
3	{ "id": 4, "name": "Corner Flags", "category": "Soccer", "price": 34.95 }
4	{ "id": 5, "name": "Thinking Cap", "category": "Chess", "price": 16 }

Figure 17.20. Using the keyvalue pipe

17.4.9 Selecting values

The `select` pipe selects a string based on a value, allowing context-sensitive values to be displayed to the user. The mapping between values and strings is defined as a simple map, as shown in listing 17.27.

Listing 17.27. Mapping values to strings in the `productTable.component.ts` file in the `src/app` folder

```
import { Component, Input, Signal, QueryList, ViewChildren,
  ChangeDetectorRef } from "@angular/core";
import { Model } from "../repository.model";
import { Product } from "../product.model";

@Component({
  selector: "paProductTable",
  templateUrl: "productTable.component.html"
})
export class ProductTableComponent {

  constructor(private changeRef: ChangeDetectorRef) {}

  @Input({ alias: "model", required: true })
  dataModel!: Model;

  get Products(): Signal<Product[]> {
    return this.dataModel.Products;
  }

  getProduct(key: number): Product | undefined {
```

```

        return this.dataModel?.getProduct(key);
    }

    deleteProduct(key: number) {
        this.dataModel.deleteProduct(key);
    }

    taxRate: number = 0;
    categoryFilter: string | undefined;
    itemCount: number = 3;

    // dateObject: Date = new Date(2020, 1, 20);
    // dateString: string = "2020-02-20T00:00:00.000Z";
    // dateNumber: number = 1582156800000;

    selectMap = {
        "Watersports": "stay dry",
        "Soccer": "score goals",
        "other": "have fun"
    }
}

```

The other mapping is used as a fallback when there is no match with the other values. In listing 17.28, I have applied the pipe to select a message to display to the user.

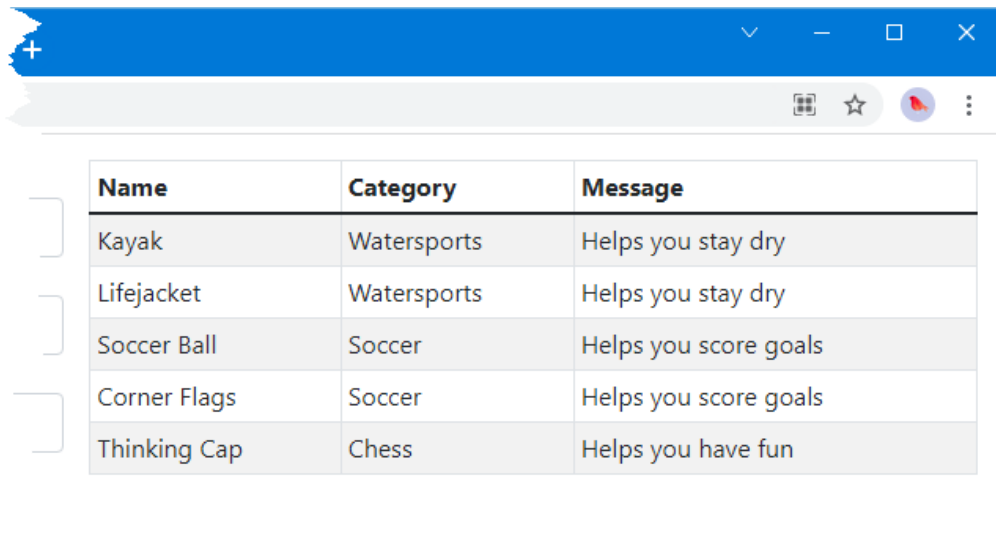
Listing 17.28. Using the pipe in the productTable.component.html file in the src/app folder

```

<table class="table table-sm table-bordered table-striped">
  <thead><tr><th>Name</th><th>Category</th><th>Message</th></tr></thead>
  <tbody>
    <tr *paFor="let item of Products()">
      <td>{{ item.name }}</td>
      <td>{{ item.category }}</td>
      <td>Helps you {{ item.category | i18nSelect:selectMap }} </td>
    </tr>
  </tbody>
</table>

```

The pipe is provided with the map as an argument and produces the response shown in figure 17.21.



Name	Category	Message
Kayak	Watersports	Helps you stay dry
Lifejacket	Watersports	Helps you stay dry
Soccer Ball	Soccer	Helps you score goals
Corner Flags	Soccer	Helps you score goals
Thinking Cap	Chess	Helps you have fun

Figure 17.21. Selecting values using the `i18nSelect` pipe

17.4.10 Pluralizing values

The `i18nPlural` pipe is used to select an expression that describes a numeric value. The mapping between values and expressions is expressed as a simple map, as shown in listing 17.29.

Listing 17.29. Mapping numbers to strings in the `productTable.component.ts` file in the `src/app` folder

```
import { Component, Input, Signal, QueryList, ViewChildren,
  ChangeDetectorRef } from "@angular/core";
import { Model } from "../repository.model";
import { Product } from "../product.model";

@Component({
  selector: "paProductTable",
  templateUrl: "productTable.component.html"
})
export class ProductTableComponent {

  // ...statements omitted for brevity...

  selectMap = {
    "Watersports": "stay dry",
    "Soccer": "score goals",
    "other": "have fun"
  }

  numberMap = {
```

```

    "=1": "one product",
    "=2": "two products",
    "other": "# products"
  }
}

```

Each mapping is expressed with an equals sign followed by the number. The `other` value is a fallback, and the result it produces can refer to the number value using the `#` placeholder character. Listing 17.30 shows the results that can be produced using the example mappings.

Listing 17.30. Using the pipe in the `productTable.component.html` file in the `src/app` folder

```

<table class="table table-sm table-bordered table-striped">
  <thead><tr><th>Name</th><th>Category</th><th>Message</th></tr></thead>
  <tbody>
    <tr *paFor="let item of Products()">
      <td>{{ item.name }}</td>
      <td>{{ item.category }}</td>
      <td>Helps you {{ item.category | i18nSelect:selectMap }} </td>
    </tr>
  </tbody>
</table>

<div class="bg-info text-white p-2">
  <div>There is {{ 1 | i18nPlural:numberMap }} </div>
  <div>There are {{ 2 | i18nPlural:numberMap }} </div>
  <div>There are {{ 100 | i18nPlural:numberMap }} </div>
</div>

```

The mapping is specified as the argument to the pipe, and the values in listing 17.30 produce the result shown in figure 17.22.

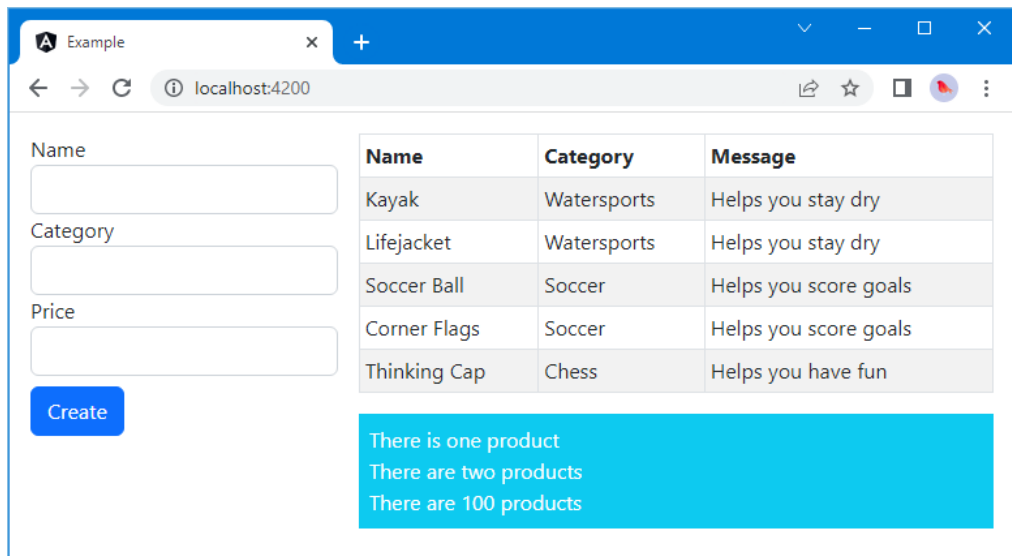


Figure 17.22. Selecting values using the `i18nPlural` pipe

17.4.11 Using the *async* pipe

Angular includes the `async` pipe, which can be used to consume `Observable` objects directly in a view, selecting the last object received from the event sequence. This is an impure pipe because its changes are driven from outside of the view in which it is used, meaning that its `transform` method will be called often, even if a new event has not been received from the `Observable`.

You can see this pipe used in later chapters to receive events from observables provided by the Angular API, but for this chapter, I am going to generate test events. Listing 17.31 adds an `Observable<number>` property to the `ProductTableComponent` class and uses it to generate a sequence of number values.

Listing 17.31. Adding an observable in the `productTable.component.ts` file in the `src/app` folder

```
import { Component, Input, Signal, QueryList, ViewChildren,
  ChangeDetectorRef } from "@angular/core";
import { Model } from "../repository.model";
import { Product } from "../product.model";
import { interval } from "rxjs";

@Component({
  selector: "paProductTable",
  templateUrl: "productTable.component.html"
})
export class ProductTableComponent {

  // ...statements omitted for brevity...

  numberMap = {
    "=1": "one product",
    "=2": "two products",
    "other": "# products"
  }

  numbers = interval(1000);
}
```

Listing 17.32 applies the `async` pipe to display the values received from the observable.

Listing 17.32. Using the `async` pipe in the `productTable.component.html` file in the `src/app` folder

```
<table class="table table-sm table-bordered table-striped">
  <thead><tr><th>Name</th><th>Category</th><th>Message</th></tr></thead>
  <tbody>
    <tr *paFor="let item of Products()">
      <td>{{ item.name }}</td>
      <td>{{ item.category }}</td>
      <td>Helps you {{ item.category | i18nSelect:selectMap }} </td>
    </tr>
```

```

    </tbody>
  </table>

  <div class="bg-info text-white p-2">
    <div> Counter: {{ numbers | async }} </div>
  </div>

```

The string interpolation binding expression gets the `numbers` property from the component and passes it to the `async` pipe, which keeps track of the most recent event that has been received, as shown in figure 17.23.

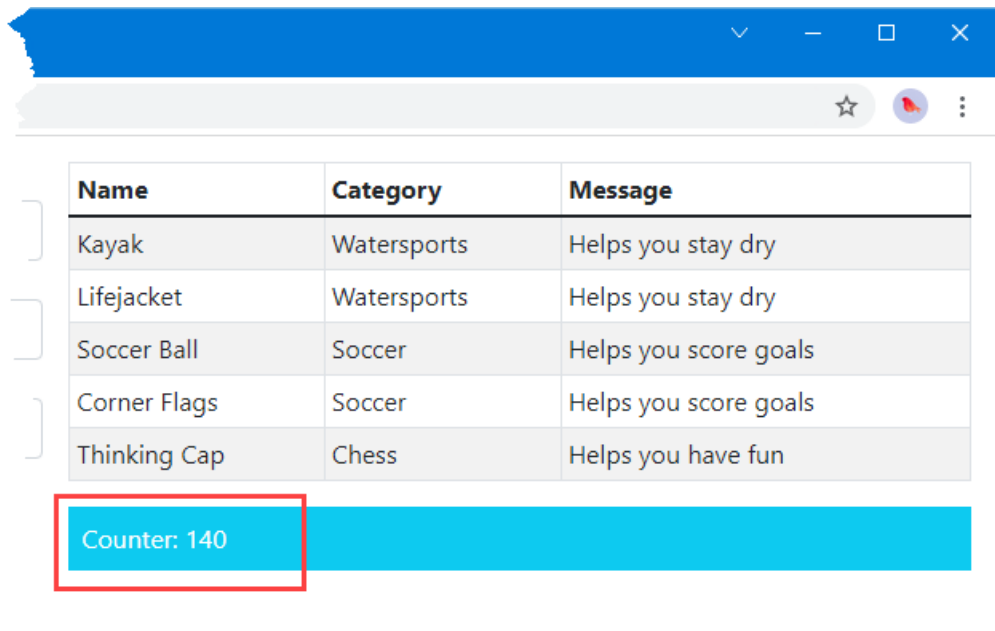


Figure 17.23. Using the `async` pipe

The `async` pipe can be used with other pipes, such as the currency pipe shown in listing 17.33.

Listing 17.33. Combining pipes in the `productTable.component.html` file in the `src/app` folder

```

<table class="table table-sm table-bordered table-striped">
  <thead><tr><th>Name</th><th>Category</th><th>Message</th></tr></thead>
  <tbody>
    <tr *paFor="let item of Products()">
      <td>{{ item.name }}</td>
      <td>{{ item.category }}</td>
      <td>Helps you {{ item.category | i18nSelect:selectMap }} </td>
    </tr>
  </tbody>
</table>

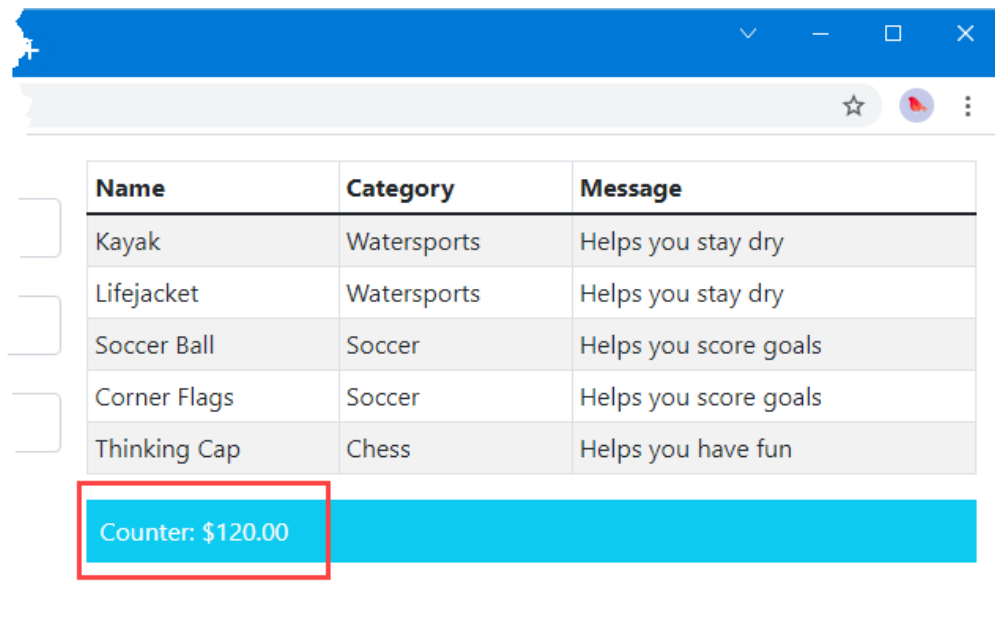
```

```

<div class="bg-info text-white p-2">
  <div>
    Counter: {{ numbers | async | currency:"USD":"symbol":"2.2-2" }}
  </div>
</div>

```

As each value is received, it is passed from the `async` pipe to the `currency` pipe, producing the result shown in figure 17.24.



Name	Category	Message
Kayak	Watersports	Helps you stay dry
Lifejacket	Watersports	Helps you stay dry
Soccer Ball	Soccer	Helps you score goals
Corner Flags	Soccer	Helps you score goals
Thinking Cap	Chess	Helps you have fun

Counter: \$120.00

Figure 17.24. Combining pipes

17.5 Summary

In this chapter, I introduced pipes and explained how they are used to transform data values so they can be presented to the user in the template. I demonstrated the process for creating custom pipes, explained how some pipes are pure and others are not and demonstrated the built-in pipes that Angular provides for handling common tasks.

- Pipes prepare data values for presentation to the user and are defined using the `@Pipe` decorator.
- Angular provides built-in pipes for common tasks, including formatting currencies and dates, and consuming observable data values.
- Custom pipes can be created to perform specialized formatting tasks.

In the next chapter, I introduce services, which can be used to simplify the design of Angular applications and allow building blocks to easily collaborate.

18

Using services

This book covers

- Understanding services and the problems they solve
- Defining and registering services
- Consuming services using dependency injection
- Using services in components, pipes, and directives
- Using services to isolate components

Services are objects that provide common functionality to support other building blocks in an application, such as directives, components, and pipes. What's important about services is the way that they are used, which is through a process called *dependency injection*. Using services can increase the flexibility and scalability of an Angular application, but dependency injection can be a difficult topic to understand. To that end, I start this chapter slowly and explain the problems that services and dependency injection can be used to solve, how dependency injection works, and why you should consider using services in your projects. Table 18.1 puts services in context.

Table 18.1. Putting services in context

Question	Answer
What are they?	Services are objects that define the functionality required by other building blocks such as components or directives. What separates services from regular objects is that they are provided to building blocks by an external provider, rather than being created directly using the <code>new</code> keyword or received by an input property.

Why are they useful?	Services simplify the structure of applications, make it easier to move or reuse functionality, and make it easier to isolate building blocks for effective unit testing.
How are they used?	Classes declare dependencies on services using constructor parameters, which are then resolved using the set of services for which the application has been configured. Services are classes to which the <code>@Injectable</code> decorator has been applied.
Are there any pitfalls or limitations?	Dependency injection is a contentious topic and not all developers like using it. If you don't perform unit tests or if your applications are relatively simple, the extra work required to implement dependency injection is unlikely to pay any long-term dividends.
Are there any alternatives?	Services and dependency injection are hard to avoid because Angular uses them to provide access to built-in functionality. But you are not required to define services for your custom functionality if that is your preference.

Table 18.2 summarizes the chapter.

Table 18.2. Chapter summary

Problem	Solution	Listing
Avoiding the need to distribute shared objects manually	Use services	1–14, 21–28
Declaring a dependency on a service	Add a constructor parameter with the type of service you require	15–20

18.1 Preparing the example project

I continue using the example project in this chapter that I have been working with since chapter 9. To prepare for this chapter, I have replaced the contents of the template for the `ProductTable` component with the elements shown in listing 18.1.

TIP You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/manningbooks/pro-angular-16>. See chapter 1 for how to get help if you have problems running the examples.

Listing 18.1. Replacing the `productTable.component.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 Products(); let i = index">
  <td>{{i + 1}}</td>
  <td>{{item.name}}</td>
  <td>{{item.category}}</td>
  <td>{{item.price | currency:"USD": "symbol" }}</td>
  <td class="text-center">
    <button class="btn btn-danger btn-sm"
      (click)="deleteProduct(item.id)">
      Delete
    </button>
  </td>
</tr>
</tbody>
</table>

```

Run the following command 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 18.1.

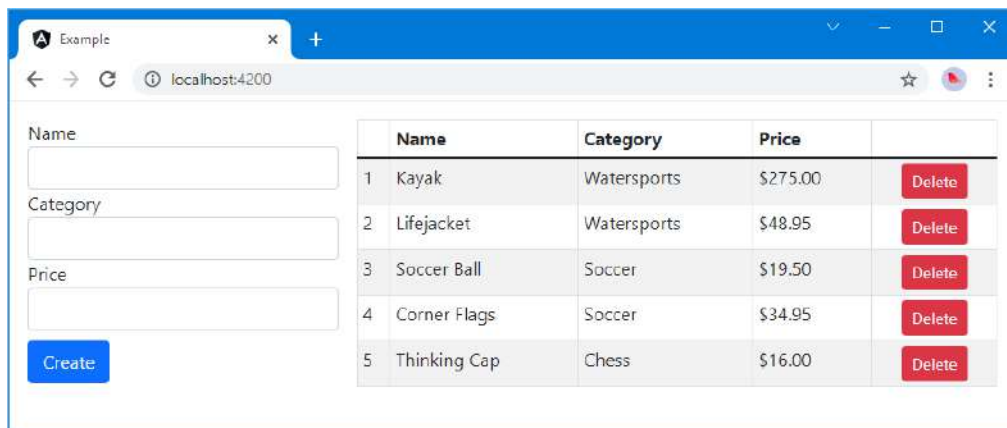


Figure 18.1. Running the example application

18.2 Understanding the object distribution problem

In chapter 16, I added components to the project to help break up the monolithic structure of the application. In doing this, I used input and output properties to connect components, using host elements to bridge the isolation that Angular enforces between a parent component and its children. I also showed you how to query the contents of the template for view children, which complements the content children feature described in chapter 15.

These techniques for coordinating between directives and components can be powerful and useful if applied carefully. But they can also end up as a general tool for distributing shared objects throughout an application, where the result is to increase the complexity of the application and to tightly bind components together.

18.2.1 Demonstrating the problem

To help demonstrate the problem, I am going to add a shared object to the project and two components that rely on it. I created a file called `discount.service.ts` to the `src/app` folder and defined the class shown in listing 18.2. I'll explain the significance of the `service` part of the filename later in the chapter.

Listing 18.2. The contents of the `discount.service.ts` file in the `src/app` folder

```
export class DiscountService {
  private discountValue: number = 10;

  public get discount(): number {
    return this.discountValue;
  }

  public set discount(newValue: number) {
    this.discountValue = newValue ?? 0;
  }

  public applyDiscount(price: number) {
    return Math.max(price - this.discountValue, 5);
  }
}
```

The `DiscountService` class defines a private property called `discountValue` that is used to store a number that will be used to reduce the product prices in the data model. This value is exposed through getters and setters called `discount`, and there is a convenience method called `applyDiscount` that reduces a price while ensuring that a price is never less than \$5.

For the first component that makes use of the `DiscountService` class, I added a file called `discountDisplay.component.ts` to the `src/app` folder and added the code shown in listing 18.3.

Listing 18.3. The contents of the `discountDisplay.component.ts` file in the `src/app` folder

```
import { Component, Input } from "@angular/core";
import { DiscountService } from "../discount.service";

@Component({
  selector: "paDiscountDisplay",
  template: `<div class="bg-info text-white p-2 my-2">
    The discount is {{discounter.discount }}
  </div>`
})
export class PaDiscountDisplayComponent {

  @Input({alias: "discounter", required: true})
  discounter!: DiscountService;
}
```

The `DiscountDisplayComponent` uses an inline template to display the discount amount, which is obtained from a `DiscountService` object received through an input property.

For the second component that makes use of the `DiscountService` class, I added a file called `discountEditor.component.ts` to the `src/app` folder and added the code shown in listing 18.4.

Listing 18.4. The contents of the `discountEditor.component.ts` file in the `src/app` folder

```
import { Component, Input } from "@angular/core";
import { DiscountService } from "../discount.service";

@Component({
  selector: "paDiscountEditor",
  template: `<div class="form-group">
    <label>Discount</label>
    <input [(ngModel)]="discounter.discount"
      class="form-control" type="number" />
  </div>`
})
export class PaDiscountEditorComponent {

  @Input({ alias: "discounter", required: true })
  discounter!: DiscountService;
}
```

The `DiscountEditorComponent` uses an inline template with an input element that allows the discount amount to be edited. The input element has a two-way binding on the `DiscountService.discount` property that targets the `ngModel` directive. Listing 18.5 shows the new components being enabled in the Angular module.

Listing 18.5. Enabling the components in the `app.module.ts` file in the `src/app` folder

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

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

import { ProductComponent } from './component';
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';
import { ProductTableComponent } from './productTable.component';
import { ProductFormComponent } from './productForm.component';
import { PaToggleView } from './toggleView.component';
import { PaAddTaxPipe } from './addTax.pipe';
import { PaCategoryFilterPipe } from './categoryFilter.pipe';

import { LOCALE_ID } from "@angular/core";
import localeFr from '@angular/common/locales/fr';
import { registerLocaleData } from '@angular/common';

import { PaDiscountDisplayComponent } from './discountDisplay.component';
```

```

import { PaDiscountEditorComponent } from "../discountEditor.component";

registerLocaleData(localeFr);

@NgModule({
  declarations: [
    ProductComponent, PaAttrDirective, PaModel, PaStructureDirective,
    PaIteratorDirective, PaCellColor, PaCellColorSwitcher,
    ProductTableComponent, ProductFormComponent, PaToggleView,
    PaAddTaxPipe, PaCategoryFilterPipe, PaDiscountDisplayComponent,
    PaDiscountEditorComponent
  ],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule
  ],
  //providers: [{ provide: LOCALE_ID, useValue: "fr-FR" }],
  bootstrap: [ProductComponent]
})
export class AppModule { }

```

To get the new components working, I added them to the parent component's template, positioning the new content underneath the table that lists the products, which means that I need to edit the `productTable.component.html` file, as shown in listing 18.6.

Listing 18.6. Adding elements in the `productTable.component.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 Products(); let i = index">
      <td>{{i + 1}}</td>
      <td>{{item.name}}</td>
      <td>{{item.category}}</td>
      <td>{{item.price | currency:"USD": "symbol" }}</td>
      <td class="text-center">
        <button class="btn btn-danger btn-sm"
          (click)="deleteProduct(item.id)">
          Delete
        </button>
      </td>
    </tr>
  </tbody>
</table>

<paDiscountEditor [discounter]="discounter"></paDiscountEditor>
<paDiscountDisplay [discounter]="discounter"></paDiscountDisplay>

```

These elements correspond to the components' selector properties in listing 18.3 and listing 18.4 and use data bindings to set the value of the input properties. The final step is to create

an object in the parent component that will provide the value for the data binding expressions, as shown in listing 18.7.

Listing 18.7. Creating the shared object in the `productTable.component.ts` file in the `src/app` folder

```
import { Component, Input, Signal, QueryList, ViewChildren,
        ChangeDetectorRef } from "@angular/core";
import { Model } from "../repository.model";
import { Product } from "../product.model";
import { interval } from "rxjs";
import { DiscountService } from "../discount.service";

@Component({
  selector: "paProductTable",
  templateUrl: "productTable.component.html"
})
export class ProductTableComponent {
  discounter: DiscountService = new DiscountService();

  //constructor(private changeRef: ChangeDetectorRef) {}

  @Input({ alias: "model", required: true })
  dataModel!: Model;

  get Products(): Signal<Product[]> {
    return this.dataModel.Products;
  }

  getProduct(key: number): Product | undefined {
    return this.dataModel?.getProduct(key);
  }

  deleteProduct(key: number) {
    this.dataModel.deleteProduct(key);
  }

  taxRate: number = 0;
  categoryFilter: string | undefined;
  itemCount: number = 3;

  // selectMap = {
  //   "Watersports": "stay dry",
  //   "Soccer": "score goals",
  //   "other": "have fun"
  // }

  // numberMap = {
  //   "=1": "one product",
  //   "=2": "two products",
  //   "other": "# products"
  // }

  // numbers = interval(1000);
}
```

Figure 18.2 shows the content from the new components. Changes to the value in the `input` element provided by one of the components will be reflected in the content presented by the other component, reflecting the use of the shared `DiscountService` object and its `discount` property.

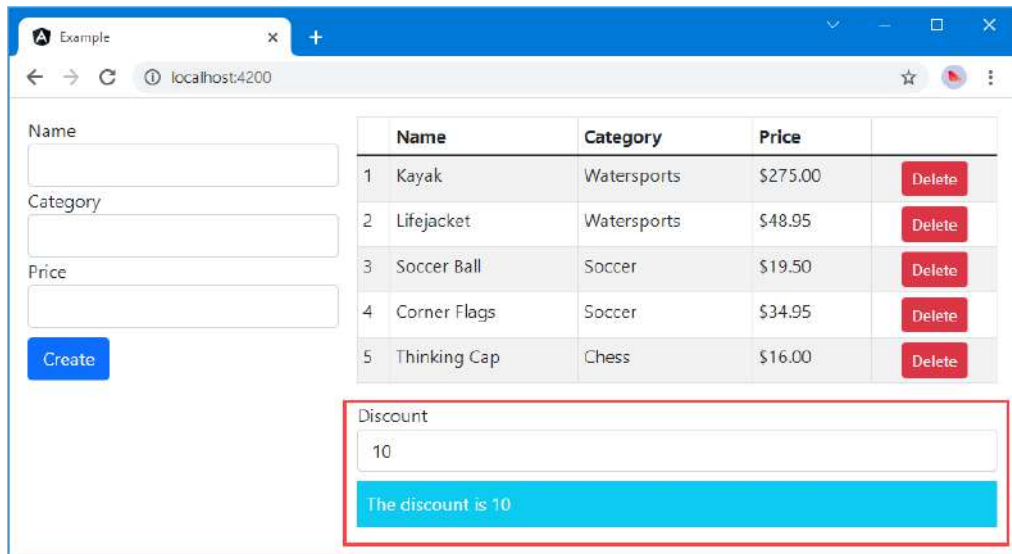


Figure 18.2. Adding components to the example application

The process for adding the new components and the shared object was straightforward and logical, until the final stage. The problem arises in the way that I had to create and distribute the shared object: the instance of the `DiscountService` class.

Because Angular isolates components from one another, I had no way to share the `DiscountService` object directly between the `DiscountEditorComponent` and `DiscountDisplayComponent`. Each component could have created its own `DiscountService` object, but that means changes from the editor component wouldn't be shown in the display component.

That is what led me to create the `DiscountService` object in the product table component, which is the first shared ancestor of the discount editor and display components. This allowed me to distribute the `DiscountService` object through the product table component's template, ensuring that a single object was shared with both of the components that need it.

But there are a couple of problems. The first is that the `ProductTableComponent` class doesn't need or use a `DiscountService` object to deliver its functionality. It just happens to be the first common ancestor of the components that do need the object. And creating the shared object in the `ProductTableComponent` class makes that class slightly more complex

and slightly more difficult to test effectively. This is a modest increment of complexity, but it will occur for every shared object that the application requires—and a complex application can depend on a lot of shared objects, each of which ends up being created by components that just happen to be the first common ancestor of the classes that depend on them.

The second problem is hinted at by the term *first common ancestor*. The `ProductTableComponent` class happens to be the parent of both of the classes that depend on the `DiscountService` object, but think about what would happen if I wanted to move the `DiscountEditorComponent` so that it was displayed under the form rather than the table. In this situation, I have to work my way up the tree of components until I find a common ancestor, which would end up being the root component. Then I would have to work my way down the component tree adding input properties and modifying templates so that each intermediate component could receive the `DiscountService` object from its parent and pass it on to any children that have descendants that need it. The same applies to any directives that depend on receiving a `DiscountService` object, where any component whose template contains data bindings that target that directive must make sure they are part of the distribution chain, too.

The result is that the components and directives in the application become tightly bound together. A major refactoring is required if you need to move or reuse a component in a different part of the application and the management of the input properties and data bindings becomes unmanageable.

18.2.2 Distributing objects as services using dependency injection

There is a better way to distribute objects to the classes that depend on them, which is to use *dependency injection*, where objects are provided to classes from an external source. Angular includes a built-in dependency injection system and supplies the external source of objects, known as *providers*. In the sections that follow, I rework the example application to provide the `DiscountService` object without needing to use the component hierarchy as a distribution mechanism.

PREPARING THE SERVICE

Any object that is managed and distributed through dependency injection is called a *service*, which is why I selected the name `DiscountService` for the class that defines the shared object and why that class is defined in a file called `discount.service.ts`. Angular denotes service classes using the `@Injectable` decorator, as shown in listing 18.8. The `@Injectable` decorator doesn't define any configuration properties.

Listing 18.8. Preparing a service in the `discount.service.ts` file in the `src/app` folder

```
import { Injectable } from "@angular/core";

@Injectable()
export class DiscountService {
  private discountValue: number = 10;

  public get discount(): number {
```

```

        return this.discountValue;
    }

    public set discount(newValue: number) {
        this.discountValue = newValue || 0;
    }

    public applyDiscount(price: number) {
        return Math.max(price - this.discountValue, 5);
    }
}

```

TIP 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.

PREPARING THE DEPENDENT COMPONENTS

A class declares dependencies using its constructor. When Angular needs to create an instance of the class—such as when it finds an element that matches the `selector` property defined by a component—its constructor is inspected, and the type of each argument is examined. Angular then uses the services that have been defined to try to satisfy the dependencies. The term *dependency injection* arises because each dependency is *injected* into the constructor to create the new instance.

For the example application, it means that the components that depend on a `DiscountService` object no longer require input properties and can declare a constructor dependency instead. Listing 18.9 shows the changes to the `DiscountDisplayComponent` class.

Listing 18.9. Declaring a dependency in the `discountDisplay.component.ts` file in the `src/app` folder

```

import { Component, Input } from "@angular/core";
import { DiscountService } from "../discount.service";

@Component({
  selector: "paDiscountDisplay",
  template: `<div class="bg-info text-white p-2 my-2">
    The discount is {{discounter.discount }}
  </div>`
})
export class PaDiscountDisplayComponent {

  constructor(public discounter: DiscountService) { }

  // @Input({alias: "discounter", required: true})
  // discounter!: DiscountService;
}

```

The same change can be applied to the `DiscountEditorComponent` class, replacing the input property with a dependency declared through the constructor, as shown in listing 18.10.

Listing 18.10. Declaring a dependency in the discountEditor.component.ts file in the src/app folder

```
import { Component, Input } from "@angular/core";
import { DiscountService } from "../discount.service";

@Component({
  selector: "paDiscountEditor",
  template: `<div class="form-group">
    <label>Discount</label>
    <input [(ngModel)]="discounter.discount"
      class="form-control" type="number" />
  </div>`
})
export class PaDiscountEditorComponent {

  constructor(public discounter: DiscountService) { }

  // @Input({ alias: "discounter", required: true})
  // discounter!: DiscountService;
}
```

These are small changes, but they avoid the need to distribute objects using templates and input properties and produce a more flexible application.

I can now remove the DiscountService object from the product table component, as shown in listing 18.11.

Listing 18.11. Removing the shared object in the productTable.component.ts file in the src/app folder

```
import { Component, Input, Signal, QueryList, ViewChildren,
  ChangeDetectorRef } from "@angular/core";
import { Model } from "../repository.model";
import { Product } from "../product.model";
import { interval } from "rxjs";
import { DiscountService } from "../discount.service";

@Component({
  selector: "paProductTable",
  templateUrl: "productTable.component.html"
})
export class ProductTableComponent {
  //discounter: DiscountService = new DiscountService();

  //constructor(private changeRef: ChangeDetectorRef) {}

  @Input({ alias: "model", required: true})
  dataModel!: Model;

  get Products(): Signal<Product[]> {
    return this.dataModel.Products;
  }

  getProduct(key: number): Product | undefined {
    return this.dataModel?.getProduct(key);
  }
}
```

```

    deleteProduct(key: number) {
      this.dataModel.deleteProduct(key);
    }

    taxRate: number = 0;
    categoryFilter: string | undefined;
    itemCount: number = 3;
  }

```

Since the parent component is no longer providing the shared object through data bindings, I can remove them from the template, as shown in listing 18.12.

Listing 18.12. Removing the data bindings in the productTable.component.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 Products(); let i = index">
      <td>{{i + 1}}</td>
      <td>{{item.name}}</td>
      <td>{{item.category}}</td>
      <td>{{item.price | currency:"USD": "symbol" }}</td>
      <td class="text-center">
        <button class="btn btn-danger btn-sm"
          (click)="deleteProduct(item.id)">
          Delete
        </button>
      </td>
    </tr>
  </tbody>
</table>

<paDiscountEditor></paDiscountEditor>
<paDiscountDisplay></paDiscountDisplay>

```

REGISTERING THE SERVICE

The final change is to configure the dependency injection feature so that it can provide `DiscountService` objects to the components that require them. To make the service available throughout the application, it is registered in the Angular module, as shown in listing 18.13.

Listing 18.13. Registering a service in the app.module.ts file in the src/app folder

```

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

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

```



```

import { ProductComponent } from './component';
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';
import { ProductTableComponent } from './productTable.component';
import { ProductFormComponent } from './productForm.component';
import { PaToggleView } from './toggleView.component';
import { PaAddTaxPipe } from './addTax.pipe';
import { PaCategoryFilterPipe } from './categoryFilter.pipe';

import { LOCALE_ID } from '@angular/core';
import localeFr from '@angular/common/locales/fr';
import { registerLocaleData } from '@angular/common';

import { PaDiscountDisplayComponent } from './discountDisplay.component';
import { PaDiscountEditorComponent } from './discountEditor.component';
import { DiscountService } from './discount.service';

registerLocaleData(localeFr);

@NgModule({
  declarations: [
    ProductComponent, PaAttrDirective, PaModel, PaStructureDirective,
    PaIteratorDirective, PaCellColor, PaCellColorSwitcher,
    ProductTableComponent, ProductFormComponent, PaToggleView,
    PaAddTaxPipe, PaCategoryFilterPipe, PaDiscountDisplayComponent,
    PaDiscountEditorComponent
  ],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule
  ],
  providers: [DiscountService],
  bootstrap: [ProductComponent]
})
export class AppModule { }

```

The `NgModule` decorator's `providers` property is set to an array of the classes that will be used as services. There is only one service at the moment, which is provided by the `DiscountService` class.

When you save the changes to the application, there won't be any visual changes, but the dependency injection feature will be used to provide the components with the `DiscountService` object they require.

REVIEWING THE DEPENDENCY INJECTION CHANGES

Angular seamlessly integrates dependency injection into its feature set. Each time Angular encounters an element that requires a new building block, such as a component or a pipe, it examines the class constructor to check what dependencies have been declared and uses its services to try to resolve them. The set of services used to resolve dependencies includes the custom services defined by the application, such as the `DiscountService` service that was

registered in listing 18.13, and a set of built-in services provided by Angular that will be described in later chapters.

The changes to introduce dependency injection in the previous section didn't result in a big-bang change in the way that the application works—or any visible change at all. But there is a profound difference in the way that the application is put together that makes it more flexible and fluid. The best demonstration of this is to add the components that require the `DiscountService` to a different part of the application, as shown in listing 18.14.

Listing 18.14. Adding components in the `productForm.component.html` file in the `src/app` folder

```
<form #form="ngForm" (ngSubmit)="submitForm(form)">
  <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="name" [(ngModel)]="newProduct.price" />
  </div>
  <button class="btn btn-primary mt-2" type="submit">
    Create
  </button>
</form>

<paDiscountEditor></paDiscountEditor>
<paDiscountDisplay></paDiscountDisplay>
```

These new elements duplicate the discount display and editor components so they appear below the form used to create new products, as shown in figure 18.3.

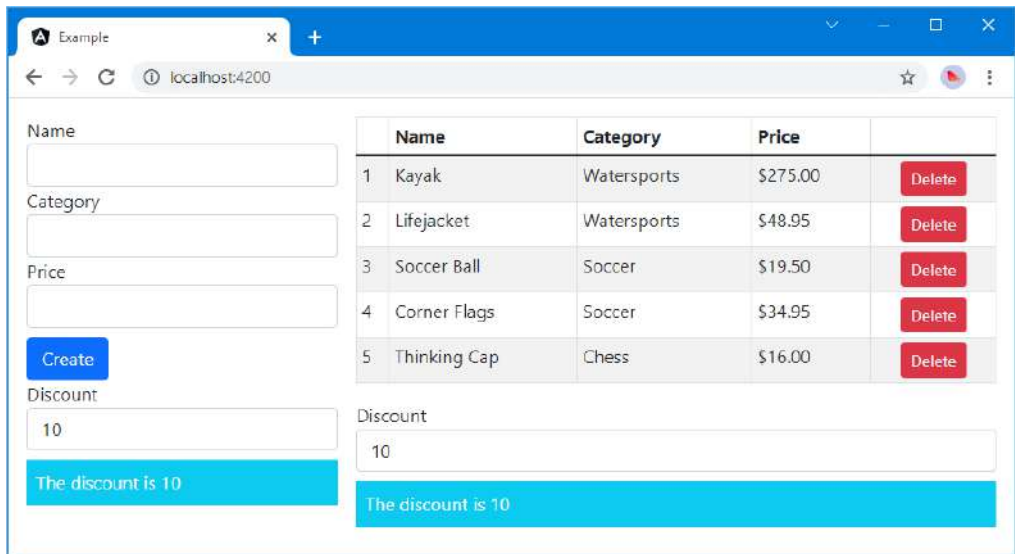


Figure 18.3. Duplicating components with dependencies

There are two points of note. First, using dependency injection made this a simple process of adding elements to a template, without needing to modify the ancestor components to provide a `DiscountService` object using input properties.

The second point of note is that all the components in the application that have declared a dependency on `DiscountService` have received the same object. If you edit the value in either of the input elements, the changes will be reflected in the other input element and the string interpolation bindings, as shown in figure 18.4.

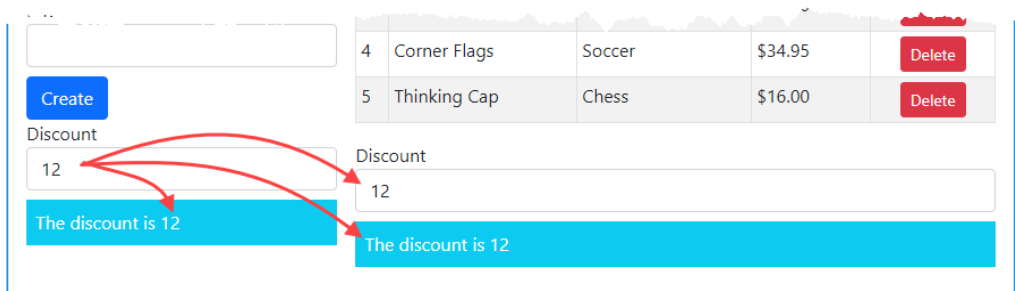


Figure 18.4. Checking that the dependency is resolved using a shared object

18.2.3 Declaring dependencies in other building blocks

It isn't just components that can declare constructor dependencies. Once you have defined a service, you can use it more widely, including other building blocks in the application, such as pipes and directives, as demonstrated in the sections that follow.

DECLARING A DEPENDENCY IN A PIPE

Pipes can declare dependencies on services by defining a constructor with arguments for each required service. To demonstrate, I added a file called `discount.pipe.ts` to the `src/app` folder and used it to define the pipe shown in listing 18.15.

Listing 18.15. The contents of the `discount.pipe.ts` file in the `src/app` folder

```
import { Pipe } from "@angular/core";
import { DiscountService } from "../discount.service";

@Pipe({
  name: "discount",
  pure: false
})
export class PaDiscountPipe {

  constructor(private discount: DiscountService) { }

  transform(price: number): number {
    return this.discount.applyDiscount(price);
  }
}
```

The `PaDiscountPipe` class is a pipe that receives a `price` and generates a result by calling the `DiscountService.applyDiscount` method, where the service is received through the constructor. The `pure` property in the `Pipe` decorator is `false`, which means that the pipe will be asked to update its result when the value stored by the `DiscountService` changes, which won't be recognized by the Angular change-detection process.

TIP This feature should be used with caution because it means that the `transform` method will be called after every change in the application, not just when the service is changed. This is an issue that I expect will be resolved when signals are fully integrated into the Angular framework.

Listing 18.16 shows the new pipe being registered in the application's Angular module.

Listing 18.16. Registering a pipe in the `app.module.ts` file in the `src/app` folder

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

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

import { ProductComponent } from './component';
```

```

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";
import { ProductTableComponent } from "../productTable.component";
import { ProductFormComponent } from "../productForm.component";
import { PaToggleView } from "../toggleView.component";
import { PaAddTaxPipe } from "../addTax.pipe";
import { PaCategoryFilterPipe } from "../categoryFilter.pipe";

import { LOCALE_ID } from "@angular/core";
import localeFr from '@angular/common/locales/fr';
import { registerLocaleData } from '@angular/common';

import { PaDiscountDisplayComponent } from "../discountDisplay.component";
import { PaDiscountEditorComponent } from "../discountEditor.component";
import { DiscountService } from "../discount.service";
import { PaDiscountPipe } from "../discount.pipe";

registerLocaleData(localeFr);

@NgModule({
  declarations: [
    ProductComponent, PaAttrDirective, PaModel, PaStructureDirective,
    PaIteratorDirective, PaCellColor, PaCellColorSwitcher,
    ProductTableComponent, ProductFormComponent, PaToggleView,
    PaAddTaxPipe, PaCategoryFilterPipe, PaDiscountDisplayComponent,
    PaDiscountEditorComponent, PaDiscountPipe
  ],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule
  ],
  providers: [DiscountService],
  bootstrap: [ProductComponent]
})
export class AppModule { }

```

Listing 18.17 shows the new pipe applied to the Price column in the product table.

Listing 18.17. Applying a pipe in the productTable.component.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 Products(); let i = index">
        <td>{{i + 1}}</td>
        <td>{{item.name}}</td>
        <td>{{item.category}}</td>
        <td>{{item.price | discount | currency:"USD":"symbol" }}</td>
        <td class="text-center">

```

```

        <button class="btn btn-danger btn-sm"
            (click)="deleteProduct(item.id)">
            Delete
        </button>
    </td>
</tr>
</tbody>
</table>

<paDiscountEditor></paDiscountEditor>
<paDiscountDisplay></paDiscountDisplay>

```

The discount pipe processes the price to apply the discount and then passes on the value to the currency pipe for formatting. You can see the effect of using the service in the pipe by changing the value in one of the discount input elements, as shown in figure 18.5.

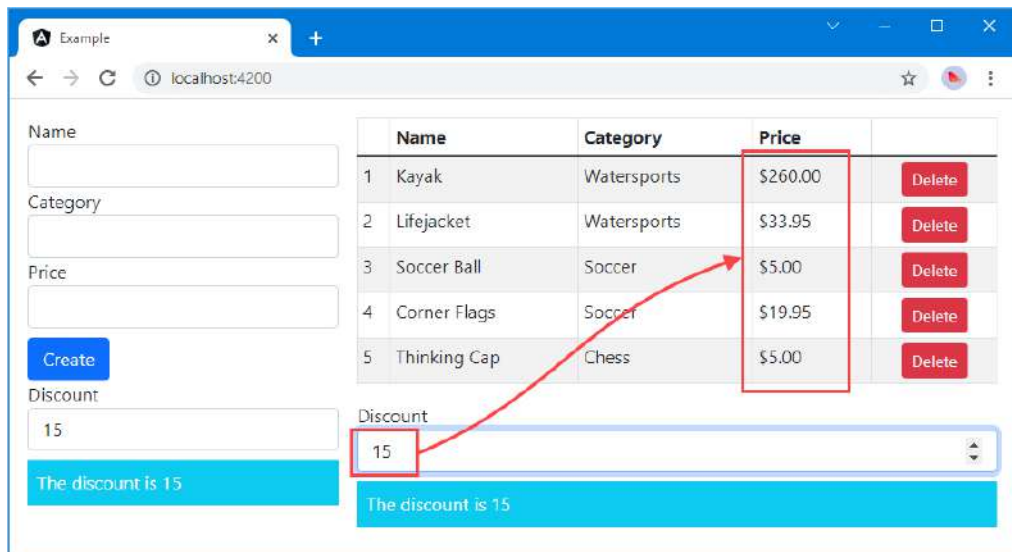


Figure 18.5. Using a service in a pipe

DECLARING DEPENDENCIES IN DIRECTIVES

Directives can also use services. As I explained in chapter 16, components are just directives with templates, so anything that works in a component will also work in a directive.

To demonstrate using a service in a directive, I added a file called `discountAmount.directive.ts` to the `src/app` folder and used it to define the directive shown in listing 18.18.

Listing 18.18. The contents of the `discountAmount.directive.ts` file in the `src/app` folder

```

import { Directive, Input, SimpleChange, KeyValueDiffer,
    KeyValueDiffers } from "@angular/core";
import { DiscountService } from "../discount.service";

```

```

@Directive({
  selector: "td[pa-price]",
  exportAs: "discount"
})
export class PaDiscountAmountDirective {
  private differ?: KeyValueDiffer<any, any>;

  constructor(private keyValueDiffers: KeyValueDiffers,
    private discount: DiscountService) { }

  @Input({ alias: "pa-price", required: true})
  originalPrice!: number;

  discountAmount?: number;

  ngOnInit() {
    this.differ =
      this.keyValueDiffers.find(this.discount).create();
  }

  ngOnChanges(changes: { [property: string]: SimpleChange }) {
    if (changes["originalPrice"] != null) {
      this.updateValue();
    }
  }

  ngDoCheck() {
    if (this.differ?.diff(this.discount) != null) {
      this.updateValue();
    }
  }

  private updateValue() {
    this.discountAmount
      = this.discount.applyDiscount(this.originalPrice);
  }
}

```

Directives don't have an equivalent to the `pure` property used by pipes and must take direct responsibility for responding to changes propagated through services. This directive displays the discounted amount for a product. The `selector` property matches `td` elements that have a `pa-price` attribute, which is also used as an input property to receive the price that will be discounted. The directive exports its functionality using the `exportAs` property and provides a property called `discountAmount` whose value is set to the discount that has been applied to the product.

There are two other points to note about this directive. The first is that the `DiscountService` object isn't the only constructor parameter in the directive's class.

```

...
constructor(private keyValueDiffers: KeyValueDiffers,
  private discount: DiscountService) { }
...

```

The `KeyValueDiffers` parameter is also a dependency that Angular will have to resolve when it creates a new instance of the directive class. This is an example of the built-in services that Angular provides that deliver commonly required functionality.

The second point of note is what the directive does with the services it receives. The components and the pipe that use the `DiscountService` service don't have to worry about tracking updates, either because Angular automatically evaluates the expressions of the data bindings and updates them when the discount rate change (for the components) or because any change in the application triggers an update (for the impure pipe). The data binding for this directive is on the `price` property, which will trigger a change if it is altered. But there is also a dependency on the `discount` property defined by the `DiscountService` class. Changes in the `discount` property are detected using the service received through the constructor, which tracks changes as described in chapter 15. When Angular invokes the `ngDoCheck` method, the directive uses the key-value pair differ to see whether there has been a change. (This change direction could also have been handled by keeping track of the previous update in the directive class, but I wanted to provide an example of using the key-value differ feature.)

The directive also implements the `ngOnChanges` method so that it can respond to changes in the value of the input property. For both types of updates, the `updateValue` method is called, which calculates the discounted price and assigns it to the `discountAmount` property.

Listing 18.19 registers the new directive in the application's Angular module.

Listing 18.19. Registering a directive in the `app.module.ts` file in the `src/app` folder

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

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

import { ProductComponent } from './component';
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';
import { ProductTableComponent } from './productTable.component';
import { ProductFormComponent } from './productForm.component';
import { PaToggleView } from './toggleView.component';
import { PaAddTaxPipe } from './addTax.pipe';
import { PaCategoryFilterPipe } from './categoryFilter.pipe';

import { LOCALE_ID } from '@angular/core';
import localeFr from '@angular/common/locales/fr';
import { registerLocaleData } from '@angular/common';

import { PaDiscountDisplayComponent } from './discountDisplay.component';
import { PaDiscountEditorComponent } from './discountEditor.component';
```



```

import { DiscountService } from "../discount.service";
import { PaDiscountPipe } from "../discount.pipe";
import { PaDiscountAmountDirective } from "../discountAmount.directive";

registerLocaleData(localeFr);

@NgModule({
  declarations: [
    ProductComponent, PaAttrDirective, PaModel, PaStructureDirective,
    PaIteratorDirective, PaCellColor, PaCellColorSwitcher,
    ProductTableComponent, ProductFormComponent, PaToggleView,
    PaAddTaxPipe, PaCategoryFilterPipe, PaDiscountDisplayComponent,
    PaDiscountEditorComponent, PaDiscountPipe,
    PaDiscountAmountDirective
  ],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule
  ],
  providers: [DiscountService],
  bootstrap: [ProductComponent]
})
export class AppModule { }

```

To apply the new directive, listing 18.20 uses the string interpolation binding to access the property provided by the directive and to pass it to the `currency` pipe.

Listing 18.20. Creating a new column in the `productTable.component.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 Products(); let i = index">
      <td>{{i + 1}}</td>
      <td>{{item.name}}</td>
      <td>{{item.category}}</td>
      <td [pa-price]="item.price" #discount="discount">
        {{ discount.discountAmount | currency:"USD":"symbol"}}
      </td>
      <td class="text-center">
        <button class="btn btn-danger btn-sm"
          (click)="deleteProduct(item.id)">
          Delete
        </button>
      </td>
    </tr>
  </tbody>
</table>

<paDiscountEditor></paDiscountEditor>
<paDiscountDisplay></paDiscountDisplay>

```

The directive could have created a host binding on the `textContent` property to set the contents of its host element, but that would have prevented the `currency` pipe from being used. Instead, the directive is assigned to the `discount` template variable, which is then used in the string interpolation binding to access and then format the `discountAmount` value. Figure 18.6 shows the results. Changes to the discount amount in either of the discount editor input elements will be reflected in the new table column.

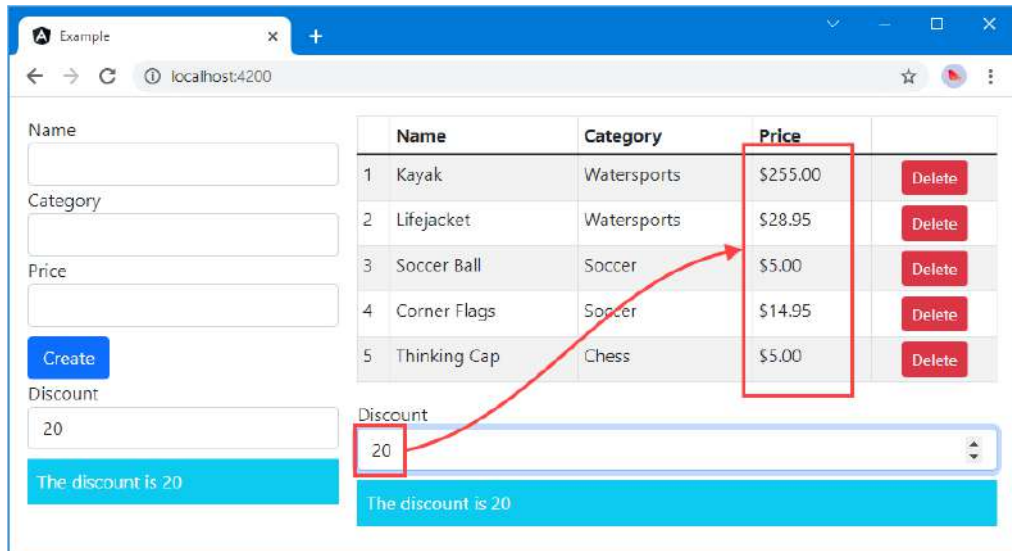


Figure 18.6. Using a service in a directive

18.3 Understanding the test isolation problem

The example application contains a related problem that services and dependency injection can be used to solve. Consider how the `Model` class is used in the root component:

```
import { Component, computed } 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();

  addProduct(p: Product) {
    this.model.saveProduct(p);
  }
}
```

The root component is defined as the `ProductComponent` class, and it sets up a value for its `model` property by creating a new instance of the `Model` class. This works—and is a legitimate way to create an object—but it makes it harder to perform unit testing effectively.

Unit testing works best when you can isolate one small part of the application and focus on it to perform tests. But when you create an instance of the `ProductComponent` class, you are implicitly creating an instance of the `Model` class as well. If you were to run tests on the root component's `addProduct` method and find a problem, you would receive no indication of whether the problem was in the `ProductComponent` or `Model` class.

18.3.1 Isolating components using services and dependency injection

The underlying problem is that the `ProductComponent` class is tightly bound to the `Model` class, which is, in turn, tightly bound to the `SimpleDataSource` class. Dependency injection can be used to tease apart the building blocks in an application so that each class can be isolated and tested on its own. In the sections that follow, I walk through the process of breaking up these tightly coupled classes, following essentially the same process as in the previous section but delving deeper into the example application.

PREPARING THE SERVICES

The `@Injectable` decorator is used to denote services, just as in the previous example. Listing 18.21 shows the decorator applied to the `SimpleDataSource` class.

Listing 18.21. Denoting a service in the `datasource.model.ts` file in the `src/app` folder

```
import { Injectable } from "@angular/core";
import { Product } from "../product.model";

@Injectable()
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;
  }
}
```

No other changes are required. Listing 18.22 shows the same decorator being applied to the data repository, and since this class has a dependency on the `SimpleDataSource` class, it declares it as a constructor dependency rather than creating an instance directly.

Listing 18.22. Denoting a service and dependency in the `repository.model.ts` file in the `src/app` folder

```

import { Product } from "../product.model";
import { SimpleDataSource } from "../datasource.model";
import { Signal, WritableSignal, signal } from "@angular/core";
import { Injectable } from "@angular/core";

@Injectable()
export class Model {
  //private dataSource: SimpleDataSource;
  private products: WritableSignal<Product[]>;
  private locator = (p: Product, id: number | any) => p.id == id;

  constructor(private dataSource: SimpleDataSource) {
    //this.dataSource = new SimpleDataSource();
    this.products = signal(new Array<Product>());
    this.products.mutate(prods =>
      this.dataSource.getData().forEach(p => prods.push(p)));
  }

  // ...statements omitted for brevity...
}

```

The important point to note in this listing is that services can declare dependencies on other services. When Angular comes to create a new instance of a service class, it inspects the constructor and tries to resolve the services in the same way as when dealing with a component or directive.

REGISTERING THE SERVICES

These services must be registered so that Angular knows how to resolve dependencies on them, as shown in listing 18.23.

Listing 18.23. Registering the services in the app.module.ts file in the src/app folder

```

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

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

import { ProductComponent } from './component';
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';
import { ProductTableComponent } from './productTable.component';
import { ProductFormComponent } from './productForm.component';
import { PaToggleView } from './toggleView.component';
import { PaAddTaxPipe } from './addTax.pipe';
import { PaCategoryFilterPipe } from './categoryFilter.pipe';

import { LOCALE_ID } from '@angular/core';
import localeFr from '@angular/common/locales/fr';
import { registerLocaleData } from '@angular/common';

```

```

import { PaDiscountDisplayComponent } from "../discountDisplay.component";
import { PaDiscountEditorComponent } from "../discountEditor.component";
import { DiscountService } from "../discount.service";
import { PaDiscountPipe } from "../discount.pipe";
import { PaDiscountAmountDirective } from "../discountAmount.directive";
import { SimpleDataSource } from "../datasource.model";
import { Model } from "../repository.model";

registerLocaleData(localeFr);

@NgModule({
  declarations: [
    ProductComponent, PaAttrDirective, PaModel, PaStructureDirective,
    PaIteratorDirective, PaCellColor, PaCellColorSwitcher,
    ProductTableComponent, ProductFormComponent, PaToggleView,
    PaAddTaxPipe, PaCategoryFilterPipe, PaDiscountDisplayComponent,
    PaDiscountEditorComponent, PaDiscountPipe,
    PaDiscountAmountDirective
  ],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule
  ],
  providers: [DiscountService, SimpleDataSource, Model],
  bootstrap: [ProductComponent]
})
export class AppModule { }

```

PREPARING THE DEPENDENT COMPONENT

Rather than create a `Model` object directly, the root component can declare a constructor dependency that Angular will resolve using dependency injection when the application starts, as shown in listing 18.24.

Listing 18.24. Declaring a service dependency in the `component.ts` file in the `src/app` folder

```

import { Component, computed } 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(public model: Model) {}

  addProduct(p: Product) {
    this.model.saveProduct(p);
  }
}

```

There is now a chain of dependencies that Angular has to resolve. When the application starts, the Angular module specifies that the `ProductComponent` class needs a `Model` object. Angular inspects the `Model` class and finds that it needs a `SimpleDataSource` object. Angular inspects the `SimpleDataSource` object and finds that there are no declared dependencies and therefore knows that this is the end of the chain. It creates a `SimpleDataSource` object and passes it as an argument to the `Model` constructor to create a `Model` object, which can then be passed to the `ProductComponent` class constructor to create the object that will be used as the root component. All of this happens automatically, based on the constructors defined by each class and the use of the `@Injectable` decorator.

These changes don't create any visible changes in the way that the application works, but they do allow a completely different way of performing unit tests. The `ProductComponent` class requires that a `Model` object is provided as a constructor argument, which allows for a mock object to be used.

Breaking up the direct dependencies between the classes in the application means that each of them can be isolated for unit testing and provided with mock objects through their constructor, allowing the effect of a method or some other feature to be consistently and independently assessed.

18.4 *Completing the adoption of services*

Once you start using services in an application, the process generally takes on a life of its own, and you start to examine the relationships between the building blocks you have created. The extent to which you introduce services is—at least in part—a matter of personal preference.

A good example is the use of the `Model` class in the root component. Although the component does implement a method that uses the `Model` object, it does so because it needs to handle a custom event from one of its child components. The only other reason that the root component has for needing a `Model` object is to pass it on via its template to the other child component using an input property.

This situation isn't an enormous problem, and your preference may be to have these kinds of relationships in a project. After all, each of the components can be isolated for unit testing, and there is some purpose, however limited, to the relationships between them. This kind of relationship between components can help make sense of the functionality that an application provides.

On the other hand, the more you use services, the more the building blocks in your project become self-contained and reusable blocks of functionality, which can ease the process of adding or changing functionality as the project matures.

There is no absolute right or wrong, and you must find the balance that suits you, your team, and, ultimately, your users and customers. Not everyone likes using dependency injection, and not everyone performs unit testing.

My preference is to use dependency injection as widely as possible. I find that the final structure of my applications can differ significantly from what I expect when I start a new project and that the flexibility offered by dependency injection helps me avoid repeated periods

of refactoring. So, to complete this chapter, I am going to push the use of the `Model` service into the rest of the application, breaking the coupling between the root component and its immediate children.

18.4.1 Updating the root component and template

The first changes I will make are to remove the `Model` object from the root component, along with the method that uses it and the input property in the template that distributes the model to one of the child components. Listing 18.25 shows the changes to the component class.

Listing 18.25. Removing the model object from the component.ts file in the src/app folder

```
import { Component, computed } from "@angular/core";
// import { Model } from "../repository.model";
// import { Product } from "../product.model";

@Component({
  selector: "app",
  templateUrl: "template.html"
})
export class ProductComponent {

  // constructor(public model: Model) {}

  // addProduct(p: Product) {
  //   this.model.saveProduct(p);
  // }
}
```

The revised root component class doesn't define any functionality and now exists only to provide the top-level application content in its template. Listing 18.26 shows the corresponding changes in the root template to remove the custom event binding and the input property.

Listing 18.26. Removing the data bindings in the template.html file in the src/app folder

```
<div class="container-fluid">
  <div class="row p-2">
    <div class="col-4 p-2 text-dark">
      <paProductForm></paProductForm>
    </div>
    <div class="col p-2">
      <paProductTable></paProductTable>
    </div>
  </div>
</div>
```

18.4.2 Updating the child components

The component that provides the form for creating new `Product` objects relied on the root component to handle its custom event and update the model. Without this support, the component must now declare a `Model` dependency and perform the update itself, as shown in listing 18.27.

Listing 18.27. Working with the model in the productForm.component.ts file in the src/app folder

```
import { Component, Output, EventEmitter } from "@angular/core";
import { Product } from "../product.model";
import { Model } from "../repository.model";

@Component({
  selector: "paProductForm",
  templateUrl: "productForm.component.html",
})
export class ProductFormComponent {
  newProduct: Product = new Product();

  constructor(private model: Model) {}

  // @Output("paNewProduct")
  // newProductEvent = new EventEmitter<Product>();

  submitForm(form: any) {
    //this.newProductEvent.emit(this.newProduct);
    this.model.saveProduct(this.newProduct);
    this.newProduct = new Product();
    form.resetForm();
  }
}
```

The component that manages the table of product objects used an input property to receive a `Model` object from its parent but must now obtain it directly by declaring a constructor dependency, as shown in listing 18.28.

Listing 18.28. Declaring a model dependency in the productTable.component.ts file in the src/app folder

```
import { Component, Input, Signal, QueryList, ViewChildren,
  ChangeDetectorRef } from "@angular/core";
import { Model } from "../repository.model";
import { Product } from "../product.model";
import { interval } from "rxjs";
import { DiscountService } from "../discount.service";

@Component({
  selector: "paProductTable",
  templateUrl: "productTable.component.html"
})
export class ProductTableComponent {

  constructor(private dataModel: Model) {}

  // @Input({ alias: "model", required: true })
  // dataModel!: Model;

  get Products(): Signal<Product[]> {
    return this.dataModel.Products;
  }

  getProduct(key: number): Product | undefined {
```



```

        return this.dataModel?.getProduct(key);
    }

    deleteProduct(key: number) {
        this.dataModel.deleteProduct(key);
    }

    taxRate: number = 0;
    categoryFilter: string | undefined;
    itemCount: number = 3;
}

```

You will see the same functionality displayed in the browser window when all the changes have been saved and the browser reloads the Angular application—but the way that the functionality is wired up has changed substantially, with each component obtaining the share objects it needs through the dependency injection feature, rather than relying on its parent component to provide it.

18.5 Summary

In this chapter, I explained the problems that dependency injection can be used to address and demonstrated the process of defining and consuming services. I described how services can be used to increase the flexibility in the structure of an application and how dependency injection makes it possible to isolate building blocks so they can be unit tested effectively.

- Services provide access to shared features in a way that doesn't distort the shape of the application.
- Application features declare dependencies on services using constructor parameters.
- Services are registered with the `providers` property in the Angular module.
- Services are decorated with the `@Injectable` decorator, which allows them to declare constructor dependencies on other services.

In the next chapter, I describe modules, which are the final building block for Angular applications.

19

Using and creating modules

This chapter covers

- Understanding the role of modules in an Angular application
- Understanding the root module
- Creating and using feature modules

In this chapter, I describe the last of the Angular building blocks: modules. In the first part of the chapter, I describe the root module, which every Angular application uses to describe the configuration of the application to Angular. In the second part of the chapter, I describe feature modules, which are used to add structure to an application so that related features can be grouped as a single unit. Table 19.1 puts modules in context.

Table 19.1. Putting modules in context

Question	Answer
What are they?	Modules provide configuration information to Angular.
Why are they useful?	The root module describes the application to Angular, setting up essential features such as components and services. Feature modules are useful for adding structure to complex projects, which makes them easier to manage and maintain.
How are they used?	Modules are classes to which the <code>@NgModule</code> decorator has been applied. The properties used by the decorator have different meanings for root and feature modules.
Are there any pitfalls or limitations?	There is no module-wide scope for providers, which means that the providers defined by a feature module will be available as though they had been defined by the root module.

Are there any alternatives?	Every application must have a root module, but the use of feature modules is entirely optional. However, if you don't use feature modules, then the files in an application can become difficult to manage.
-----------------------------	---

Table 19.2 summarizes the chapter.

Table 19.2. Chapter summary

Problem	Solution	Listing
Describing an application and the building blocks it contains	Use the root module	1–7
Grouping related features together	Create a feature module	8–27

19.1 Preparing the example project

As with the other chapters in this part of the book, I am going to use the example project that was created in chapter 9 and has been expanded and extended in each chapter since.

TIP You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/manningbooks/pro-angular-16>. See chapter 1 for how to get help if you have problems running the examples.

To prepare for this chapter, I have removed some functionality from the component templates. Listing 19.1 shows the template for the product table, in which I have commented out the elements for the discount editor and display components.

Listing 19.1. The contents of the `productTable.component.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 Products(); let i = index">
      <td>{{i + 1}}</td>
      <td>{{item.name}}</td>
      <td>{{item.category}}</td>
      <td [pa-price]="item.price" #discount="discount">
        {{ discount.discountAmount | currency:"USD":"symbol"}}
      </td>
      <td class="text-center">
        <button class="btn btn-danger btn-sm"
          (click)="deleteProduct(item.id)">
          Delete
        </button>
      </td>
    </tr>
  </tbody>
</table>
```

```

        </td>
      </tr>
    </tbody>
  </table>

  <!-- <paDiscountEditor></paDiscountEditor> -->
  <!-- <paDiscountDisplay></paDiscountDisplay> -->

```

Listing 19.2 shows the template from the product form component, in which I have commented out the elements that I used to demonstrate the difference between providers for view children and content children.

Listing 19.2. The contents of the productForm.component.html file in the src/app folder

```

<form #form="ngForm" (ngSubmit)="submitForm(form)">
  <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="name" [(ngModel)]="newProduct.price" />
  </div>
  <button class="btn btn-primary mt-2" type="submit">
    Create
  </button>
</form>

<!-- <paDiscountEditor></paDiscountEditor> -->
<!-- <paDiscountDisplay></paDiscountDisplay> -->

```

Run the following command in the `example` folder to start the Angular development tools:

```
ng serve
```

Open a new browser window and navigate to <http://localhost:4200> to see the content shown in figure 19.1.

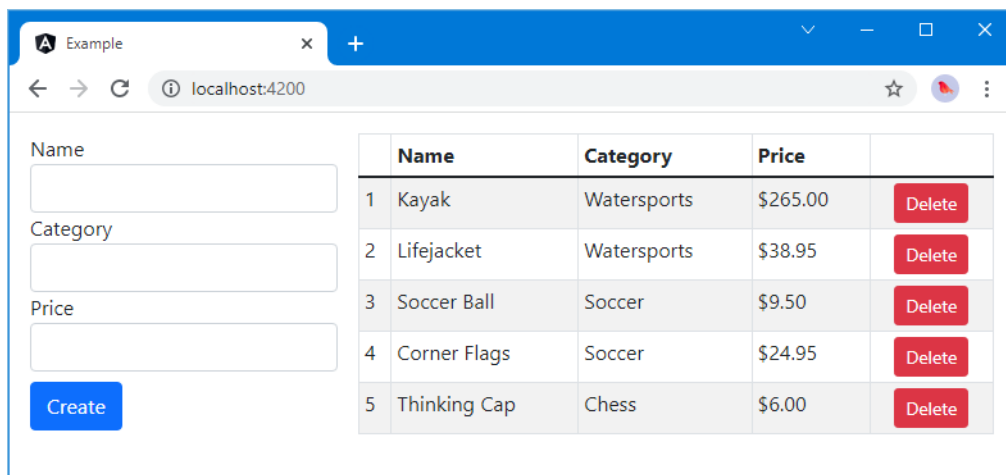


Figure 19.1. Running the example application

19.2 Understanding the root module

Every Angular has at least one module, known as the *root module*. The root module is conventionally defined in a file called `app.module.ts` in the `src/app` folder, and it contains a class to which the `@NgModule` decorator has been applied. Listing 19.3 shows the root module from the example application.

Listing 19.3. The root module in the `app.module.ts` file in the `src/app` folder

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

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

import { ProductComponent } from './component';
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';
import { ProductTableComponent } from './productTable.component';
import { ProductFormComponent } from './productForm.component';
import { PaToggleView } from './toggleView.component';
import { PaAddTaxPipe } from './addTax.pipe';
import { PaCategoryFilterPipe } from './categoryFilter.pipe';

import { LOCALE_ID } from '@angular/core';
import localeFr from '@angular/common/locales/fr';
```

```

import { registerLocaleData } from '@angular/common';

import { PaDiscountDisplayComponent } from "../discountDisplay.component";
import { PaDiscountEditorComponent } from "../discountEditor.component";
import { DiscountService } from "../discount.service";
import { PaDiscountPipe } from "../discount.pipe";
import { PaDiscountAmountDirective } from "../discountAmount.directive";
import { SimpleDataSource } from "../datasource.model";
import { Model } from "../repository.model";

registerLocaleData(localeFr);

@NgModule({
  declarations: [
    ProductComponent, PaAttrDirective, PaModel, PaStructureDirective,
    PaIteratorDirective, PaCellColor, PaCellColorSwitcher,
    ProductTableComponent, ProductFormComponent, PaToggleView,
    PaAddTaxPipe, PaCategoryFilterPipe, PaDiscountDisplayComponent,
    PaDiscountEditorComponent, PaDiscountPipe,
    PaDiscountAmountDirective
  ],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule
  ],
  providers: [DiscountService, SimpleDataSource, Model],
  bootstrap: [ProductComponent]
})
export class AppModule { }

```

There can be multiple modules in a project, but the root module is the one used in the bootstrap file, which is conventionally called `main.ts` and is defined in the `src` folder. Listing 19.4 shows the `main.ts` file for the example project.

Listing 19.4. The Angular bootstrap in the `main.ts` file in the `src` folder

```

import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from '../app/app.module';

platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.error(err));

```

Angular applications can be run in different environments, such as web browsers and native application containers. The job of the bootstrap file is to select the platform and identify the root module. The `platformBrowserDynamic` method creates the browser runtime, and the `bootstrapModule` method is used to specify the module, which is the `AppModule` class from listing 19.3.

Using standalone components

Angular supports creating applications without modules, to streamline the development process. I have not covered standalone components in this book because they are

surprisingly difficult to work with and also because the structure introduced by modules is helpful in all but the simplest projects. Future editions of this book may include standalone components if they mature into something useful but until then, see <https://angular.io/guide/standalone-components> for details.

When defining the root module, the `@NgModule` decorator properties described in table 19.3 are used. (There are additional decorator properties, which are described later in the chapter.)

Table 19.3. The `@NgModule` decorator root module properties

Name	Description
<code>imports</code>	This property specifies the Angular modules that are required to support the directives, components, and pipes in the application.
<code>declarations</code>	This property is used to specify the directives, components, and pipes that are used in the application.
<code>providers</code>	This property defines the services available for use with dependency injection, as described in chapter 18.
<code>bootstrap</code>	This property specifies the root components of the application.

19.2.1 Understanding the `imports` property

The `imports` property is used to list the other modules that the application requires. In the example application, these are all modules provided by the Angular framework.

```
...
imports: [
  BrowserModule,
  BrowserAnimationsModule,
  FormsModule
],
...
```

The `BrowserModule` provides the functionality required to run Angular applications in web browsers. The `BrowserAnimationsModule` module was added to the project by the Angular Material package and enables the features used to animate content, typically in response to user interaction or data changes. The `FormsModule` module contains the features for working with HTML form elements, which are described in part 3.

The `imports` property is also used to declare dependencies on custom modules, which are used to manage complex Angular applications and to create units of reusable functionality. I explain how custom modules are defined in the “Creating feature modules” section.

19.2.2 Understanding the `declarations` property

The `declarations` property is used to provide Angular with a list of the directives, components, and pipes that the application requires, known collectively as the *declarable*

classes. The `declarations` property in the example project root module contains a long list of classes, each of which is available for use elsewhere in the application only because it is listed here.

```
...
declarations: [
  ProductComponent, PaAttrDirective, PaModel, PaStructureDirective,
  PaIteratorDirective, PaCellColor, PaCellColorSwitcher,
  ProductTableComponent, ProductFormComponent, PaToggleView,
  PaAddTaxPipe, PaCategoryFilterPipe, PaDiscountDisplayComponent,
  PaDiscountEditorComponent, PaDiscountPipe,
  PaDiscountAmountDirective
],
...
```

Notice that the built-in declarable classes, such as the directives pipes described in earlier chapters, are not included in the `declarations` property for the root module. This is because they are part of the `BrowserModule` module, and when you add a module to the `imports` property, its declarable classes are automatically available for use in the application.

19.2.3 Understanding the providers property

The `providers` property is used to define the service providers that will be used to resolve dependencies. The use of services is described in chapters 18.

19.2.4 Understanding the bootstrap property

The `bootstrap` property specifies the root component or components for the application. When Angular processes the main HTML document, which is conventionally called `index.html`, it inspects the root components and applies them using the value of the `selector` property in the `@Component` decorators.

TIP The components listed in the `bootstrap` property must also be included in the `declarations` list.

Here is the `bootstrap` property from the example project's root module:

```
...
bootstrap: [ProductComponent]
...
```

The `ProductComponent` class provides the root component, and its `selector` property specifies the app element, as shown in listing 19.5.

Listing 19.5. The root component in the `component.ts` file in the `src/app` folder

```
import { Component, computed } from "@angular/core";

@Component({
  selector: "app",
  templateUrl: "template.html"
})
export class ProductComponent {

}
```


When I started the example project in chapter 9, the root component had a lot of functionality. But since the introduction of additional components, the role of this component has been reduced, and it is now essentially a placeholder that tells Angular to project the contents of the `app/template.html` file into the `app` element in the HTML document, which allows the components that do the real work in the application to be loaded.

There is nothing wrong with this approach, but it does mean the root component in the application doesn't have a great deal to do. If this kind of redundancy feels untidy, then you can specify multiple root components in the root module, and all of them will be used to target elements in the HTML document. To demonstrate, I have removed the existing root component from the root module's `bootstrap` property and replaced it with the component classes that are responsible for the product form and the product table, as shown in listing 19.6.

Listing 19.6. Specifying multiple root components in the `app.module.ts` file in the `src/app` folder

```
...
@NgModule({
  declarations: [
    ProductComponent, PaAttrDirective, PaModel, PaStructureDirective,
    PaIteratorDirective, PaCellColor, PaCellColorSwitcher,
    ProductTableComponent, ProductFormComponent, PaToggleView,
    PaAddTaxPipe, PaCategoryFilterPipe, PaDiscountDisplayComponent,
    PaDiscountEditorComponent, PaDiscountPipe,
    PaDiscountAmountDirective
  ],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule
  ],
  providers: [DiscountService, SimpleDataSource, Model],
  bootstrap: [ProductFormComponent, ProductTableComponent]
})
export class AppModule { }
...
```

Listing 19.7 reflects the change in the root components in the main HTML document. The inconsistent element names are from an earlier chapter, where I changed the selector for the form component to demonstrate the use of the shadow DOM feature.

Listing 19.7. Changing the root component elements in the `index.html` file in the `src` 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&dis
lay=swap" rel="stylesheet">
<link href="https://fonts.googleapis.com/icon?family=Material+Icons"
rel="stylesheet">
</head>
<body class="container-fluid">
  <div class="row">
    <div class="col-8 p-2">
      <paProductTable></paProductTable>
    </div>
    <div class="col-4 p-2">
      <paProductForm></paProductForm>
    </div>
  </div>
</body>
</html>

```

I have reversed the order in which these components appear compared to previous examples, just to create a detectable change in the application's layout. When all the changes are saved and the browser has reloaded the page, you will see the new root components displayed, as illustrated by figure 19.2. (You may have to manually reload the browser or restart the Angular development tools to see the change).

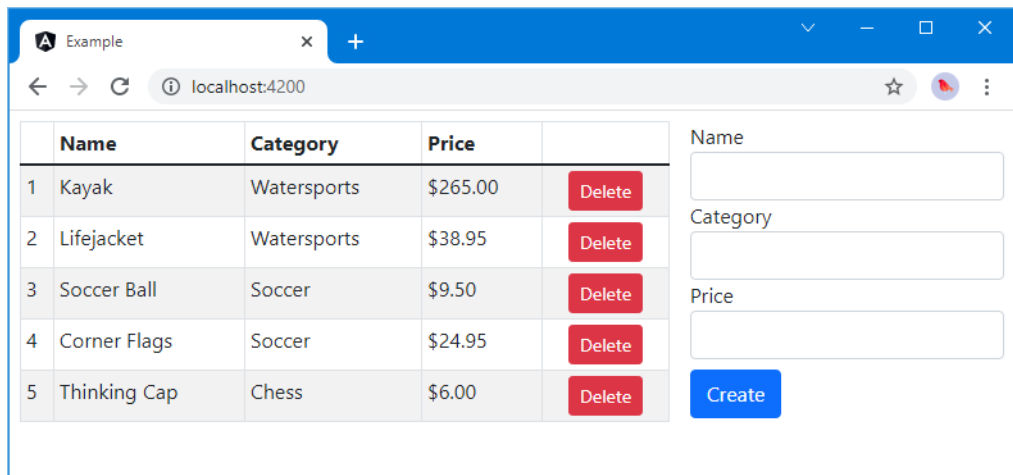


Figure 19.2. Using multiple root components

The module's service providers are used to resolve dependencies for all root components. In the case of the example application, this means there is a single `Model` service object that is shared throughout the application and that allows products created with the HTML form to be displayed automatically in the table, even though these components have been promoted to be root components.

19.3 Creating feature modules

The root module has become increasingly complex as I added features in earlier chapters, with a long list of `import` statements to load JavaScript modules and a set of classes in the `declarations` property of the `@NgModule` decorator that spans several lines, as shown in listing 19.8.

Listing 19.8. The contents of the `app.module.ts` file in the `src/app` folder

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

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

import { ProductComponent } from './component';
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';
import { ProductTableComponent } from './productTable.component';
import { ProductFormComponent } from './productForm.component';
import { PaToggleView } from './toggleView.component';
import { PaAddTaxPipe } from './addTax.pipe';
import { PaCategoryFilterPipe } from './categoryFilter.pipe';

import { LOCALE_ID } from '@angular/core';
import localeFr from '@angular/common/locales/fr';
import { registerLocaleData } from '@angular/common';

import { PaDiscountDisplayComponent } from './discountDisplay.component';
import { PaDiscountEditorComponent } from './discountEditor.component';
import { DiscountService } from './discount.service';
import { PaDiscountPipe } from './discount.pipe';
import { PaDiscountAmountDirective } from './discountAmount.directive';
import { SimpleDataSource } from './datasource.model';
import { Model } from './repository.model';

registerLocaleData(localeFr);

@NgModule({
  declarations: [
    ProductComponent, PaAttrDirective, PaModel, PaStructureDirective,
    PaIteratorDirective, PaCellColor, PaCellColorSwitcher,
    ProductTableComponent, ProductFormComponent, PaToggleView,
    PaAddTaxPipe, PaCategoryFilterPipe, PaDiscountDisplayComponent,
    PaDiscountEditorComponent, PaDiscountPipe,
    PaDiscountAmountDirective
  ],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
```

```

    FormsModule
  ],
  providers: [DiscountService, SimpleDataSource, Model],
  bootstrap: [ProductFormComponent, ProductTableComponent]
})
export class AppModule { }

```

Feature modules are used to group related functionality so that it can be used as a single entity, just like the Angular modules such as `BrowserModule`. When I need to use the features for working with forms, for example, I don't have to add `import` statements and `declarations` entries for each directive, component, or pipe. Instead, I just add `BrowserModule` to the decorator's `imports` property, and all of the functionality it contains is available throughout the application.

When you create a feature module, you can choose to focus on an application function or elect to group a set of related building blocks that provide your application's infrastructure. I'll do both in the sections that follow because they work in slightly different ways and have different considerations. Feature modules use the same `@NgModule` decorator but with an overlapping set of configuration properties, some of which are new and some of which are used in common with the root module but have a different effect. I explain how these properties are used in the following sections, but table 19.4 provides a summary for quick reference.

Table 19.4. The `@NgModule` decorator properties for feature modules

Name	Description
<code>imports</code>	This property is used to import the modules that are required by the classes in the modules.
<code>providers</code>	This property is used to define the module's services. When the feature module is loaded, the set of services is combined with those in the root module, which means that the feature module's services are available throughout the application (and not just within the module).
<code>declarations</code>	This property is used to specify the directives, components, and pipes in the module. This property must contain the classes that are used within the module and those that are exposed by the module to the rest of the application.
<code>exports</code>	This property is used to define the public exports from the module. It contains some or all of the directives, components, and pipes from the <code>declarations</code> property and some or all of the modules from the <code>imports</code> property.

19.3.1 Creating a model module

The term *model module* might be a tongue twister, but it is generally a good place to start when refactoring an application using feature modules because just about every other building block in the application depends on the model.

The first step is to create the folder that will contain the module. Module folders are defined within the `src/app` folder and are given a meaningful name. For this module, I created an `src/app/model` folder by running the following command in the `example` folder:

```
mkdir src/app/model
```

The naming conventions used for Angular files make it easy to move and delete multiple files. Run the following command in the `example` folder to move the files (they will work in Windows PowerShell, Linux, and macOS):

```
mv src/app/*.model.ts src/app/model/
```

The result is that the files listed in table 19.5 are moved to the `model` folder.

Table 19.5. The file moves required for the module

File	New Location
<code>src/app/datasource.model.ts</code>	<code>src/app/model/datasource.model.ts</code>
<code>src/app/product.model.ts</code>	<code>src/app/model/product.model.ts</code>
<code>src/app/repository.model.ts</code>	<code>src/app/model/repository.model.ts</code>
<code>src/app/ticker.model.ts</code>	<code>src/app/model/ticker.model.ts</code>

If you try to build the project once you have moved the files, the TypeScript compiler will list a series of compiler errors because some of the key declarable classes are unavailable. I'll deal with these problems shortly.

CREATING THE MODULE DEFINITION

The next step is to define a module that brings together the functionality in the files that have been moved to the new folder. I added a file called `model.module.ts` in the `src/app/model` folder and defined the module shown in listing 19.9.

Listing 19.9. The contents of the `model.module.ts` file in the `src/app/model` folder

```
import { NgModule } from "@angular/core";
import { SimpleDataSource } from "../datasource.model";
import { Model } from "../repository.model";

@NgModule({
  providers: [Model, SimpleDataSource]
})
export class ModelModule { }
```

The purpose of a feature module is to selectively expose the contents of the folder to the rest of the application. The `@NgModule` decorator for this module uses only the `providers` property to define class providers for the `Model` and `SimpleDataSource` services. When you use providers in a feature module, they are registered with the root module's injector, which means they are available throughout the application, which is exactly what is required for the data model in the example application.

TIP A common mistake is to assume that services defined in a module are accessible only to the classes within that module. There is no module scope in Angular. Providers defined by a feature module are used as though they were defined by the root module. Local providers defined by directives and components in the feature module are available to their view and content children even if they are defined in other modules.

UPDATING THE OTHER CLASSES IN THE APPLICATION

Moving classes into the `model` folder has broken `import` statements in other parts of the application. The next step is to update those `import` statements to point to the new module. There are four affected files: `attr.directive.ts`, `categoryFilter.pipe.ts`, `productForm.component.ts`, and `productTable.component.ts`. Listing 19.10 shows the changes required to the `attr.directive.ts` file.

Listing 19.10. Updating the import 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 "../model/product.model";

@Directive({
  selector: "[pa-attr]"
})
export class PaAttrDirective {

  @Input("pa-attr")
  @HostBinding("class")
  bgClass: string | null = "";

  @Input("pa-product")
  product: Product = new Product();

  @Output("pa-category")
  click = new EventEmitter<string>();

  @HostListener("click")
  triggerCustomEvent() {
    if (this.product != null) {
      this.click.emit(this.product.category);
    }
  }
}
```

The only change that is required is to update the path used in the `import` statement to reflect the new location of the code file. Listing 19.11 shows the same change applied to the `categoryFilter.pipe.ts` file.

Listing 19.11. Updating the import in the `categoryFilter.pipe.ts` file in the `src/app` folder

```
import { Pipe } from "@angular/core";
import { Product } from "../model/product.model";

@Pipe({
  name: "filter",
```

```

        pure: false
    })
    export class PaCategoryFilterPipe {
        transform(products: Product[] | undefined,
            category: string | undefined): Product[] {
            if (products == undefined) {
                return [];
            }
            return category == undefined ?
                products : products.filter(p => p.category == category);
        }
    }
}

```

Listing 19.12 updates the import statements in the `productForm.component.ts` file.

Listing 19.12. Updating paths in the `productForm.component.ts` file in the `src/app` folder

```

import { Component, Output, EventEmitter } from "@angular/core";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";

@Component({
    selector: "paProductForm",
    templateUrl: "productForm.component.html",
})
export class ProductFormComponent {
    newProduct: Product = new Product();

    constructor(private model: Model) {}

    submitForm(form: any) {
        this.model.saveProduct(this.newProduct);
        this.newProduct = new Product();
        form.resetForm();
    }
}

```

Listing 19.13 updates the paths in the final file, `productTable.component.ts`.

Listing 19.13. Updating paths in the `productTable.component.ts` file in the `src/app` folder

```

import { Component, Input, Signal, QueryList, ViewChildren,
    ChangeDetectorRef } from "@angular/core";
import { Model } from "../model/repository.model";
import { Product } from "../model/product.model";
import { interval } from "rxjs";
import { DiscountService } from "../discount.service";

@Component({
    selector: "paProductTable",
    templateUrl: "productTable.component.html"
})
export class ProductTableComponent {

    // ...statements omitted for brevity...
}

```

Using a JavaScript module with an Angular module

Creating an Angular module allows related application features to be grouped together but still requires that each one is imported from its own file when it is needed elsewhere in the application, as you have seen in the listings in this section.

You can also define a JavaScript module that exports the public-facing features of the Angular module so they can be accessed with the same kind of `import` statement that is used for the `@angular/core` module, for example. To use a JavaScript module, add a file called `index.ts` alongside the TypeScript file that defines the Angular module, which is the `src/app/model` folder for the examples in this section. For each of the application features that you want to use outside of the application, add an `export...from` statement, like this:

```
...
export { ModelModule } from "./model.module";
export { Product } from "./product.model";
export { SimpleDataSource } from "./datasource.model";
export { Model } from "./repository.model";
...
```

These statements export the contents of the individual TypeScript files. You can then import the features you require without having to specify individual files, like this:

```
...
import { Product, Model } from "./model";
...
```

Using the filename `index.ts` means that you only have to specify the name of the folder in the `import` statement, producing a result that is neater and more consistent with the Angular core packages.

That said, I don't use this technique in my own projects. Using an `index.ts` file means you have to remember to add every feature to both the Angular and JavaScript modules, which is an extra step that I often forget to do. Instead, I use the approach shown in this chapter and import directly from the files that contain the application's features.

UPDATING THE ROOT MODULE

The final step is to update the root module so that the services defined in the feature module are made available throughout the application. Listing 19.14 shows the required changes.

Listing 19.14. Updating the root module in the `app.module.ts` file in the `src/app` folder

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

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



```

import { ProductComponent } from './component';
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';
import { ProductTableComponent } from './productTable.component';
import { ProductFormComponent } from './productForm.component';
import { PaToggleView } from './toggleView.component';
import { PaAddTaxPipe } from './addTax.pipe';
import { PaCategoryFilterPipe } from './categoryFilter.pipe';

import { LOCALE_ID } from '@angular/core';
import localeFr from '@angular/common/locales/fr';
import { registerLocaleData } from '@angular/common';

import { PaDiscountDisplayComponent } from './discountDisplay.component';
import { PaDiscountEditorComponent } from './discountEditor.component';
import { DiscountService } from './discount.service';
import { PaDiscountPipe } from './discount.pipe';
import { PaDiscountAmountDirective } from './discountAmount.directive';
// import { SimpleDataSource } from './datasource.model';
// import { Model } from './repository.model';
import { ModelModule } from './model/model.module';

registerLocaleData(localeFr);

@NgModule({
  declarations: [
    ProductComponent, PaAttrDirective, PaModel, PaStructureDirective,
    PaIteratorDirective, PaCellColor, PaCellColorSwitcher,
    ProductTableComponent, ProductFormComponent, PaToggleView,
    PaAddTaxPipe, PaCategoryFilterPipe, PaDiscountDisplayComponent,
    PaDiscountEditorComponent, PaDiscountPipe,
    PaDiscountAmountDirective
  ],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule,
    ModelModule
  ],
  providers: [DiscountService],
  bootstrap: [ProductFormComponent, ProductTableComponent]
})
export class AppModule { }

```

I imported the feature module and added it to the root module's imports list. Since the feature module defines providers for `Model` and `SimpleDataSource`, I removed the entries from the root module's providers list and removed the associated import statements.

Once you have saved the changes, you can run `ng serve` to start the Angular development tools. The application will compile, and the revised root module will provide access to the model service. There are no visible changes to the content displayed in the browser, and the changes

are limited to the structure of the project. (You may need to restart the Angular development tools and reload the browser to see the changes.)

19.3.2 Creating a utility feature module

A model module is a good place to start because it demonstrates the basic structure of a feature module and how it relates to the root module. The impact on the application was slight, however, and not a great deal of simplification was achieved.

The next step up in complexity is a utility feature module, which groups together all of the common functionality in the application, such as pipes and directives. In a real project, you might be more selective about how you group these types of building blocks together so that there are several modules, each containing similar functionality. For the example application, I am going to move all of the pipes, directives, and services into a single module.

CREATING THE MODULE FOLDER AND MOVING THE FILES

As with the previous module, the first step is to create the folder. For this module, I created a folder called `src/app/common` and moved code files for the pipes and directives by running the following commands in the `example` folder:

```
mkdir src/app/common
mv src/app/*.pipe.ts src/app/common/
mv src/app/*.directive.ts src/app/common/
```

These commands should work in Windows PowerShell, Linux, and macOS. Some of the directives and pipes in the application rely on services provided to them through dependency injection. Run the following command in the `example` folder to move the TypeScript file for the service into the module folder:

```
mv src/app/*.service.ts src/app/common/
```

The result is that the files listed in table 19.6 are moved to the `common` module folder.

Table 19.6. The file moves required for the module

File	New Location
app/addTax.pipe.ts	app/common/addTax.pipe.ts
app/attr.directive.ts	app/common/attr.directive.ts
app/categoryFilter.pipe.ts	app/common/categoryFilter.pipe.ts
app/cellColor.directive.ts	app/common/cellColor.directive.ts
app/cellColorSwitcher.directive.ts	app/common/cellColorSwitcher.directive.ts
app/discount.pipe.ts	app/common/discount.pipe.ts
app/discount.service.ts	app/common/discount.service.ts
app/discountAmount.directive.ts	app/common/discountAmount.directive.ts

app/iterator.directive.ts	app/common/iterator.directive.ts
app/structure.directive.ts	app/common/structure.directive.ts
app/twoway.directive.ts	app/common/twoway.directive.ts

UPDATING THE CLASSES IN THE NEW MODULE

Some of the classes that have been moved into the new folder have `import` statements that have to be updated to reflect the new path to the model module. Listing 19.15 shows the change required to the `attr.directive.ts` file.

Listing 19.15. Updating imports in the `attr.directive.ts` file in the `src/app/common` folder

```
import { Directive, ElementRef, Input, SimpleChanges, Output,
        EventEmitter, HostListener, HostBinding } from "@angular/core";
import { Product } from "../model/product.model";

@Directive({
  selector: "[pa-attr]"
})
export class PaAttrDirective {

  // ...statements omitted for brevity...
}
```

Listing 19.16 shows the corresponding change to the `categoryFilter.pipe.ts` file.

Listing 19.16. Updating the `categoryFilter.pipe.ts` file in the `src/app/common` folder

```
import { Pipe } from "@angular/core";
import { Product } from "../model/product.model";

@Pipe({
  name: "filter",
  pure: false
})
export class PaCategoryFilterPipe {

  transform(products: Product[] | undefined,
            category: string | undefined): Product[] {
    if (products == undefined) {
      return [];
    }
    return category == undefined ?
      products : products.filter(p => p.category == category);
  }
}
```

CREATING THE MODULE DEFINITION

The next step is to define a module that brings together the functionality in the files that have been moved to the new folder. I added a file called `common.module.ts` in the `src/app/common` folder and defined the module shown in listing 19.17.

Listing 19.17. The contents of the `common.module.ts` file in the `src/app/common` folder

```

import { NgModule } from "@angular/core";
import { PaAddTaxPipe } from "../addTax.pipe";
import { PaAttrDirective } from "../attr.directive";
import { PaCategoryFilterPipe } from "../categoryFilter.pipe";
import { PaCellColor } from "../cellColor.directive";
import { PaCellColorSwitcher } from "../cellColorSwitcher.directive";
import { PaDiscountPipe } from "../discount.pipe";
import { PaDiscountAmountDirective } from "../discountAmount.directive";
import { PaIteratorDirective } from "../iterator.directive";
import { PaStructureDirective } from "../structure.directive";
import { PaModel } from "../twoway.directive";
import { DiscountService } from "../discount.service";
import { ModelModule } from "../model/model.module";

@NgModule({
  imports: [ModelModule],
  providers: [DiscountService],
  declarations: [PaAddTaxPipe, PaAttrDirective, PaCategoryFilterPipe,
    PaCellColor, PaCellColorSwitcher, PaDiscountPipe,
    PaDiscountAmountDirective, PaIteratorDirective,
    PaStructureDirective, PaModel],
  exports: [PaAddTaxPipe, PaAttrDirective, PaCategoryFilterPipe,
    PaCellColor, PaCellColorSwitcher, PaDiscountPipe,
    PaDiscountAmountDirective, PaIteratorDirective,
    PaStructureDirective, PaModel]
})
export class CommonModule { }

```

This is a more complex module than the one required for the data model. In the sections that follow, I describe the values that are used for each of the decorator's properties.

UNDERSTANDING THE IMPORTS

Some of the directives and pipes in the module depend on the services defined in the `model` module, created earlier in this chapter. To ensure that the features in that module are available, I have added to the common module's `imports` property.

UNDERSTANDING THE PROVIDERS

The `providers` property ensures that the services, directives, and pipes in the feature module have access to the services they require. This means adding class providers to create `LogService` and `DiscountService` services, which will be added to the root module's providers when the module is loaded. Not only will the services be available to the directives and pipes in the `common` module; they will also be available throughout the application.

UNDERSTANDING THE DECLARATIONS

The `declarations` property is used to provide Angular with a list of the directives and pipes (and components, if there are any) in the module. In a feature module, this property has two purposes: it enables the declarable classes for use in any templates contained within the module, and it allows a module to make those declarable classes available outside of the module. I create a module that contains template content later in this chapter, but for this module, the value of the `declarations` property is that it must be used to prepare for the `exports` property, described in the next section.

UNDERSTANDING THE EXPORTS

For a module that contains directives and pipes intended for use elsewhere in the application, the `exports` property is the most important in the `@NgModule` decorator because it defines the set of directives, components, and pipes that the module provides for use when it is imported elsewhere in the application. The `exports` property can contain individual classes and module types, although both must already be listed in the `declarations` or `imports` property. When the module is imported, the types listed behave as though they had been added to the importing module's `declarations` property.

UPDATING THE OTHER CLASSES IN THE APPLICATION

Now that the module has been defined, I can update the other files in the application that contain `import` statements for the types that are now part of the `common` module. Listing 19.18 shows the changes required to the `discountDisplay.component.ts` file.

Listing 19.18. Updating the `discountDisplay.component.ts` file in the `src/app` folder

```
import { Component, Input } from "@angular/core";
import { DiscountService } from "../common/discount.service";

@Component({
  selector: "paDiscountDisplay",
  template: `<div class="bg-info text-white p-2 my-2">
    The discount is {{discounter.discount }}
  </div>`
})
export class PaDiscountDisplayComponent {

  constructor(public discounter: DiscountService) { }
}
```

Listing 19.19 shows the changes to the `discountEditor.component.ts` file.

Listing 19.19. Updating the import in the `discountEditor.component.ts` file in the `src/app` folder

```
import { Component, Input } from "@angular/core";
import { DiscountService } from "../common/discount.service";

@Component({
  selector: "paDiscountEditor",
  template: `<div class="form-group">
    <label>Discount</label>
    <input [(ngModel)]="discounter.discount"
      class="form-control" type="number" />
  </div>`
})
export class PaDiscountEditorComponent {

  constructor(public discounter: DiscountService) { }
}
```

The next change is to the `productTable.component.ts` file, as shown in listing 19.20. This component's code file contains an import statement for `DiscountService` but doesn't use

the service's functionality. An import statement that is not used doesn't produce an error, but any import statement to a non-existent file needs to be updated or removed.

Listing 19.20. Removing the import in the `productTable.component.ts` file in the `src/app` folder

```
import { Component, Input, Signal, QueryList, ViewChildren,
        ChangeDetectorRef } from "@angular/core";
import { Model } from "../model/repository.model";
import { Product } from "../model/product.model";
import { interval } from "rxjs";
// import { DiscountService } from "../common/discount.service";

@Component({
  selector: "paProductTable",
  templateUrl: "productTable.component.html"
})
export class ProductTableComponent {

  constructor(private dataModel: Model) {}

  get Products(): Signal<Product[]> {
    return this.dataModel.Products;
  }

  getProduct(key: number): Product | undefined {
    return this.dataModel?.getProduct(key);
  }

  deleteProduct(key: number) {
    this.dataModel.deleteProduct(key);
  }

  taxRate: number = 0;
  categoryFilter: string | undefined;
  itemCount: number = 3;
}
```

UPDATING THE ROOT MODULE

The final step is to update the root module so that it loads the `common` module to provide access to the directives and pipes it contains, as shown in listing 19.21.

Listing 19.21. Importing a feature module in the `app.module.ts` file in the `src/app` folder

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

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

import { ProductComponent } from './component';
// 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";
import { ProductTableComponent } from "../productTable.component";
import { ProductFormComponent } from "../productForm.component";
import { PaToggleView } from "../toggleView.component";
// import { PaAddTaxPipe } from '../addTax.pipe';
// import { PaCategoryFilterPipe } from '../categoryFilter.pipe';

import { LOCALE_ID } from "@angular/core";
import localeFr from '@angular/common/locales/fr';
import { registerLocaleData } from '@angular/common';

import { PaDiscountDisplayComponent } from "../discountDisplay.component";
import { PaDiscountEditorComponent } from "../discountEditor.component";
// import { DiscountService } from "../discount.service";
// import { PaDiscountPipe } from "../discount.pipe";
// import { PaDiscountAmountDirective } from "../discountAmount.directive";
import { ModelModule } from '../model/model.module';
import { CommonModule } from "../common/common.module";

registerLocaleData(localeFr);

@NgModule({
  declarations: [
    ProductComponent, ProductTableComponent, ProductFormComponent,
    PaToggleView, PaDiscountDisplayComponent, PaDiscountEditorComponent
  ],
  imports: [
    BrowserModule,
    BrowserAnimationsModule,
    FormsModule,
    ModelModule,
    CommonModule
  ],
  //providers: [DiscountService],
  bootstrap: [ProductFormComponent, ProductTableComponent]
})
export class AppModule { }

```

The root module has been substantially simplified with the creation of the `common` module, which has been added to the `imports` list. All of the individual classes for directives and pipes have been removed from the `declarations` list, and their associated `import` statements have been removed from the file. When the `common` module is imported, all of the types listed in its `exports` property will be added to the root module's `declarations` property.

Once you have saved the changes in this section, you can run the `ng serve` command to start the Angular development tools. Once again, there is no visible change in the content presented to the user, and the differences are all in the structure of the application.

19.3.3 Creating a feature module with components

The final module that I am going to create will contain the application's components. The process for creating the module is the same as in the previous examples, as described in the sections that follow.

CREATING THE MODULE FOLDER AND MOVING THE FILES

The module will be called `components`, and I created the folder `src/app/components` to contain the files. Run the following commands in the `example` folder to create the folder; move the directive TypeScript, HTML, and CSS files into the new folder; and delete the corresponding JavaScript files:

```
mkdir src/app/components
mv src/app/*.component.ts src/app/components/
mv src/app/*.component.html src/app/components/
mv src/app/*.component.css src/app/components/
```

The result of these commands is that the component code files, templates, and style sheets are moved into the new folder, as listed in table 19.7.

Table 19.7. The file moves required for the component module

File	New Location
<code>src/app/app.component.ts</code>	<code>src/app/components/app.component.ts</code>
<code>src/app/app.component.html</code>	<code>src/app/components/app.component.html</code>
<code>src/app/app.component.css</code>	<code>src/app/components/app.component.css</code>
<code>src/app</code> <code>/discountDisplay.component.ts</code>	<code>src/app/components</code> <code>/discountDisplay.component.ts</code>
<code>src/app/discountEditor.component.ts</code>	<code>src/app/components/discountEditor.component.ts</code>
<code>src/app/productForm.component.ts</code>	<code>src/app/components/productForm.component.ts</code>
<code>src/app/productForm.component.html</code>	<code>src/app/components/productForm.component.html</code>
<code>src/app/productForm.component.css</code>	<code>src/app/components/productForm.component.css</code>
<code>src/app/productTable.component.ts</code>	<code>src/app/components/productTable.component.ts</code>
<code>src/app/productTable.component.html</code>	<code>src/app/components/productTable.component.html</code>
<code>src/app/toggleView.component.ts</code>	<code>src/app/components/toggleView.component.ts</code>
<code>src/app/toggleView.component.html</code>	<code>src/app/components/toggleView.component.html</code>

CREATING THE MODULE DEFINITION

To create the module, I added a file called `components.module.ts` to the `src/app/components` folder and added the statements shown in listing 19.22.

Listing 19.22. The `components.module.ts` file in the `src/app/components` folder

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { CommonModule } from "../common/common.module";
import { FormsModule } from "@angular/forms"
import { PaDiscountDisplayComponent } from "../discountDisplay.component";
import { PaDiscountEditorComponent } from "../discountEditor.component";
import { ProductFormComponent } from "../productForm.component";
import { ProductTableComponent } from "../productTable.component";

@NgModule({
  imports: [BrowserModule, FormsModule, CommonModule],
  declarations: [PaDiscountDisplayComponent, PaDiscountEditorComponent,
    ProductFormComponent, ProductTableComponent],
  exports: [ProductFormComponent, ProductTableComponent]
})
export class ComponentsModule { }
```

This module exports the `ProductFormComponent` and `ProductTableComponent` components, which are the two components used in the root component's `bootstrap` property. The other components are private to the module.

UPDATING THE OTHER CLASSES

Moving the TypeScript files into the `components` folder requires some changes to the paths in the `import` statements. Listing 19.23 shows the change required for the `discountDisplay.component.ts` file.

Listing 19.23. Updating the `discountDisplay.component.ts` file in the `src/app/component` folder

```
import { Component, Input } from "@angular/core";
import { DiscountService } from "../common/discount.service";

@Component({
  selector: "paDiscountDisplay",
  template: `<div class="bg-info text-white p-2 my-2">
    The discount is {{discounter.discount}}
  </div>`
})
export class PaDiscountDisplayComponent {

  constructor(public discounter: DiscountService) { }
```

Listing 19.24 shows the change required to the `discountEditor.component.ts` file.

Listing 19.24. Updating the `discountEditor.component.ts` file in the `src/app/component` folder

```
import { Component, Input } from "@angular/core";
import { DiscountService } from "../common/discount.service";
```

```

@Component({
  selector: "paDiscountEditor",
  template: `<div class="form-group">
    <label>Discount</label>
    <input [(ngModel)]="discounter.discount"
      class="form-control" type="number" />
  </div>`
})
export class PaDiscountEditorComponent {

  constructor(public discounter: DiscountService) { }

}

```

Listing 19.25 shows the changes required for the `productForm.component.ts` file.

Listing 19.25. Updating the `productForm.component.ts` file in the `src/app/component` folder

```

import { Component, Output, EventEmitter } from "@angular/core";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";

@Component({
  selector: "paProductForm",
  templateUrl: "productForm.component.html",
})
export class ProductFormComponent {
  newProduct: Product = new Product();

  constructor(private model: Model) {}

  submitForm(form: any) {
    this.model.saveProduct(this.newProduct);
    this.newProduct = new Product();
    form.resetForm();
  }
}

```

Listing 19.26 shows the changes required to the `productTable.component.ts` file.

Listing 19.26. Updating the `productTable.component.ts` file in the `src/app/component` folder

```

import { Component, Input, Signal, QueryList, ViewChildren,
  ChangeDetectorRef } from "@angular/core";
import { Model } from "../model/repository.model";
import { Product } from "../model/product.model";
import { interval } from "rxjs";
//import { DiscountService } from "../discount.service";

@Component({
  selector: "paProductTable",
  templateUrl: "productTable.component.html"
})
export class ProductTableComponent {

  constructor(private dataModel: Model) {}

```

```

    get Products(): Signal<Product[]> {
        return this.dataModel.Products;
    }

    getProduct(key: number): Product | undefined {
        return this.dataModel?.getProduct(key);
    }

    deleteProduct(key: number) {
        this.dataModel.deleteProduct(key);
    }

    taxRate: number = 0;
    categoryFilter: string | undefined;
    itemCount: number = 3;
}

```

UPDATING THE ROOT MODULE

The final step is to update the root module to remove the outdated references to the individual files and to import the new module, as shown in listing 19.27.

Listing 19.27. Importing a feature module in the app.module.ts file in the src/app folder

```

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

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

import { ProductComponent } from './component';
import { ProductTableComponent }
    from './components/productTable.component';
import { ProductFormComponent } from './components/productForm.component';
// import { PaToggleView } from './toggleView.component';

import { LOCALE_ID } from '@angular/core';
import localeFr from '@angular/common/locales/fr';
import { registerLocaleData } from '@angular/common';

// import { PaDiscountDisplayComponent } from
// './discountDisplay.component';
// import { PaDiscountEditorComponent } from './discountEditor.component';
import { ModelModule } from './model/model.module';
import { CommonModule } from './common/common.module';
import { ComponentsModule } from './components/components.module';

registerLocaleData(localeFr);

@NgModule({
    declarations: [ProductComponent],
    imports: [
        BrowserModule,
        BrowserAnimationsModule,
        FormsModule,

```

```

    ModelModule,
    CommonModule,
    ComponentsModule
  ],
  //providers: [DiscountService],
  bootstrap: [ProductFormComponent, ProductTableComponent]
})
export class AppModule { }

```

Restart the Angular development tools to build and display the application. Adding modules to the application has radically simplified the root module and allows related features to be defined in self-contained blocks, which can be extended or modified in relative isolation from the rest of the application.

19.4 Summary

In this chapter, I described the last of the Angular building blocks: modules. I explained the role of the root module and demonstrated how to create feature modules to add structure to an application.

- Modules introduce structure into a project so that related features can be grouped.
- Modules define the dependencies for a related group of features, the services those features use, and determine which of the features will be available for use in the rest of the application.
- The root module is used to configure the overall application, but additional feature modules can be used to structure complex applications.

In the next part of the book, I describe the features that Angular provides to shape the building blocks into complex and responsive applications.

Part III

20

Creating the example project

Throughout the chapters in the previous part of the book, I added classes and content to the example project to demonstrate different Angular features and then, in chapter 19, introduced feature modules to add some structure to the project. The result is a project with a lot of redundant and unused functionality, and for this part of the book, I am going to start a new project that takes some of the core features from earlier chapters and provides a clean foundation on which to build 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/manningbooks/pro-angular-16>. See chapter 1 for how to get help if you have problems running the examples.

20.1 Starting the example project

To create the project with tools and placeholder content, open a new command prompt, navigate to a convenient location, and run the command shown in listing 20.1.

Listing 20.1. Creating the example project

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

To distinguish the project used in this part of the book from earlier examples, I created a project called `exampleApp`. The project initialization process will take a while to complete as all the required packages are downloaded.

20.1.1 Adding and configuring the Bootstrap CSS package

I continue to use the Bootstrap CSS framework to style the HTML elements in this chapter and the rest of the book. Run the command shown in listing 20.2 in the `exampleApp` folder to add the Bootstrap package to the project.

Listing 20.2. Adding a package to the project

```
npm install bootstrap@5.2.3
```

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. For Linux, run the command shown in listing 20.3 in the `exampleApp` folder. Take care to enter the command exactly as shown and do not introduce additional spaces or quotes.

Listing 20.3. Changing the application configuration

```
ng config projects.exampleApp.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 20.4 in the `exampleApp` folder.

Listing 20.4. Changing the application configuration using PowerShell

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

20.1.2 Creating the project structure

Create the folders shown in table 20.1 in preparation for the feature modules that the example project will contain.

Table 20.1. The folders required for the example application

Name	Description
<code>src/app/model</code>	This folder will contain a feature module containing the data model.
<code>src/app/core</code>	This folder will contain a feature module containing components that provide the core features of the application.
<code>src/app/messages</code>	This folder will contain a feature module that is used to display messages and errors to the user.

20.2 Creating the model module

The first feature module will contain the project's data model, which is similar to the one used in part 2.

20.2.1 Creating the product data type

To define the basic data type around which the application is based, I added a file called `product.model.ts` to the `src/app/model` folder and defined the class shown in listing 20.5.

Listing 20.5. 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 price?: number) { }

    static fromProduct(p: Product) {
        return new Product(p.id, p.name, p.category, p.price);
    }
}

```

20.2.2 Creating the data source and repository

To provide the application with some initial data, I created a file called `static.datasource.ts` in the `src/app/model` folder and defined the service shown in listing 20.6. This class will be used as the data source until I explain how to use asynchronous HTTP requests to request data from web services.

TIP I am more relaxed about following the name conventions for Angular files when creating files within a feature module, especially if the purpose of the module is obvious from its name.

Listing 20.6. The contents of the `static.datasource.ts` file in the `src/app/model` folder

```

import { Injectable } from "@angular/core";
import { Product } from "../product.model";

@Injectable()
export class StaticDataSource {
    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 next step is to define the repository, through which the rest of the application will access the model data. I created a file called `repository.model.ts` in the `src/app/model` folder and used it to define the class shown in listing 20.7.

Listing 20.7. The contents of the `repository.model.ts` file in the `src/app/model` folder

```

import { Injectable, Signal, signal } from "@angular/core";
import { Product } from "../product.model";
import { StaticDataSource } from "../static.datasource";

```



```

@Injectable()
export class Model {
  private products = signal<Product[]>([]);
  private locator = (p: Product, id?: number) => p.id == id;

  constructor(private dataSource: StaticDataSource) {
    this.products.set(this.dataSource.getData());
  }

  get Products(): Signal<Product[]> {
    return this.products.asReadonly();
  }

  getProduct(id: number): Product | undefined {
    return this.products().find(p => this.locator(p, id));
  }

  saveProduct(product: Product) {
    if (product.id == 0 || product.id == undefined) {
      product.id = this.generateID();
      this.products.mutate(prods => prods.push(product));
    } else {
      this.products.mutate(prods => {
        let index = prods.findIndex(p =>
          this.locator(p, product.id));
        prods.splice(index, 1, product);
      });
    }
  }

  deleteProduct(id: number) {
    this.products.mutate(prods => {
      let index = prods.findIndex(p => this.locator(p, id));
      if (index > -1) {
        prods.splice(index, 1);
      }
    });
  }

  private generateID(): number {
    let candidate = 100;
    while (this.getProduct(candidate) != null) {
      candidate++;
    }
    return candidate;
  }
}

```

20.2.3 Completing the model module

To complete the data model, I need to define the module. I created a file called `model.module.ts` in the `src/app/model` folder and used it to define the Angular module shown in listing 20.8.

Listing 20.8. The contents of the `model.module.ts` file in the `src/app/model` folder

```

import { NgModule } from "@angular/core";

```

```
import { StaticDataSource } from "../static.datasource";
import { Model } from "../repository.model";

@NgModule({
  providers: [Model, StaticDataSource]
})
export class ModelModule { }
```

20.3 Creating the messages module

The messages module will contain a service that is used to report messages or errors that should be displayed to the user and a component that presents them.

20.3.1 Creating the message model and service

To represent messages that should be displayed to the user, I added a file called `message.model.ts` to the `src/app/messages` folder and added the code shown in listing 20.9.

Listing 20.9. The contents of the `message.model.ts` file in the `src/app/messages` folder

```
export class Message {
  constructor(public text: string,
    public error: boolean = false) { }
}
```

The `Message` class defines properties that present the text that will be displayed to the user and whether the message represents an error. Next, I created a file called `message.service.ts` in the `src/app/messages` folder and used it to define the service shown in listing 20.10, which will be used to register messages that should be displayed to the user.

Listing 20.10. The contents of the `message.service.ts` file in the `src/app/messages` folder

```
import { Injectable, signal } from "@angular/core";
import { Message } from "../message.model";

@Injectable()
export class MessageService {
  private writableMessages = signal<Message[]>([]);

  messages = this.writableMessages.asReadonly();

  reportMessage(msg: Message) {
    this.writableMessages.mutate(msgs => msgs.push(msg));
  }
}
```

This Angular service uses a signal to keep track of messages, which are received by the `reportMessage` method and distributed as a read-only signal.

20.3.2 Creating the component and template

Now that I have a source of messages, I can create a component that will display them to the user. I added a file called `message.component.ts` to the `src/app/messages` folder and defined the component shown in listing 20.11.

Listing 20.11. The `message.component.ts` file in the `src/app/messages` folder

```
import { Component, Signal, computed } from "@angular/core";
import { MessageService } from "../message.service";
import { Message } from "../message.model";

@Component({
  selector: "paMessages",
  templateUrl: "message.component.html",
})
export class MessageComponent {
  lastMessage!: Signal<Message>;

  constructor(messageService: MessageService) {
    this.lastMessage = computed(() => {
      return messageService.messages()
        [messageService.messages().length - 1];
    })
  }
}
```

The component receives a `MessageService` object as its constructor argument and creates a computed signal that obtains the most recent message. To provide a template for the component, I created a file called `message.component.html` in the `src/app/messages` folder and added the markup shown in listing 20.12, which displays the message to the user.

Listing 20.12. The `message.component.html` file in the `src/app/messages` folder

```
<div *ngIf="lastMessage()"
  class="bg-primary text-white p-2 text-center"
  [class.bg-danger]="lastMessage().error">
  <h4>{{lastMessage().text}}</h4>
</div>
```

20.3.3 Completing the message Module

I added a file called `message.module.ts` in the `src/app/messages` folder and defined the module shown in listing 20.13.

Listing 20.13. The `message.module.ts` file in the `src/app/messages` folder

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { MessageComponent } from "../message.component";
import { MessageService } from "../message.service";

@NgModule({
  imports: [BrowserModule],
  declarations: [MessageComponent],
  exports: [MessageComponent],
})
```

```

        providers: [MessageService]
    })
    export class MessageModule { }

```

20.4 Creating the core module

The core module will contain the central functionality of the application, built on features that were described in part 2, that presents the user with a list of the products in the model and the ability to create and edit them.

20.4.1 Creating the shared state service

To help the components in this module to collaborate, I am going to add a service that records the current mode, noting whether the user is editing or creating a product. I added a file called `sharedState.service.ts` to the `src/app/core` folder and defined the enum and class shown in listing 20.14.

Listing 20.14. The contents of the `sharedState.service.ts` file in the `src/app/core` folder

```

import { Injectable, signal } from "@angular/core";

export enum MODES {
    CREATE, EDIT
}

export class State {
    constructor(public mode: MODES, public id?: number) {}
}

@Injectable()
export class SharedState {
    private stateVal = signal(new State(MODES.CREATE));

    get state() { return this.stateVal.asReadonly(); }

    update(mode: MODES, id?: number) {
        this.stateVal.set(new State(mode, id));
    }
}

```

The `SharedState` class uses a signal to track the current mode and the ID of the data model object that is being operated on. A read-only signal is provided so that the current state can be accessed in the rest of the application, and the state is modified using the `update` method.

20.4.2 Creating the table component

This component will present the user with the table that lists all the products in the application and that will be the main focal point in the application, providing access to other areas of functionality through buttons that allow objects to be created, edited, or deleted. Listing 20.15 shows the contents of the `table.component.ts` file, which I created in the `src/app/core` folder.

Listing 20.15. The contents of the `table.component.ts` file in the `src/app/core` folder

```

import { Component, Signal } from "@angular/core";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";
import { MODES, SharedState } from "../sharedState.service";

@Component({
  selector: "paTable",
  templateUrl: "table.component.html"
})
export class TableComponent {

  constructor(private model: Model, private state: SharedState) { }

  getProduct(key: number): Product | undefined {
    return this.model.getProduct(key);
  }

  get Products(): Signal<Product[]> {
    return this.model.Products;
  }

  deleteProduct(key?: number) {
    if (key != undefined) {
      this.model.deleteProduct(key);
    }
  }

  editProduct(key?: number) {
    this.state.update(MODES.EDIT, key)
  }

  createProduct() {
    this.state.update(MODES.CREATE);
  }
}

```

This component provides the same basic functionality used in part 2, with the addition of the `editProduct` and `createProduct` methods. These methods update the shared state service when the user wants to edit or create a product.

CREATING THE TABLE COMPONENT TEMPLATE

To provide the table component with a template, I added an HTML file called `table.component.html` to the `src/app/core` folder and added the markup shown in listing 20.16.

Listing 20.16. The contents of the `table.component.html` file in the `src/app/core` folder

```

<table class="table table-sm table-bordered table-striped">
  <thead>
    <tr>
      <th>ID</th><th>Name</th><th>Category</th>
      <th>Price</th><th></th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let item of Products()">

```

```

        <td>{{item.id}}</td>
        <td>{{item.name}}</td>
        <td>{{item.category}}</td>
        <td>{{item.price | currency:"USD" }}</td>
        <td class="text-center">
            <button class="btn btn-danger btn-sm m-1"
                (click)="deleteProduct(item.id)">
                Delete
            </button>
            <button class="btn btn-warning btn-sm"
                (click)="editProduct(item.id)">
                Edit
            </button>
        </td>
    </tr>
</tbody>
</table>
<button class="btn btn-primary mt-1" (click)="createProduct()">
    Create New Product
</button>

```

This template uses the `ngFor` directive to create rows in a table for each product in the data model, including buttons that call the `deleteProduct` and `editProduct` methods. There is also a `button` element outside of the table that calls the component's `createProduct` method when it is clicked.

20.4.3 Creating the form component

For this project, I am going to create a form component that will manage an HTML form that will allow new products to be created and allow existing products to be modified. To define the component, I added a file called `form.component.ts` to the `src/app/core` folder and added the code shown in listing 20.17.

Listing 20.17. The contents of the `form.component.ts` file in the `src/app/core` folder

```

import { Component } from "@angular/core";
import { NgForm } 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 } from "../sharedState.service";
import { toObservable } from "@angular/core/rxjs-interop";

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

  constructor(private model: Model, private stateService: SharedState,
    messageService: MessageService) {

```

```

    toObservable(stateService.state).subscribe(state => {
      this.editing = state.mode == MODES.EDIT;
      if (this.editing && state.id) {
        this.product = Product.fromProduct(
          this.model.getProduct(state.id) ?? new Product();
        );
      } else {
        this.product = new Product;
      }
      messageService.reportMessage(state.id
        ? new Message(`Editing ${this.product.name}`)
        : new Message("Creating New Product"));
    });
  }

  submitForm(form: NgForm) {
    if (form.valid) {
      this.model.saveProduct(this.product);
      this.product = new Product();
      this.stateService.update(MODES.CREATE);
      form.resetForm();
    }
  }
}

```

A single component will present a form used to both create new products and edit existing ones. The component depends on two services. The signals provided by the `SharedState` service provide updates that change the details of the form shown to the user, based on whether a product is being created or edited. I want to respond to changes in the state signal in a way that will allow me to use two-way data bindings, and so I have used the `toObservable` function to create an observable sequence that will produce a value every time the underlying signal changes.

The other service is `MessageService`, which is used to send messages that indicate the create/edit mode so they can be displayed to the user.

CREATING THE FORM COMPONENT TEMPLATE

To provide the component with a template, I added an HTML file called `form.component.html` to the `src/app/core` folder and added the markup shown in listing 20.18.

Listing 20.18. The contents of the `form.component.html` file in the `src/app/core` folder

```

<form #form="ngForm" (ngSubmit)="submitForm(form)"
      (reset)="form.resetForm()" >
  <div class="form-group">
    <label>Name</label>
    <input class="form-control" name="name"
      [(ngModel)]="product.name" required />
  </div>

  <div class="form-group">
    <label>Category</label>
    <input class="form-control" name="category"
      [(ngModel)]="product.category" required />
  </div>

```

```

<div class="form-group">
  <label>Price</label>
  <input class="form-control" name="price"
    [(ngModel)]="product.price"
    required pattern="^[0-9\\.]+$" />
</div>

<div class="mt-2">
  <button type="submit" class="btn btn-primary"
    [class.btn-warning]="editing" [disabled]="form.invalid">
    {{editing ? "Save" : "Create"}}
  </button>
  <button type="reset" class="btn btn-secondary m-1">Cancel</button>
</div>
</form>

```

The most important part of this template is the `form` element, which contains `input` elements for the `name`, `category`, and `price` properties required to create or edit a product. The header at the top of the template and the submit button for the form change their content and appearance based on the editing mode to distinguish between different operations.

CREATING THE FORM COMPONENT STYLES

To keep the example simple, I have used the basic form validation without any error messages. Instead, I rely on CSS styles that are applied using Angular validation classes. I added a file called `form.component.css` to the `src/app/core` folder and defined the styles shown in listing 20.19.

Listing 20.19. The contents of the `form.component.css` file in the `src/app/core` folder

```

input.ng-dirty.ng-invalid { border: 2px solid #ff0000 }
input.ng-dirty.ng-valid { border: 2px solid #6bc502 }

```

20.4.4 Completing the core module

To define the module that contains the components, I added a file called `core.module.ts` to the `src/app/core` folder and created the Angular module shown in listing 20.20.

Listing 20.20. The contents of the `core.module.ts` file in the `src/app/core` folder

```

import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule } from "@angular/forms";
import { ModelModule } from "../model/model.module";
import { TableComponent } from "../table.component";
import { FormComponent } from "../form.component";
import { SharedState } from "../sharedState.service";

@NgModule({
  imports: [BrowserModule, FormsModule, ModelModule],
  declarations: [TableComponent, FormComponent],
  exports: [ModelModule, TableComponent, FormComponent],
  providers: [SharedState]
})
export class CoreModule { }

```


This module imports the core Angular functionality, the Angular form features, and the application's data model, created earlier in the chapter. It also sets up a provider for the `SharedState` service.

20.5 Completing the project

To bring all the different modules together, I made the changes shown in listing 20.21 to the root module.

Listing 20.21. Configuring the application 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 { ModelModule } from "../model/model.module";
import { CoreModule } from "../core/core.module";
import { TableComponent } from "../core/table.component";
import { FormComponent } from "../core/form.component";
import { MessageModule } from "../messages/message.module";
import { MessageComponent } from "../messages/message.component";

@NgModule({
  declarations: [],
  imports: [BrowserModule, ModelModule, CoreModule, MessageModule],
  providers: [],
  bootstrap: [TableComponent, FormComponent, MessageComponent]
})
export class AppModule { }
```

The module imports the feature modules created in this chapter and specifies three bootstrap components, two of which were defined in `CoreModule` and one from `MessageModule`. These will display the product table and form and any messages or errors.

The final step is to update the HTML file so that it contains elements that will be matched by the `selector` properties of the bootstrap components, as shown in listing 20.22.

Listing 20.22. Adding custom elements in the `index.html` file in the `src` folder

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>ExampleApp</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>
  <paMessages></paMessages>
  <div class="row m-2">
    <div class="col-8 p-2">
      <paTable></paTable>
    </div>
    <div class="col-4 p-2">
```

```
        <paForm></paForm>
      </div>
    </div>
  </body>
</html>
```

Run the following command in the `exampleApp` folder to start the Angular development tools and build the project:

```
ng serve
```

Once the initial build process has completed, open a new browser window and navigate to `http://localhost:4200` to see the content shown in figure 20.1.

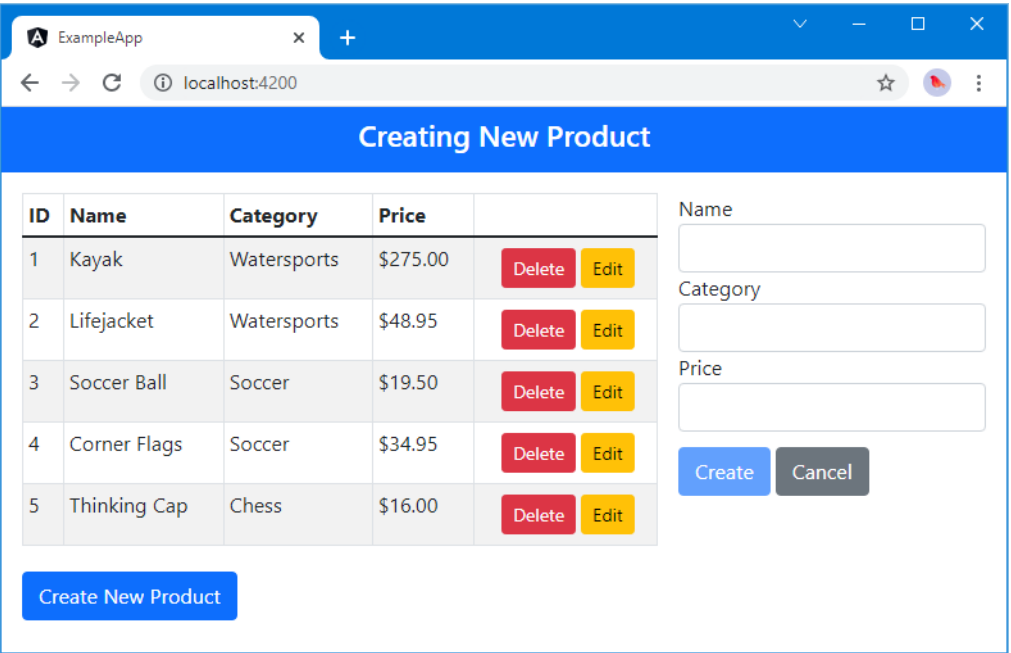


Figure 20.1. Running the example application

Fill out the form and click the Create button to add a product to the repository. You can also click the Edit button to select a product for editing and click the Delete button to remove a button.

20.6 Summary

In this chapter, I created the example project that I will use in this part of the book. The basic structure is the same as the example used in earlier chapters but without the redundant code and markup that I used to demonstrate earlier features. In the next chapter, I describe the advanced features Angular provides for working with forms.

21

Using the forms API, part 1

This chapter covers

- Using the Angular API to create and manage form elements
- Validating form elements created with the API
- Grouping form elements programmatically

In this chapter, I describe the Angular forms API, which provides an alternative to the template-based approach to forms introduced in chapter 13. The forms API is a more complicated way of creating forms, but it allows fine-grained control over how forms behave, how they respond to user interaction, and how they are validated. Table 21.1 puts the forms API in context.

Table 21.1. Putting the forms API in context

Question	Answer
What is it?	The forms API allows for the creation of reactive forms, which are managed using the code in a component class.
Why is it useful?	The forms API provides a component with more control over the elements in forms and allows their behavior to be customized.
How is it used?	<code>FormControl</code> and <code>FormGroup</code> objects are created by the component class and associated with elements in the template using directives.
Are there any pitfalls or limitations?	The forms API is complex, and additional work is required to ensure that features such as validation behave consistently.

Are there any alternatives?	The forms API is optional. Forms can be defined using the basic features described in chapter 13.
-----------------------------	---

Table 21.2 summarizes the chapter.

Table 21.2. Chapter summary

Problem	Solution	Listing
Creating a reactive form	Create a <code>FormControl</code> object in the component class and associate it with a form element in the template using the <code>formControl</code> directive	1–3
Responding to element value changes	Use the observable <code>valueChanges</code> property defined by the <code>FormControl</code> class	4, 5
Managing element state	Use the properties defined by the <code>FormControl</code> class	6
Responding to element validation changes	Use the observable <code>statusChanges</code> property defined by the <code>FormControl</code> class	7–13
Defining multiple related form elements	Use a <code>FormGroup</code> object	14–18, 22–26
Displaying validation messages for controls in a group	Obtain a <code>FormControl</code> object through the enclosing <code>FormGroup</code> object	19–20, 27, 28

21.1 Preparing for this chapter

For this chapter, I will continue using the `exampleApp` project that I created in chapter 20. No changes are required for this chapter.

TIP You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/manningbooks/pro-angular-16>. See chapter 1 for how to get help if you have problems running the examples.

To start the development server, open a command prompt, navigate to the `exampleApp` folder, and run the following command:

```
ng serve
```

Open a new browser window and navigate to `http://localhost:4200` to see the content shown in figure 21.1.

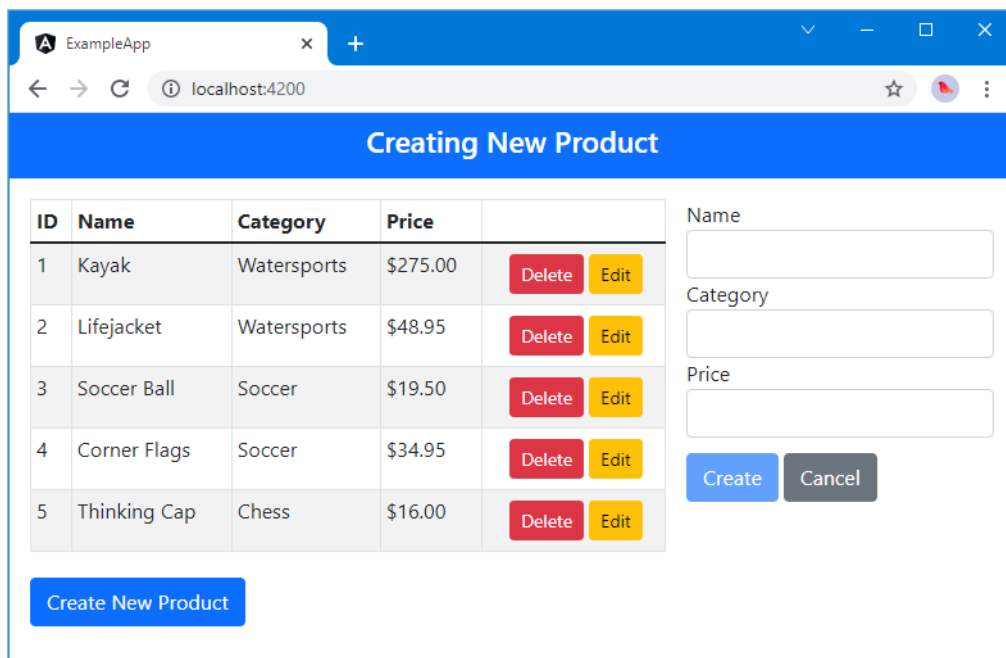


Figure 21.1. Running the example application

21.2 Understanding the reactive forms API

The simplest way to use HTML forms is to use Angular two-way bindings to connect `input` elements to component properties, which is the approach I demonstrated in part 2 and which I used to create the form in the example project in chapter 20. These are known as *template-driven forms*.

For more complex forms, Angular provides a complete API that exposes the state of HTML forms and allows their data and structure to be managed, known as *reactive forms*. You can get a glimpse of the API in the way that the form element is defined in the `form.component.html` file in the `src/app` folder:

```
...
<form #form="ngForm" (ngSubmit)="submitForm(form)"
      (reset)="form.resetForm()">
...

```

The template variable named `form` is assigned the value `ngForm`, which is then used in the event bindings, as a method argument in the `ngSubmit` event or to invoke a method in the `reset` event. The `ngForm` value and the events themselves are defined as part of the Angular form API.

One of the themes of this book has been that nothing in Angular is magic. Every feature is implanted using the capabilities of the browser or builds on other Angular features. This includes `ngForm`, which is a directive that acts as a wrapper around a `FormGroup` object,

exposing its capabilities using the directive features described in chapter 13. The `FormGroup` class, which is defined in the `@angular/forms` package, provides an API for working with a form and can be used directly in a component class, allowing forms to be manipulated in code and not just through HTML elements in a template. In turn, the `FormGroup` is a container for `FormControl` objects, each of which represents an element in the form. As you will learn, the `FormGroup` and `FormControl` classes are the building blocks of the reactive forms API, and the `ngForm` directive, with which you are already familiar, simply presents this API so it can be used easily in templates.

21.3 Rebuilding the form using the API

The simplest way to get started is with a single form element so you can understand the basic building blocks of the API. The reactive form features require a new module, `ReactiveFormsModule`, as shown in listing 21.1.

Listing 21.1. Importing a module in the `core.module.ts` file in the `src/app/core` folder

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule, ReactiveFormsModule } from "@angular/forms";
import { ModelModule } from "../model/model.module";
import { TableComponent } from "../table.component";
import { FormComponent } from "../form.component";
import { SharedState } from "../sharedState.service";

@NgModule({
  imports: [BrowserModule, FormsModule, ModelModule,
    ReactiveFormsModule],
  declarations: [TableComponent, FormComponent],
  exports: [ModelModule, TableComponent, FormComponent],
  providers: [SharedState]
})
export class CoreModule { }
```

Listing 21.2 simplifies the form template so that it contains only a single input element, along with a label and a div element.

Listing 21.2. Simplifying the HTML template in the `form.component.html` file in the `src/app/core` folder

```
<div class="form-group">
  <label>Name</label>
  <input class="form-control" name="name" [formControl]="nameField" />
</div>
```

In addition to simplifying the template, listing 21.1 makes an important change in the way that the input element is configured. The `ngModel` directive isn't used with the forms API, and a different directive is applied:

```
...
<input class="form-control" name="name" [formControl]="nameField" />
...
```

The `formControl` directive creates the relationship between the HTML element in the template and a `FormControl` property in the component class, through which the element

will be managed. Listing 21.3 simplifies the component, adds a property for the `formControl` component to use, and takes advantage of one of the features provided by the `FormControl` class.

Listing 21.3. Using the forms API in the `form.components.ts` file in the `src/app/core` folder

```
import { Component } from "@angular/core";
import { FormControl, NgForm } 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 } from "../sharedState.service";
import { toObservable } from "@angular/core/rxjs-interop";

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

  constructor(private model: Model, private stateService: SharedState,
    messageService: MessageService) {

    toObservable(stateService.state).subscribe(state => {
      this.editing = state.mode == MODES.EDIT;
      if (this.editing && state.id) {
        this.product = Product.fromProduct(
          this.model.getProduct(state.id) ?? new Product() );
      } else {
        this.product = new Product;
      }
      this.nameField.setValue(this.product.name ?? "");
      messageService.reportMessage(state.id
        ? new Message(`Editing ${this.product.name}`)
        : new Message("Creating New Product"));
    });
  }

  // submitForm(form: NgForm) {
  //   if (form.valid) {
  //     this.model.saveProduct(this.product);
  //     this.product = new Product();
  //     this.stateService.update(MODES.CREATE);
  //     form.resetForm();
  //   }
  // }
```

Individual form controls are represented by `FormControl` objects. Listing 21.3 defines a `FormControl` property whose name matches the one specified by the `formControl` directive in listing 21.2, creating the relationship between the HTML element and the component class:

```
...
nameField = new FormControl({value: "", disabled: false});
...
```

The constructor accepts an object with `value` and `disabled` properties that specify the initial value of the control and whether the control is disabled.

The reactive forms API provides direct access to the features that were previously managed through the `ngForm` and `ngModel` directives. In this case, I have used the `setValue` method to set the contents of the `input` element when there is a change in the application state, such as when the user clicks an Edit button:

```
...
this.nameField.setValue(this.product.name ?? "");
...
```

As its name suggests, the `setValue` method sets the value of the form control, which was previously done by the `ngModel` directive. The `setValue` method is one of the basic features provided by the `FormControl` class, the most useful of which are described in table 21.3 and which I describe in the following sections.

NOTE Some of the methods described in table 21.3 and later tables take an optional argument that manages the effect of changes. I don't describe these options because they are not typically required. See the Angular API description for the `FormControl` class (<https://angular.io/api/forms/FormControl>) for details.

Table 21.3. Useful basic `FormControl` members

Name	Description
<code>value</code>	This property returns the current value of the form control, defined using the <code>any</code> type.
<code>setValue(value)</code>	This method sets the value of the form control.
<code>valueChanges</code>	This property returns an <code>Observable<any></code> , through which changes can be observed.
<code>enabled</code>	This property returns <code>true</code> if the form control is enabled.
<code>disabled</code>	This property returns <code>true</code> if the form control is disabled.
<code>enable()</code>	This method enables the form control.
<code>disable()</code>	This method disables the form control.
<code>reset(value)</code>	This method resets the form control, with an optional value. The form control will be reset to its default state if the value argument is omitted.

The overall effect is to transfer control of the `input` element from the template to the component, through the `FormControl` property. When the form component is displayed, the `input` element is populated with an initial value, which is replaced when the user clicks one of the Edit buttons, as shown in figure 21.2.

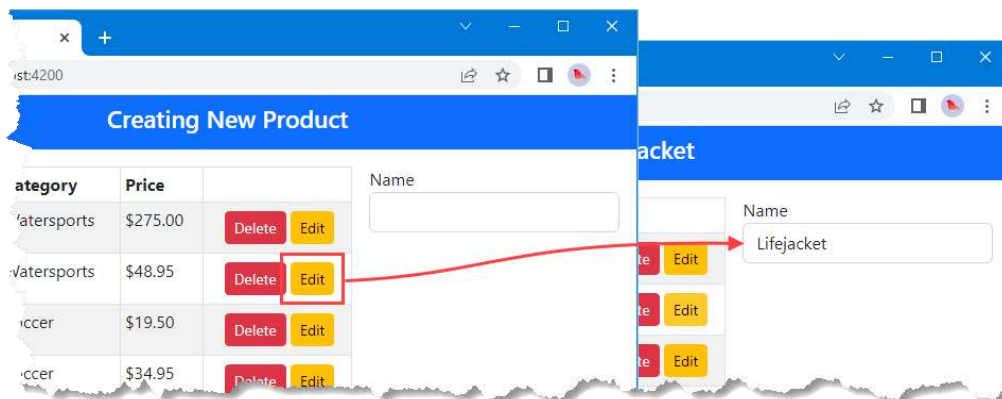


Figure 21.2. Using a `FormControl`

21.3.1 Responding to form control changes

The `valueChanges` property returns an observable that emits new values from the form control. Components can observe these changes to respond to user interaction, as shown in listing 21.4.

Listing 21.4. Observing changes in the `form.component.ts` file in the `src/app/core` folder

```
import { Component } from "@angular/core";
import { FormControl, NgForm } 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 } from "../sharedState.service";
import { toObservable } from "@angular/core/rxjs-interop";

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

  constructor(private model: Model, private stateService: SharedState,
    messageService: MessageService) {
```

```

toObservable(stateService.state).subscribe(state => {
  this.editing = state.mode == MODES.EDIT;
  if (this.editing && state.id) {
    this.product = Product.fromProduct(
      this.model.getProduct(state.id) ?? new Product();
    )
  } else {
    this.product = new Product;
  }
  this.nameField.setValue(this.product.name ?? "");
  messageService.reportMessage(state.id
    ? new Message(`Editing ${this.product.name}`)
    : new Message("Creating New Product"));
});

this.nameField.valueChanges.subscribe(newValue =>
  messageService.reportMessage(
    new Message(newValue || "(Empty)"));
}
}

```

In the constructor, the component subscribes to the `Observable<any>` returned by the `valueChanges` property and passes on the values it receives to the message service. As the user types into the input element, the changed value is displayed at the top of the layout, as shown in figure 21.3.

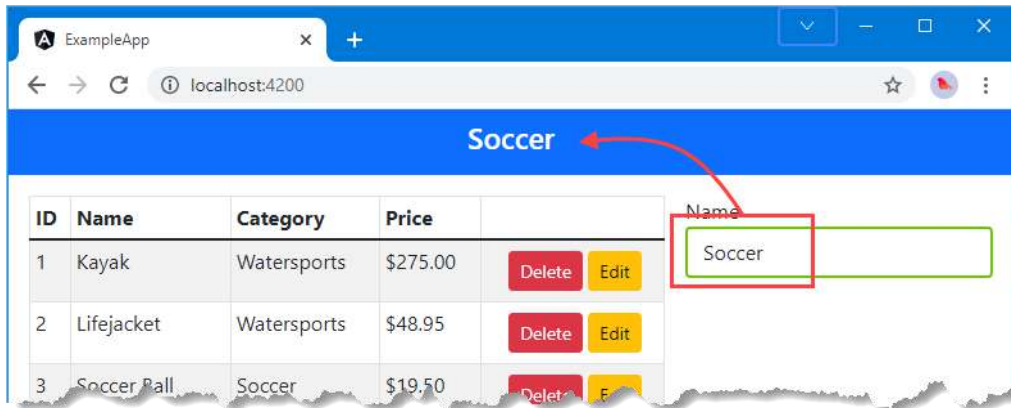


Figure 21.3. Responding to form control changes

By default, the observable will emit a new event in response to the HTML element's `change` event, but this can be altered by configuring the `FormControl` with a constructor argument, as shown in listing 21.5.

Listing 21.5. Configuring a `FormControl` in the `form.component.ts` file in the `src/app/core` folder

```

import { Component } from "@angular/core";

```

```

import { FormControl, NgForm } 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 } from "../sharedState.service";
import { toObservable } from "@angular/core/rxjs-interop";

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

  constructor(private model: Model, private stateService: SharedState,
    messageService: MessageService) {

    // ...statements omitted for brevity...
  }
}

```

The new argument to the `FormControl` constructor implements the `AbstractControlOptions` interface, which defines the properties described in table 21.4. Those properties are all optional, which means you can omit those you do not need to change.

Table 21.4. The properties defined by the `AbstractControlOptions` interface

Name	Description
<code>validators</code>	This property is used to configure the validation for the form control, as described in the “Managing Control Validation” section.
<code>asyncValidators</code>	This property is used to configure the async validation for the form control, as described in chapter 22.
<code>updateOn</code>	This property is used to configure when the <code>valueChanges</code> observable will emit a new value. It can be set to <code>change</code> , the default; <code>blur</code> ; or <code>submit</code> . The <code>submit</code> value is used with form elements.

In listing 21.5, I used the `blur` value for the `updateOn` property, which means that the observable will emit new values only when the `input` element loses focus, such as when the user tabs to another element, as shown in figure 21.4.

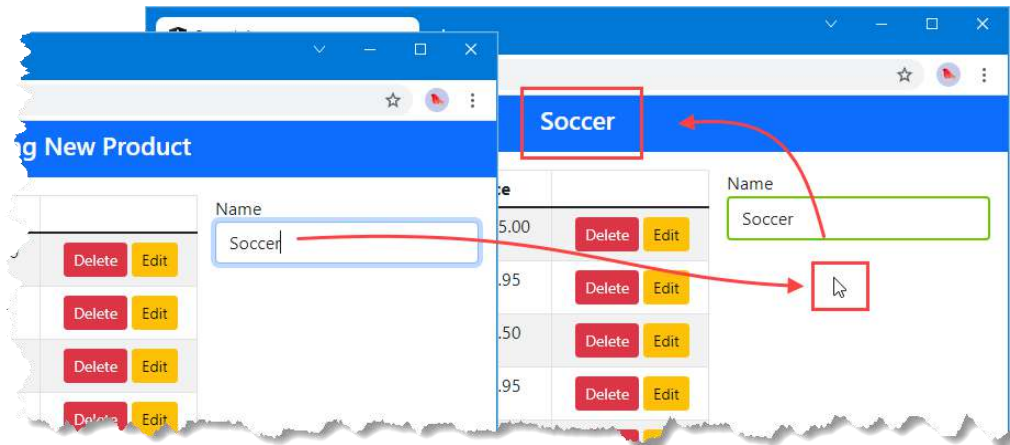


Figure 21.4. Changing the update setting

21.3.2 Managing control state

In chapter 13, I showed you how form elements are added to classes to denote their state. The `FormControl` class defines properties that indicate the state of the HTML element and methods for manually changing the state, as described in table 21.5.

Table 21.5. The `FormControl` members for element state

Name	Description
<code>untouched</code>	This property returns <code>true</code> if the HTML element is untouched, meaning that the element has not been selected.
<code>touched</code>	This property returns <code>true</code> if the HTML element has been touched, meaning that the element has been selected.
<code>markAsTouched()</code>	Calling method marks the element as touched.
<code>markAsUntouched()</code>	Calling this method marks the element as untouched.
<code>pristine</code>	This property returns <code>true</code> if the element contents have not been edited by the user.
<code>dirty</code>	This property returns <code>true</code> if the element contents have been edited by the user.
<code>markAsPristine()</code>	Calling this method marks the element as pristine.
<code>markAsDirty()</code>	Calling this method marks the element as dirty.

One benefit of using the reactive forms API is that you can control the way that the form features are applied, tailoring the behavior to the needs of your project. As a simple

demonstration, listing 21.6 changes the state of the element based on the number of characters in the value.

Listing 21.6. Changing state in the form.component.ts file in the src/app/core folder

```
import { Component } from "@angular/core";
import { FormControl, NgForm } 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 } from "../sharedState.service";
import { toObservable } from "@angular/core/rxjs-interop";

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

  constructor(private model: Model, private stateService: SharedState,
    messageService: MessageService) {

    toObservable(stateService.state).subscribe(state => {
      this.editing = state.mode == MODES.EDIT;
      if (this.editing && state.id) {
        this.product = Product.fromProduct(
          this.model.getProduct(state.id) ?? new Product() );
      } else {
        this.product = new Product;
      }
      this.nameField.setValue(this.product.name ?? "");
      messageService.reportMessage(state.id
        ? new Message(`Editing ${this.product.name}`)
        : new Message("Creating New Product"));
    });

    this.nameField.valueChanges.subscribe(newValue => {
      messageService.reportMessage(
        new Message(newValue || "(Empty)")
      );
      if (typeof(newValue) == "string"
        && newValue.length % 2 == 0) {
        this.nameField.markAsPristine();
      }
    });
  }
}
```

I have changed the `updateOn` property so that a new value is emitted via the observable after every change, and I added an `if` expression to the subscriber function that calls the `FormControl.markAsPristine` method if the length of the character is an even number.

The effect is that the border of the input element toggles on and off as the user types, as shown in figure 21.5.

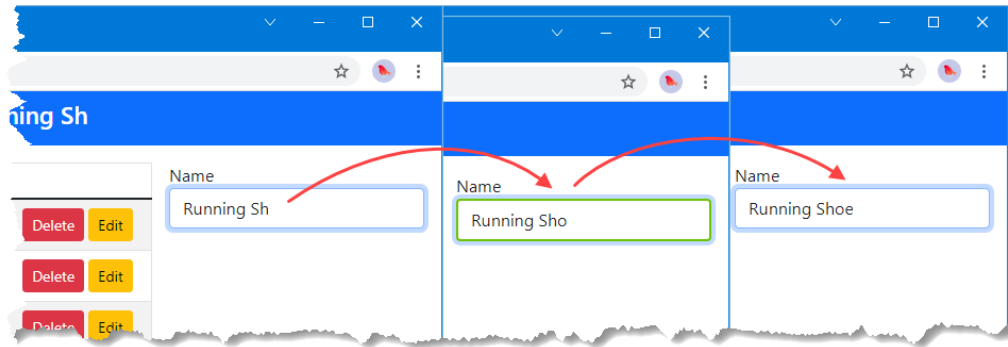


Figure 21.5. Changing element state

The reason the border color changes is that Angular marks form elements as valid, even if no validation requirements have been applied. This means that for an odd number of characters, the input element is added to the `ng-valid`, `ng-touched`, and `ng-dirty` classes, like this:

```
...
<input name="name" class="form-control ng-valid ng-touched ng-dirty">
...
```

This combination is matched by the selector for one of the styles defined in the `form.component.css` file, which I added to the project in chapter 20:

```
...
input.ng-dirty.ng-valid { border: 2px solid #6bc502 }
...
```

When there is an even number of characters, the call to the `markAsPristine` method creates a different combination of element classes:

```
...
<input name="name" class="form-control ng-valid ng-touched ng-pristine">
...
```

This doesn't match the CSS selector, and no border is displayed. This example demonstrates that you can customize the behavior of form elements by using the reactive forms API, even if this particular behavior is unlikely to be required in many projects.

21.3.3 Managing control validation

Form elements can be subject to validation, even when using the reactive forms API. Validation constraints can be added to the template, as described in chapter 13, or applied through the `FormControl` constructor, using the `validators` and `asyncValidators` properties of the `AbstractControlOptions` interface.

Listing 21.7 uses this feature to apply validation to the example form element, using the `validators` property. (I explain the use of the `asyncValidators` property in chapter 22.)

Listing 21.7. Applying validation in the form.component.ts file in the src/app/core folder

```

import { Component } from "@angular/core";
import { FormControl, NgForm, 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 } from "../sharedState.service";
import { toObservable } from "@angular/core/rxjs-interop";

@Component({
  selector: "paForm",
  templateUrl: "form.component.html",
  styleUrls: ["form.component.css"]
})
export class FormComponent {
  product: Product = new Product();
  editing: boolean = false;
  nameField = new FormControl({value: "", disabled: false},
    { updateOn: "change",
      validators: [
        Validators.required,
        Validators.minLength(3),
        Validators.pattern("^[A-Za-z ]+$")
      ]
    });

  constructor(private model: Model, private stateService: SharedState,
    messageService: MessageService) {

    // ...statements omitted for brevity...
  }
}

```

The built-in validators are defined as static properties of the `Validators` class, where each property name corresponds to the validation attributes described in chapter 13. Validation rules are specified using the `validators` property of the `AbstractControlOptions` constructor argument, which is assigned an array of validators. In listing 21.7, I used the `required`, `minLength`, and `pattern` properties to set the validation policy for the input element.

In addition to the constructor, the validators applied to a form control can be managed through the `FormControl` properties and methods described in table 21.6. For each entry in the table, there is a corresponding member for managing asynchronous validators, which I describe in chapter 22.

Table 21.6. The `FormControl` members for managing validators

Name	Description
<code>validator</code>	This property returns a function that combines all of the configured validators so that the form control can be validated with a single function call.

<code>hasValidator(v)</code>	This function returns <code>true</code> if the form control has been configured with the specified validator.
<code>setValidators(vals)</code>	This method sets the form control validators. The argument can be a single validator or an array of validators.
<code>addValidators(v)</code>	This method adds one or more validators to the form control.
<code>removeValidators(v)</code>	This method removes one or more validators from the control.
<code>clearValidators()</code>	This method removes all of the validators from the form control.

The validation state of a `FormControl` is determined and managed through the members described in table 21.7.

Table 21.7. The `FormControl` members for validation state

Name	Description
<code>status</code>	This property returns the validation state of the form control, expressed using a <code>FormControlStatus</code> value, which will be <code>VALID</code> , <code>INVALID</code> , <code>PENDING</code> , or <code>DISABLED</code> .
<code>statusChanges</code>	This property returns an <code>Observable<FormControlStatus></code> , which will emit a <code>FormControlStatus</code> value when the state of the form control changes.
<code>valid</code>	This property returns <code>true</code> if the form control's value passes validation.
<code>invalid</code>	This property returns <code>true</code> if the form control's value fails validation.
<code>pending</code>	This property returns <code>true</code> if the form control's value is being validated asynchronously, as described in chapter 22.
<code>errors</code>	This property returns a <code>ValidationErrors</code> object that contains the errors generated by the form control's validators, or <code>null</code> if there are no errors.
<code>getError(v)</code>	This method returns the error message, if there is one, for the specified validator. This method accepts an optional path for use with nested form controls, as described in the "Working with Multiple Form Controls" section.
<code>hasError(v)</code>	This method returns <code>true</code> if the specified validator has generated an error message. This method accepts an optional path for use with nested form controls, as described in the "Working with Multiple Form Controls" section.
<code>setErrors(errs)</code>	This method is used to add errors to the form control's validation status, which is useful when performing manual validation in the component. This method accepts an optional path for use with nested form controls, as described in the "Working with Multiple Form Controls" section.

The effect of the validators configured in listing 21.7 can be determined through the features described in table 21.7. Listing 21.8 uses a subscription to the `statusChanges` observable to generate messages summarizing the validation state of the form control.

Listing 21.8. Generating validation messages in the `form.component.ts` file in the `src/app/core` folder

```
import { Component } from "@angular/core";
import { FormControl, NgForm, 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 } from "../sharedState.service";
import { toObservable } from "@angular/core/rxjs-interop";

@Component({
  selector: "paForm",
  templateUrl: "form.component.html",
  styleUrls: ["form.component.css"]
})
export class FormComponent {
  product: Product = new Product();
  editing: boolean = false;
  nameField = new FormControl({value: "", disabled: false},
    { updateOn: "change",
      validators: [
        Validators.required,
        Validators.minLength(3),
        Validators.pattern("^[A-Za-z ]+$")
      ]
    });

  constructor(private model: Model, private stateService: SharedState,
    messageService: MessageService) {

    toObservable(stateService.state).subscribe(state => {
      this.editing = state.mode == MODES.EDIT;
      if (this.editing && state.id) {
        this.product = Product.fromProduct(
          this.model.getProduct(state.id) ?? new Product() );
      } else {
        this.product = new Product;
      }
      this.nameField.setValue(this.product.name ?? "");
      messageService.reportMessage(state.id
        ? new Message(`Editing ${this.product.name}`)
        : new Message("Creating New Product"));
    });

    // this.nameField.valueChanges.subscribe(newValue => {
    //   messageService.reportMessage(
    //     new Message(newValue || "(Empty)")
    //   )
    //   if (typeof(newValue) == "string"
    //     && newValue.length % 2 == 0) {
    //     this.nameField.markAsPristine();
    //   }
    // });
  }
}
```

```

// });

this.nameField.statusChanges.subscribe(newStatus => {
  if (newStatus == "INVALID" && this.nameField.errors != null) {
    let errs = Object.keys(this.nameField.errors).join(", ");
    messageService.reportMessage(
      new Message(`INVALID: ${errs}`))
  } else {
    messageService.reportMessage(new Message(newStatus));
  }
});
}
}

```

In response to status changes, a message is sent that details the status and, if it is `INVALID`, includes a list of the validators that have reported an error, as shown in figure 21.6. (See chapter 13 for details of how to process a `ValidationErrors` object to display validation messages that are more usefully presented to the user.)

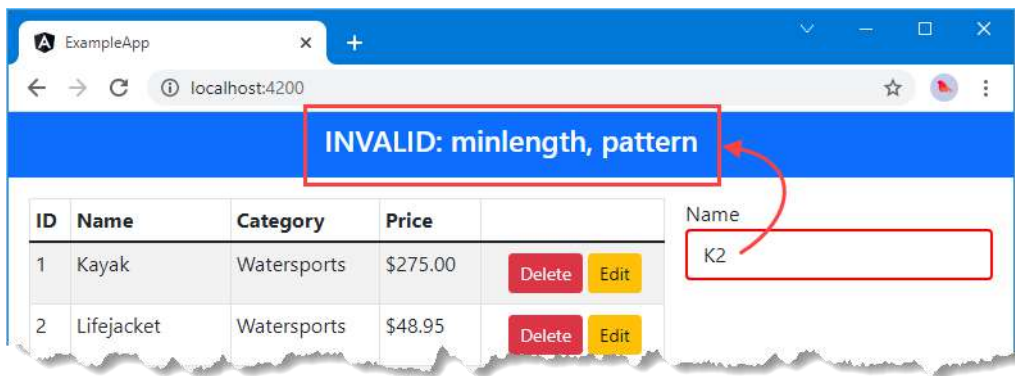


Figure 21.6. Responding to status changes

The message shown in figure 21.6 isn't especially useful to the user, but the `formControl` directive uses the `exportAs` property to provide an identifier named `ngForm` for use in template variables, and this can be used to generate more helpful validation messages for a control. To prepare, add a file named `validationHelper.pipe.ts` to the `src/app/core` folder with the content shown in listing 21.9, which creates a pipe to format validation messages.

Listing 21.9. The contents of the `validationHelper.pipe.ts` file in the `src/app/core` folder

```

import { Pipe } from "@angular/core";
import { FormControl, ValidationErrors } from "@angular/forms";

@Pipe({
  name: "validationFormat"
})
export class ValidationHelper {

```

```

transform(source: any, name: any) : string[] {
  if (source instanceof FormControl) {
    return this.format((source as FormControl).errors, name)
  }
  return this.format(source as ValidationErrors, name)
}

format(errors: ValidationErrors | null, name: string): string[] {
  let messages: string[] = [];
  for (let errorName in errors) {
    switch (errorName) {
      case "required":
        messages.push(`You must enter a ${name}`);
        break;
      case "minlength":
        messages.push(`A ${name} must be at least
          ${errors['minlength'].requiredLength}
          characters`);
        break;
      case "pattern":
        messages.push(`The ${name} contains
          illegal characters`);
        break;
    }
  }
  return messages;
}
}

```

This code is based on the approach I took in chapter 13 to generate user-friendly messages for template-driven forms. Listing 21.10 registers the pipe so that it will be available in the rest of the module.

Listing 21.10. Registering a pipe in the core.modules.ts file in the src/app/core folder

```

import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule, ReactiveFormsModule } from "@angular/forms";
import { ModelModule } from "../model/model.module";
import { TableComponent } from "../table.component";
import { FormComponent } from "../form.component";
import { SharedState } from "../sharedState.service";
import { ValidationHelper } from "../validationHelper.pipe";

@NgModule({
  imports: [BrowserModule, FormsModule, ModelModule,
    ReactiveFormsModule],
  declarations: [TableComponent, FormComponent, ValidationHelper],
  exports: [ModelModule, TableComponent, FormComponent],
  providers: [SharedState]
})
export class CoreModule { }

```

Listing 21.11 uses a template variable to obtain a reference for the `FormControl` object, which is used to get validation messages that can be displayed to the user.

Listing 21.11. Generating error messages in the form.component.html file in the src/app/core folder

```
<div class="form-group">
  <label>Name</label>
  <input class="form-control" name="name" [formControl]="nameField"
    #name="ngForm" />
  <ul class="text-danger list-unstyled mt-1"
    *ngIf="name.dirty && name.invalid">
    <li *ngFor="let err of name.errors | validationFormat:'name'">
      {{ err }}
    </li>
  </ul>
</div>
```

The `formControl` directive defines properties that correspond to those described in table 21.5 and table 21.7, which provides access to the validation state and errors, which are formatted using the new pipe. Validation messages are displayed to the user, as shown in figure 21.7, achieving the same result as with the template-based form from earlier chapters.

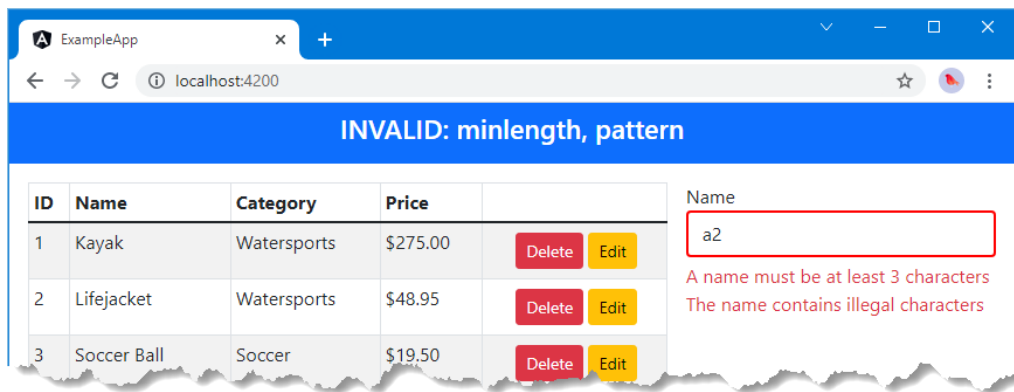


Figure 21.7. Displaying validation messages

21.3.4 Adding additional controls

The advantage of using the forms API is that you can customize the way that your forms work, and this extends to using the state of one control to determine the state of another. To prepare, listing 21.12 introduces another input element to the template.

Listing 21.12. Adding an element in the form.component.html file in the src/app/core folder

```
<div class="form-group">
  <label>Name</label>
  <input class="form-control" name="name" [formControl]="nameField"
    #name="ngForm" />
  <ul class="text-danger list-unstyled mt-1"
    *ngIf="name.dirty && name.invalid">
```

```

        <li *ngFor="let err of name.errors | validationFormat:'name'">
            {{ err }}
        </li>
    </ul>
</div>

<div class="form-group">
    <label>Category</label>
    <input class="form-control" name="category"
        [formControl]="categoryField" />
</div>

```

The new input element is configured with a `formControl` binding that specifies a property named `categoryField`. Listing 21.13 defines this property and uses the features provided by the `FormControl` class to change the element's state based on the `name` field.

Listing 21.13. Adding a FormControl in the form.component.ts file in the src/app/core folder

```

import { Component } from "@angular/core";
import { FormControl, NgForm, 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 } from "../sharedState.service";
import { toObservable } from "@angular/core/rxjs-interop";

@Component({
    selector: "paForm",
    templateUrl: "form.component.html",
    styleUrls: ["form.component.css"]
})
export class FormComponent {
    product: Product = new Product();
    editing: boolean = false;
    nameField = new FormControl({value: "", disabled: false},
        { updateOn: "change",
            validators: [
                Validators.required,
                Validators.minLength(3),
                Validators.pattern("^[A-Za-z ]+$")
            ]
        });

    categoryField: FormControl = new FormControl();

    constructor(private model: Model, private stateService: SharedState,
        messageService: MessageService) {

        toObservable(stateService.state).subscribe(state => {
            this.editing = state.mode == MODES.EDIT;
            if (this.editing && state.id) {
                this.product = Product.fromProduct(
                    this.model.getProduct(state.id) ?? new Product() );
            } else {
                this.product = new Product;
            }
        });
    }
}

```

```

        this.nameField.setValue(this.product.name ?? "");
        this.categoryField.setValue(this.product.category);
        messageService.reportMessage(state.id
            ? new Message(`Editing ${this.product.name}`)
            : new Message("Creating New Product"));
    });

    this.nameField.statusChanges.subscribe(newStatus => {
        if (newStatus == "INVALID") {
            this.categoryField.disable();
        } else {
            this.categoryField.enable();
        }
    });
}
}

```

The `category` input element is created without constructor arguments, which means that there will be no initial value and no changes to the default configuration. The new element is enabled and disabled based on the validation status of the `name` element. To see the effect, start typing characters into the Name field, which will be disabled by the error produced by the `minlength` validator. Once the minimum length is reached, the Name field will pass validation, and the Category field will be enabled, as shown in figure 21.8.

The screenshot shows a web browser window with the title 'ExampleApp' and the address 'localhost:4200'. The page has a blue header with the text 'Creating New Product'. Below the header is a table with 5 rows of product data. To the right of the table is a form for creating a new product. The form has two input fields: 'Name' and 'Category'. The 'Name' field is highlighted with a red border and has a red error message below it: 'A name must be at least 3 characters'. The 'Category' field is disabled and has a light gray background. Below the form is a blue button labeled 'Create New Product'.

ID	Name	Category	Price	
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	Thinking Cap	Chess	\$16.00	Delete Edit

Name:

A name must be at least 3 characters

Category:

Create New Product

Figure 21.8. Working with multiple controls

21.4 Working with multiple form controls

Manipulating individual `FormControl` objects can be a powerful technique, but it can also be cumbersome in more complex forms, where there can be many objects to create and manage. The reactive forms API includes the `FormGroup` class, which represents a group of form controls, which can be manipulated individually or as a combined group. Listing 21.14 introduces a `FormGroup` property to the example component.

Listing 21.14. Using a `FormGroup` in the `form.component.ts` file in the `src/app/core` folder

```
import { Component } from "@angular/core";
import { FormControl, FormGroup, NgForm, 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 } from "../sharedState.service";
import { toObservable } from "@angular/core/rxjs-interop";

@Component({
  selector: "paForm",
  templateUrl: "form.component.html",
  styleUrls: ["form.component.css"]
})
export class FormComponent {
  product: Product = new Product();
  editing: boolean = false;
  nameField = new FormControl({value: "", disabled: false},
    { updateOn: "change",
      validators: [
        Validators.required,
        Validators.minLength(3),
        Validators.pattern("^[A-Za-z ]+$")
      ]});

  categoryField: FormControl = new FormControl();

  productForm: FormGroup = new FormGroup({
    name: this.nameField, category: this.categoryField
  });

  constructor(private model: Model, private stateService: SharedState,
    messageService: MessageService) {

    toObservable(stateService.state).subscribe(state => {
      this.editing = state.mode == MODES.EDIT;
      if (this.editing && state.id) {
        this.product = Product.fromProduct(
          this.model.getProduct(state.id) ?? new Product());
      } else {
        this.product = new Product;
      }
      // this.nameField.setValue(this.product.name ?? "");
    });
  }
}
```

```

        // this.categoryField.setValue(this.product.category);
        this.productForm.reset(this.product);
        messageService.reportMessage(state.id
            ? new Message(`Editing ${this.product.name}`)
            : new Message("Creating New Product"));
    });

    // this.nameField.statusChanges.subscribe(newStatus => {
    //     if (newStatus == "INVALID") {
    //         this.categoryField.disable();
    //     } else {
    //         this.categoryField.enable();
    //     }
    // });

    this.productForm.statusChanges.subscribe(newStatus => {
        if (newStatus == "INVALID") {
            let invalidControls: string[] = [];
            for (let controlName in this.productForm.controls) {
                if (this.productForm.controls[controlName].invalid) {
                    invalidControls.push(controlName)
                }
            }
            messageService.reportMessage(
                new Message(`INVALID: ${invalidControls.join(", ")}`)
            )
        } else {
            messageService.reportMessage(new Message(newStatus));
        }
    })
}

```

This is the simplest use of a `FormGroup`, where a property is used to group existing `FormControl` objects so they can be processed collectively. The individual `FormControl` objects are passed to the `FormGroup` constructor, in a map that assigns each a key. Controls can also be added, removed, and inspected in the `FormGroup` using the members defined in table 21.8.

Table 21.8. `FormGroup` members for adding and removing controls

Name	Description
<code>addControl(name, ctrl)</code>	This method adds a control to the <code>FormGroup</code> with the specified name. No action is taken if there is already a control with this name.
<code>setControl(name, ctrl)</code>	This method adds a control to the <code>FormGroup</code> with the specified name, replacing any existing control with this name.
<code>removeControl(name)</code>	This method removes the control with the specified name.
<code>controls</code>	This property returns a map containing the controls in the group, using their names as keys.
<code>get(name)</code>	This property returns the control with the specified name.

In listing 21.14, I created a `FormGroup` and added the existing `FormControl` objects with name and category keys:

```
...
productForm: FormGroup = new FormGroup({
  name: this.nameField, category: this.categoryField
});
...
```

The names used to register `FormControl` objects make it easy to get and set values for all the individual controls in a single step, using the property and methods described in table 21.9.

Table 21.9. `FormGroup` methods for managing control values

Name	Description
<code>value</code>	This method returns an object containing the values of the form controls in the group, using the names given to each control as the names of the properties.
<code>setValue(val)</code>	This method sets the contents of the form controls using an object, whose property names correspond to the names given to each control. The specified value object must define properties for all the form controls in the group.
<code>patchValue(val)</code>	This method sets the contents of the form controls using an object, whose property names correspond to the names given to each control. Unlike the <code>setValue</code> method, values are not required for all form controls.
<code>reset(val)</code>	This method resets the form to its pristine and untouched state and uses the specified value to populate the form controls.

Being able to get and set all the form control values together makes it easier to work with complex forms. I used the `reset` method to populate or clear the form controls when the user clicks the Create New Product or Edit button:

```
...
this.productForm.reset(this.product);
...
```

The `reset` method looks for properties in the object it receives whose names correspond to those used for regular `FormControl` objects with the `FormGroup`. The value of each property is used to set the control value, as shown in figure 21.9.

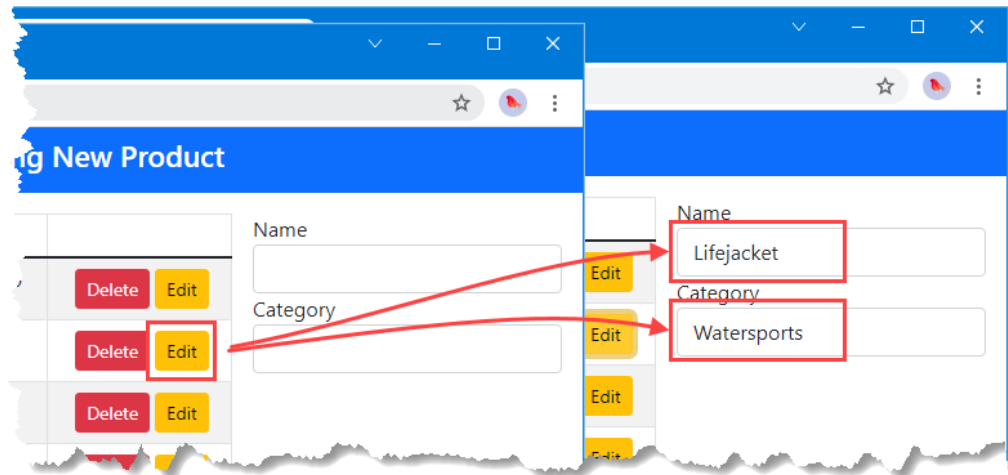


Figure 21.9. Using a form group

The `FormGroup` and `FormControl` classes share a common base class, which means that many of the properties and methods provided by `FormControl` are also available on a `FormGroup` object but apply to all of the controls in the group. In listing 21.14, I subscribed to the `FormGroup`'s `statusChanges` observable to receive events that indicate the status of the form, which Angular determines by examining all of the controls in the group:

```
...
this.productForm.statusChanges.subscribe(newStatus => {
  if (newStatus == "INVALID") {
    ...
  }
})
...
```

If any of the individual controls are invalid, then the overall form status will be invalid, which allows me to assess the validation results without needing to inspect controls individually. But access to the individual controls is still available using the `controls` property, which lets me build up a list of invalid controls:

```
...
for (let controlName in this.productForm.controls) {
  if (this.productForm.controls[controlName].invalid) {
    invalidControls.push(controlName)
  }
}
...
```

The result is that a list of invalid form controls is shown to the user, as illustrated by figure 21.10.

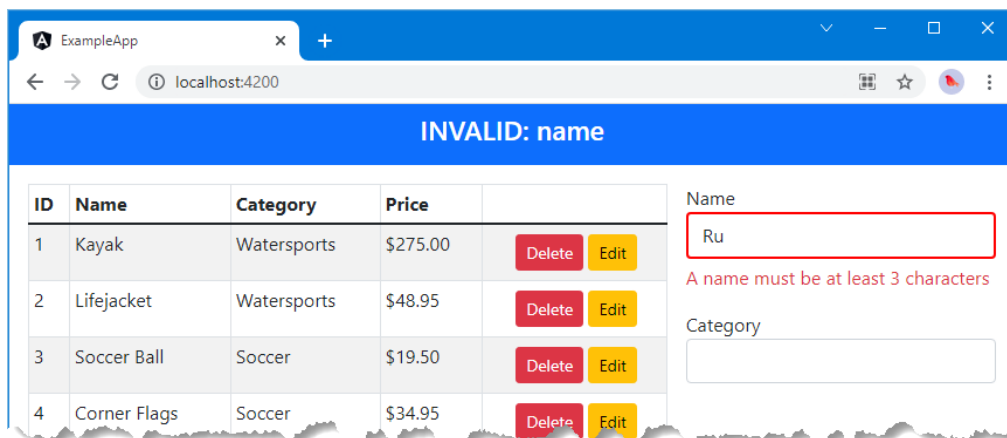


Figure 21.10. Assessing validation status using the form group

21.4.1 Using a form group with a form element

The `formGroup` directive associates a `FormGroup` object with an element in the template, in the same way the `formControl` directive is used with a `FormControl` object, as shown in listing 21.15.

Listing 21.15. Introducing a form element in the `form.component.html` file in the `src/app/core` folder

```
<div [formGroup]="productForm">
  <div class="form-group">
    <label>Name</label>
    <input class="form-control" formControlName="name" />
    <!-- <ul class="text-danger list-unstyled mt-1"
      *ngIf="name.dirty && name.invalid">
        <li *ngFor="let err of name.errors | validationFormat:'name'">
          {{ err }}
        </li>
      </ul> -->
  </div>

  <div class="form-group">
    <label>Category</label>
    <input class="form-control" formControlName="category" />
  </div>
</div>
```

The `formGroup` directive is used to specify the `FormGroup` object, and the individual form elements are associated with their `FormControl` objects using the `formControlName` attribute, specifying the name used when adding the `FormControl` to the `FormGroup`.

Using the `formControlName` attribute means that I don't have to define properties for each `FormControl` object in the controller class, allowing me to simplify the code, as shown in listing 21.16.

Listing 21.16. Removing properties in the form.component.ts file in the src/app/core folder

```

import { Component } from "@angular/core";
import { FormControl, FormGroup, NgForm, 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 } from "../sharedState.service";
import { toObservable } from "@angular/core/rxjs-interop";

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

  // nameField = new FormControl({value: "", disabled: false},
  // { updateOn: "change",
  //   validators: [
  //     Validators.required,
  //     Validators.minLength(3),
  //     Validators.pattern("^[A-Za-z ]+$")
  //   ]});

  // categoryField: FormControl = new FormControl();

  productForm: FormGroup = new FormGroup({
    name: new FormControl("", {
      validators: [
        Validators.required,
        Validators.minLength(3),
        Validators.pattern("^[A-Za-z ]+$")
      ],
      updateOn: "change"
    }),
    category: new FormControl()
  });

  constructor(private model: Model, private stateService: SharedState,
    messageService: MessageService) {

    // ...statements omitted for brevity...
  }
}

```

There is no change in the output produced by the application, and this change just consolidates the individual form controls within the form group.

21.4.2 Accessing the form group from the template

In addition to simplifying the application code, the `formGroup` directive defines some useful properties that allow me to complete the transition to the reactive forms API, restoring the features that were present when the form was managed solely through a template. Table 21.10 describes the most useful features provided by the `formGroup` directive.

Table 21.10. Use features provided by the `formgroup` directive

Name	Description
<code>ngSubmit</code>	This event is triggered when the form is submitted.
<code>submitted</code>	This property returns <code>true</code> if the form has been submitted.
<code>control</code>	This property returns the <code>FormControl</code> object that has been associated with the directive.

These features ensure that the reactive forms API can still be used effectively in a template, while still providing the ability to customize the form behavior in the component class. Listing 21.17 restores the `price` input element that was present at the start of the chapter, along with the buttons that submit and reset the form.

Listing 21.17. Using the `FormGroup` features in the `form.component.html` file in the `src/app/core` folder

```
<form [formGroup]="productForm" #form="ngForm"
      (ngSubmit)="submitForm()" (reset)="resetForm()">

  <div class="form-group">
    <label>Name</label>
    <input class="form-control" formControlName="name" />
  </div>

  <div class="form-group">
    <label>Category</label>
    <input class="form-control" formControlName="category" />
  </div>

  <div class="form-group">
    <label>Price</label>
    <input class="form-control" formControlName="price" />
  </div>

  <div class="mt-2">
    <button type="submit" class="btn btn-primary"
      [class.btn-warning]="editing"
      [disabled]="form.invalid">
      {{editing ? "Save" : "Create"}}
    </button>
    <button type="reset" class="btn btn-secondary m-1">
      Cancel
    </button>
  </div>
```

```
</form>
```

I used the directive's `ngForm` property to create a template variable named `form`, through which I can check the overall validation status for the Save/Create button. The `ngSubmit` form is used to invoke a method named `submitForm`, and I used the form element's `reset` event to invoke a method named `resetForm`. Listing 21.18 shows the changes required to the component class to support the additions to the template.

Listing 21.18. Completing the form in the `form.component.ts` file in the `src/app/core` folder

```
import { Component } from "@angular/core";
import { FormControl, FormGroup, NgForm, 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 } from "../sharedState.service";
import { toObservable } from "@angular/core/rxjs-interop";

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

  productForm: FormGroup = new FormGroup({
    name: new FormControl("", {
      validators: [
        Validators.required,
        Validators.minLength(3),
        Validators.pattern("^[A-Za-z ]+$")
      ],
      updateOn: "change"
    }),
    category: new FormControl("", { validators: Validators.required }),
    price: new FormControl("", {
      validators: [
        Validators.required,
        Validators.pattern("^[0-9\\.]+$")
      ]
    })
  });

  constructor(private model: Model, private stateService: SharedState,
    messageService: MessageService) {

    toObservable(stateService.state).subscribe(state => {
      this.editing = state.mode == MODES.EDIT;
      if (this.editing && state.id) {
        this.product = Product.fromProduct(
          this.model.getProduct(state.id) ?? new Product());
      }
    });
  }
}
```

```

    } else {
      this.product = new Product;
    }
    this.productForm.reset(this.product);
    messageService.reportMessage(state.id
      ? new Message(`Editing ${this.product.name}`)
      : new Message("Creating New Product"));
  });

  // this.productForm.statusChanges.subscribe(newStatus => {
  //   if (newStatus == "INVALID") {
  //     let invalidControls: string[] = [];
  //     for (let controlName in this.productForm.controls) {
  //       if (this.productForm.controls[controlName].invalid) {
  //         invalidControls.push(controlName)
  //       }
  //     }
  //     messageService.reportMessage(
  //       new Message(`INVALID: ${invalidControls.join(", ")}`)
  //     )
  //   } else {
  //     messageService.reportMessage(new Message(newStatus));
  //   }
  // })
}

submitForm() {
  if (this.productForm.valid) {
    Object.assign(this.product, this.productForm.value);
    this.model.saveProduct(this.product);
    this.product = new Product();
    this.productForm.reset();
  }
}

resetForm() {
  this.editing = true;
  this.product = new Product();
  this.productForm.reset();
}
}

```

This listing adds a `FormControl` named `price`, adds validation to the `category` control and defines the `submitForm` and `resetForm` method that will be invoked by the event bindings defined in listing 21.17. The effect is to complete the form and restore the functionality that was previously defined using only the template form features, as shown in figure 21.11.

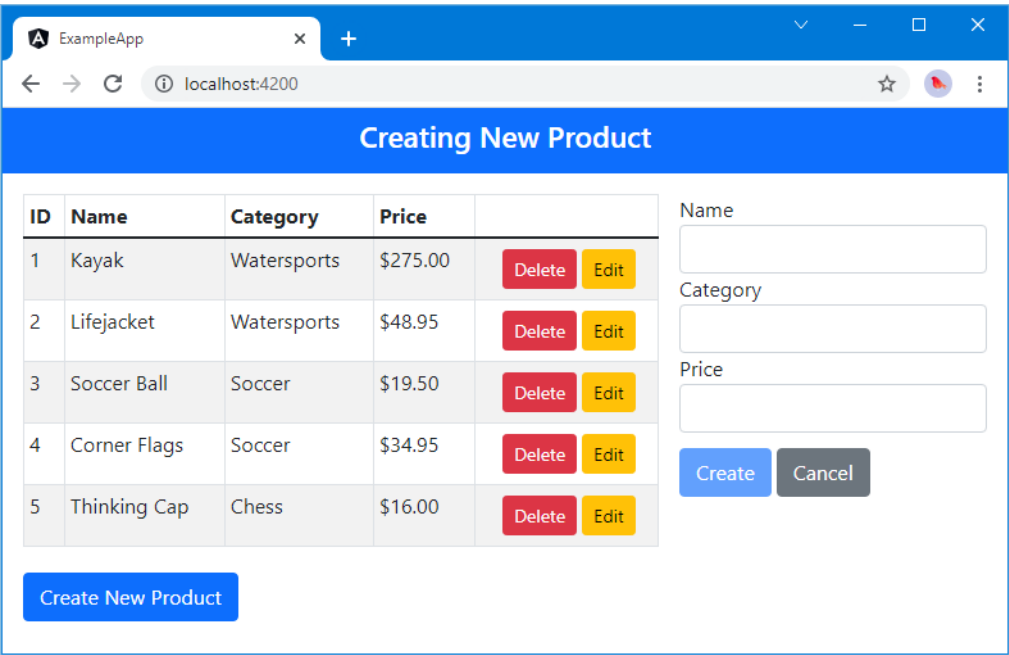


Figure 21.11. Using the reactive forms API

21.4.3 Displaying validation messages with a form group

The `formControlName` directive doesn't export an identifier for use in a template variable, which complicates the process of displaying validation messages. Instead, errors must be obtained through the `FormGroup`, using the optional `path` argument for the error-related methods, which I have repeated in table 21.11 for quick reference.

Table 21.11. The `FormGroup` methods for dealing with errors

Name	Description
<code>getError(v, path)</code>	This method returns the error message, if there is one, for the specified validator. The optional <code>path</code> argument is used to identify the control.
<code>hasError(v, path)</code>	This method returns <code>true</code> if the specified validator has generated an error message. The optional <code>path</code> argument is used to identify the control.

These methods require the error to be specified, which means that it is possible to determine if a specific control has a specific error, like this:

```
...
form.getError("required", "category")
...
```


This expression would return details of errors reported by the `required` validator on the `category` control, which is identified by the name used to register the control in the `FormGroup`. This isn't a useful approach for the example application, where I want to display validation messages by getting all of the errors for a single `FormControl` object. For this, I can use the `get` method defined by the `FormGroup` class, although this can produce verbose and repetitive templates and so the best approach is to create a directive. Add a file named `validationErrors.directive.ts` to the `src/app/core` folder with the code shown in listing 21.19.

Listing 21.19. The `validationErrors.directive.ts` file in the `src/app/core` folder

```
import { Directive, Input, TemplateRef, ViewContainerRef }
    from "@angular/core";
import { FormGroup } from "@angular/forms";
import { ValidationHelper } from "../validationHelper.pipe";

@Directive({
    selector: "[validationErrors]"
})
export class ValidationErrorsDirective {

    constructor(private container: ViewContainerRef,
                private template: TemplateRef<Object>) { }

    @Input("validationErrorsControl")
    name: string = ""

    @Input("validationErrorsLabel")
    label?: string;

    @Input("validationErrors")
    formGroup?: FormGroup;

    ngOnInit() {
        let formatter = new ValidationHelper();
        if (this.formGroup && this.name) {
            let control = this.formGroup?.get(this.name);
            if (control) {
                control.statusChanges.subscribe(() => {
                    if (this.container.length > 0) {
                        this.container.clear();
                    }
                    if (control && control.dirty && control.invalid
                        && control.errors) {
                        formatter.format(control.errors,
                            this.label ?? this.name).forEach(err => {
                            this.container.createEmbeddedView(
                                this.template,
                                { $implicit: err });
                        });
                    }
                });
            }
        }
    }
}
```

```

    }
  }
}

```

The new directive obtains a `FormControl` object via its `FormGroup` and subscribes to the observable for status changes. Each time the status changes, the validation state is checked, and any error messages are formatted and used to generate template content.

This is a simple task in code, but it is more difficult to achieve in a template without creating long expressions. Listing 21.20 registers the directive so that it can be used in the module.

Listing 21.20. Registering a directive in the `core.module.ts` file in the `src/app/core` folder

```

import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule, ReactiveFormsModule } from "@angular/forms";
import { ModelModule } from "../model/model.module";
import { TableComponent } from "../table.component";
import { FormComponent } from "../form.component";
import { SharedState } from "../sharedState.service";
import { ValidationHelper } from "../validationHelper.pipe";
import { ValidationErrorsDirective } from "../validationErrors.directive";

@NgModule({
  imports: [BrowserModule, FormsModule, ModelModule,
    ReactiveFormsModule],
  declarations: [TableComponent, FormComponent, ValidationHelper,
    ValidationErrorsDirective],
  exports: [ModelModule, TableComponent, FormComponent],
  providers: [SharedState]
})
export class CoreModule { }

```

Listing 21.21 uses the new directive to display validation messages for each form element.

Listing 21.21. Displaying validation messages in the `form.component.html` file in the `src/app/core` folder

```

<form [formGroup]="productForm" #form="ngForm"
      (ngSubmit)="submitForm()" (reset)="resetForm()">
  <div class="form-group">
    <label>Name</label>
    <input class="form-control" formControlName="name" />
    <ul class="text-danger list-unstyled mt-1">
      <li *validationErrors="productForm; control:'name'; let err">
        {{ err }}
      </li>
    </ul>
  </div>

  <div class="form-group">
    <label>Category</label>
    <input class="form-control" formControlName="category" />
    <ul class="text-danger list-unstyled mt-1">
      <li *validationErrors=
        "productForm; control:'category'; let err">
        {{ err }}
      </li>
    </ul>
  </div>

```

```

    </ul>
  </div>

  <div class="form-group">
    <label>Price</label>
    <input class="form-control" formControlName="price" />
    <ul class="text-danger list-unstyled mt-1">
      <li *validationErrors="productForm; control:'price'; let err">
        {{ err }}
      </li>
    </ul>
  </div>

  <div class="mt-2">
    <button type="submit" class="btn btn-primary"
      [class.btn-warning]="editing"
      [disabled]="form.invalid">
      {{editing ? "Save" : "Create"}}
    </button>
    <button type="reset" class="btn btn-secondary m-1">
      Cancel
    </button>
  </div>
</form>

```

The directive is configured with the `FormGroup` property defined by the component class and the name of the `FormControl` object for which errors are required. This is an indirect way of working, but it works as seamlessly as dealing with individual elements once the building blocks are in place, as shown in figure 21.12.

The screenshot shows a web browser window with the title 'ExampleApp' and the address 'localhost:4200'. The page is titled 'Creating New Product'. It features a table with 5 rows of product data. To the right of the table is a form for creating a new product. The form has three input fields: 'Name' (containing 'r2'), 'Category' (empty), and 'Price' (containing 'a'). The 'Name' and 'Price' fields have red borders and red error messages below them. The 'Price' field also has a red error message. At the bottom of the form are 'Create' and 'Cancel' buttons. A 'Create New Product' button is located at the bottom left of the page.

ID	Name	Category	Price	
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	Thinking Cap	Chess	\$16.00	Delete Edit

Name: r2
A name must be at least 3 characters
The name contains illegal characters

Category:

Price: a
The price contains illegal characters

Create New Product

Create Cancel

Figure 21.12. Displaying validation messages from a form group

21.5 Summary

In this chapter, I introduced the Angular reactive forms API and showed you how it can be used to create and manage forms, providing a more code-centered approach than the standard template-driven forms described in chapter 13.

- Angular provides an API for working with form elements.
- The form API allows control validation and behavior to be defined in code.
- The API provides events that describe changes in control content and validation state.
- Controls can be grouped and operated on as a single unit.

In the next chapter, I continue to describe the forms API, explaining how to create controls dynamically and how to create custom validators.

22

Using the forms API, part 2

This chapter covers

- Creating form components dynamically
- Validating dynamically created form components
- Using the asynchronous validation features
- Writing custom form validators.

In this chapter, I continue to describe the Angular forms API, explaining how to create form controls dynamically and how to create custom validation. Table 22.1 summarizes the chapter.

Table 22.1. Chapter summary

Problem	Solution	Listing
Creating and managing form controls dynamically	Use a <code>FormArray</code> object	1–7
Validating dynamically created form controls	Use a control's position in its enclosing <code>FormArray</code> as identification during the validation process	8, 9
Altering the values produced by dynamically created controls	Override the methods defined by the <code>FormArray</code> class	10, 11
Creating custom form validation	Create a function that returns an implementation of the <code>ValidatorFn</code> interface, which performs validation on a control's value	12–14
Applying a custom validator in a template-driven form	Create a directive that calls the validator function	15–18

Validating multiple related fields	Perform validation on a <code>FormGroup</code> or <code>FormArray</code>	19–23
Performing complex or remote validation	Create an asynchronous validator	24–27

22.1 Preparing for this chapter

For this chapter, I will continue using the `exampleApp` project from chapter 21. No changes are required for this chapter.

TIP You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/manningbooks/pro-angular-16>. See chapter 1 for how to get help if you have problems running the examples.

To start the development server, open a command prompt, navigate to the `exampleApp` folder, and run the following command:

```
ng serve
```

Open a new browser window and navigate to `http://localhost:4200` to see the content shown in figure 22.1.

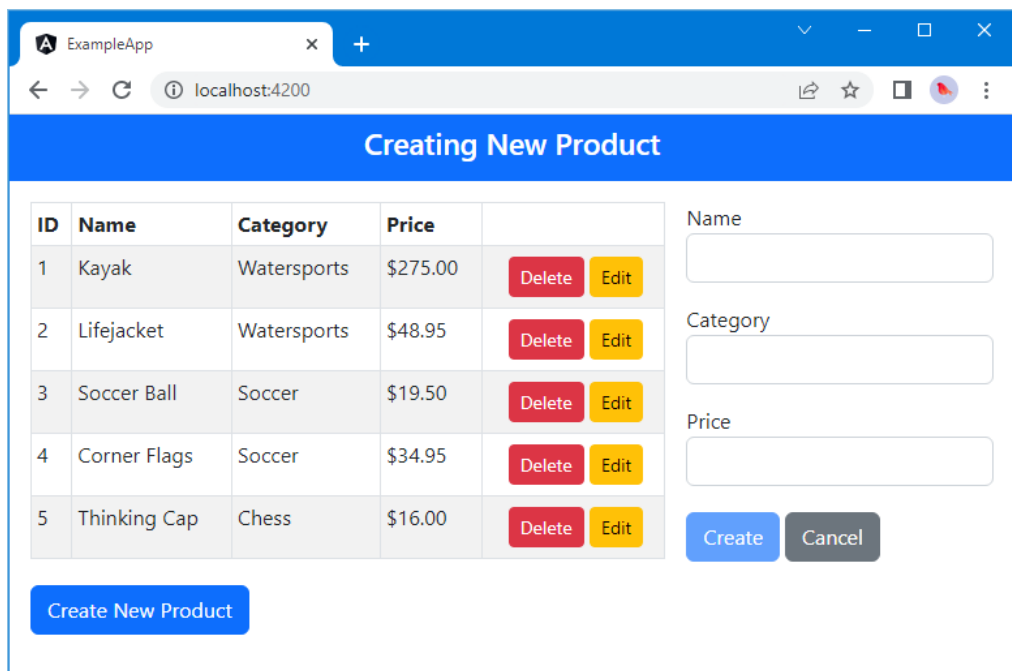


Figure 22.1. Running the example application

22.2 Creating form components dynamically

The `FormGroup` class is useful when the structure and number of elements in the form are known in advance. For applications that need to dynamically add and remove elements, Angular provides the `FormArray` class. Both `FormArray` and `FormControl` are derived from the `AbstractControl` class and provide the same features for managing `FormGroup` objects; the difference is that the `FormArray` class allows `FormControl` objects to be created without specifying names and stores its controls as an array, making it easier to add and remove controls. To prepare, listing 22.1 adds an array of keywords to the `Product` model class.

Listing 22.1. Adding a property in 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 price?: number,
               public keywords?: string[]) { }

  static fromProduct(p: Product) {
    return new Product(p.id, p.name, p.category, p.price, p.keywords);
  }
}
```

Listing 22.2 updates the static example data to reflect the addition of the `keywords` property.

Listing 22.2. Updating data in the `static.datasource.ts` file in the `src/app/model` folder

```
import { Injectable } from "@angular/core";
import { Product } from "../product.model";

@Injectable()
export class StaticDataSource {
  private data: Product[];

  constructor() {
    this.data = new Array<Product>(
      new Product(1, "Kayak", "Watersports", 275, ["boat", "small"]),
      new Product(2, "Lifejacket", "Watersports", 48.95, ["safety"]),
      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;
  }
}
```

Listing 22.3 adds a new column to the template for the table component so the keywords are displayed to the user.

Listing 22.3. Adding a column in the `table.component.html` file in the `src/app/core` folder

```

<table class="table table-sm table-bordered table-striped">
  <thead>
    <tr>
      <th>ID</th><th>Name</th><th>Category</th>
      <th>Price</th><th>Keywords</th><th></th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let item of Products()">
      <td>{{item.id}}</td>
      <td>{{item.name}}</td>
      <td>{{item.category}}</td>
      <td>{{item.price | currency:"USD" }}</td>
      <td>{{ item.keywords?.join(", ") }}</td>
      <td class="text-center">
        <button class="btn btn-danger btn-sm m-1"
          (click)="deleteProduct(item.id)">
          Delete
        </button>
        <button class="btn btn-warning btn-sm"
          (click)="editProduct(item.id)">
          Edit
        </button>
      </td>
    </tr>
  </tbody>
</table>
<button class="btn btn-primary mt-1" (click)="createProduct()">
  Create New Product
</button>

```

22.2.1 Using a form array

The `FormArray` class stores its child controls in an array and provides the properties and methods described in table 22.2 for managing the array, in addition to those it inherits from the `AbstractControl` class and that are shared with the `FormGroup` and `FormControl` classes.

Table 22.2. Useful `FormArray` members for managing controls

Name	Description
<code>controls</code>	This property returns an array containing the child controls.
<code>length</code>	This property returns the number of controls that are in the <code>FormArray</code> .
<code>at(index)</code>	This property returns the control at the specified index in the <code>FormArray</code> .
<code>push(control)</code>	This method adds a control to the end of the array.
<code>insert(index, control)</code>	This method inserts a control at the specified index.

<code>setControl(index, control)</code>	This method replaces the control at the specified index.
<code>removeAt(index)</code>	This method removes the control at the specified index.
<code>clear()</code>	This method removes all of the controls from the <code>FormArray</code> .

The `FormArray` class also provides methods for setting the values of the controls it manages using arrays, rather than name-value maps, as described in table 22.3.

Table 22.3. The `FormArray` methods for setting values

Name	Description
<code>setValue(values)</code>	This method accepts an array of values and uses them to set the values of the child controls based on the order in which they are defined. The number of elements in the values array must match the number of controls in the <code>FormArray</code> .
<code>patchValue(values)</code>	This method accepts an array of values and uses them to set the values of the child controls based on the order in which they are defined. Unlike the <code>setValue</code> method, the number of elements in the values array does not have to match the number of controls in the <code>FormArray</code> , and this method will ignore values for which there are no controls and will ignore controls for which there are no values.
<code>reset(values)</code>	This method resets the controls in the <code>FormArray</code> and sets their values using the optional array argument. The number of elements in the values array does not have to match the number of controls in the <code>FormArray</code> . Values for which there are no controls are ignored, and controls for which there are no values are reset to their default state.

Listing 22.4 uses the features described in these tables to vary the number of controls displayed for the `keywords` model property.

Listing 22.4. Using a `FormArray` in the `form.components.ts` file in the `src/app/core` folder

```
import { Component } from "@angular/core";
import { FormArray, FormControl, FormGroup, NgForm, 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 } from "../sharedState.service";
import { toObservable } from "@angular/core/rxjs-interop";

@Component({
  selector: "paForm",
  templateUrl: "form.component.html",
  styleUrls: ["form.component.css"]
})
```

```

    })
    export class FormComponent {
        product: Product = new Product();
        editing: boolean = false;

        keywordGroup = new FormArray([
            this.createKeywordFormControl()
        ]);

        productForm: FormGroup = new FormGroup({
            name: new FormControl("", {
                validators: [
                    Validators.required,
                    Validators.minLength(3),
                    Validators.pattern("[A-Za-z ]+$")
                ],
                updateOn: "change"
            }),
            category: new FormControl("", { validators: Validators.required }),
            price: new FormControl("", {
                validators: [
                    Validators.required,
                    Validators.pattern("[0-9\\.]+$")
                ]
            }),
            keywords: this.keywordGroup
        });

        constructor(private model: Model, private stateService: SharedState,
            messageService: MessageService) {

            toObservable(stateService.state).subscribe(state => {
                this.editing = state.mode == MODES.EDIT;
                if (this.editing && state.id) {
                    this.product = Product.fromProduct(
                        this.model.getProduct(state.id) ?? new Product() );
                } else {
                    this.product = new Product;
                }
                this.createKeywordFormControls(this.product.keywords?.length);
                this.productForm.reset(this.product);
                messageService.reportMessage(state.id
                    ? new Message(`Editing ${this.product.name}`)
                    : new Message("Creating New Product"));
            });
        }

        submitForm() {
            if (this.productForm.valid) {
                Object.assign(this.product, this.productForm.value);
                this.model.saveProduct(this.product);
                this.product = new Product();
                this.productForm.reset();
            }
        }

        resetForm() {

```

```

        this.editing = true;
        this.product = new Product();
        this.productForm.reset();
    }

    createKeywordFormControls(count: number = 0) {
        this.keywordGroup.clear();
        for (let i = 0; i < count + 1; i++) {
            this.keywordGroup.push(this.createKeywordFormControl());
        }
    }

    createKeywordFormControl() {
        return new FormControl();
    }
}

```

I have defined a `FormArray` property so that I can access it in the template, which is important because there are no built-in directives that export the `FormArray` for use as a template variable:

```

...
keywordGroup = new FormArray([
    this.createKeywordFormControl()
]);
...

```

The `FormArray` is initialized with the initial set of controls it will manage, expressed as an array. Controls are not given names, and the features described in table 22.2 and table 22.3 all work on arrays. Consistency is important when creating controls, so I defined a method named `createKeywordFormControl` that is responsible for instantiating `FormControl` objects for keywords:

```

...
createKeywordFormControl() {
    return new FormControl();
}
...

```

Using a method to create the `FormControl` objects ensures that I can easily alter the control configuration without having to figure out all of the places where controls are created. To populate the form with controls for keywords, I defined the `createKeywordFormControls` method, which creates the `FormControl` objects:

```

...
createKeywordFormControls(count: number = 0) {
    this.keywordGroup.clear();
    for (let i = 0; i < count + 1; i++) {
        this.keywordGroup.push(this.createKeywordFormControl());
    }
}
...

```

NOTE Angular includes the `FormBuilder` class, which can be used to simplify creating and configuring `FormArray` objects and the controls it contains. I don't find this class useful, which is why I don't describe it in this chapter, but you may feel differently. See <https://angular.io/api/forms/FormBuilder> for details.

The `FormArray` is added to the overall structure of controls in the same way as a nested `FormGroup`:

```
...
productForm: FormGroup = new FormGroup({
  name: new FormControl("", {
    validators: [
      Validators.required,
      Validators.minLength(3),
      Validators.pattern("^[A-Za-z ]+$")
    ],
    updateOn: "change"
  }),
  category: new FormControl("", { validators: Validators.required }),
  price: new FormControl("", {
    validators: [
      Validators.required,
      Validators.pattern("[0-9\\.]+$")
    ]
  }),
  keywords: this.keywordGroup
});
...
```

Within the component, I can manage the array of controls in the `FormArray` to match the selected `Product` object, ensuring that there is at least one empty control in the array so the user can add values to `Product` objects that don't currently have any `keyword` values.

Listing 22.5 uses the `FormArray` property to create the HTML elements to match the number of `FormControl` objects.

Listing 22.5. New elements in the `form.component.html` file in the `src/app/core` folder

```
<form [formGroup]="productForm" #form="ngForm"
      (ngSubmit)="submitForm()" (reset)="resetForm()">
  <div class="form-group">
    <label>Name</label>
    <input class="form-control" formControlName="name" />
    <ul class="text-danger list-unstyled mt-1">
      <li *validationErrors="productForm; control:'name'; let err">
        {{ err }}
      </li>
    </ul>
  </div>

  <div class="form-group">
    <label>Category</label>
    <input class="form-control" formControlName="category" />
    <ul class="text-danger list-unstyled mt-1">
      <li *validationErrors=
        "productForm; control:'category'; let err">
        {{ err }}
      </li>
    </ul>
  </div>

  <div class="form-group">
```

```

        <label>Price</label>
        <input class="form-control" formControlName="price" />
        <ul class="text-danger list-unstyled mt-1">
            <li *validationErrors="productForm; control:'price'; let err">
                {{ err }}
            </li>
        </ul>
    </div>

    <div formGroupName="keywords">
        <div class="form-group" *ngFor="let c of keywordGroup.controls;
            let i = index">
            <label>Keyword {{ i + 1 }}</label>
            <input class="form-control"
                [formControlName]="i" [value]="c.value" />
        </div>
    </div>

    <div class="mt-2">
        <button type="submit" class="btn btn-primary"
            [class.btn-warning]="editing"
            [disabled]="form.invalid">
            {{editing ? "Save" : "Create"}}
        </button>
        <button type="reset" class="btn btn-secondary m-1">
            Cancel
        </button>
    </div>
</form>

```

It is important to reflect the structure of the `FormGroup` and `FormArray` objects when creating HTML elements, ensuring that each is correctly configured with the `formGroupName` directive. I used the `ng-container` element to avoid introducing an HTML element for the `FormArray` object and used the `ngFor` directive to create elements for each `FormControl` in the `FormArray`:

```

...
<div class="form-group" *ngFor="let c of keywordGroup.controls;
    let i = index">
...

```

Each input element must be configured with the `formControlName` directive, using an array position as its value, instead of a name:

```

...
<input class="form-control" [formControlName]="i" [value]="c.value" />
...

```

The result is that the number of form controls displayed to the user varies based on the `Product` value that is selected, as shown in figure 22.2. Notice that Angular correctly populates the input elements through the `FormArray`, mapping the values in the `keywords` model array to the elements in the form.

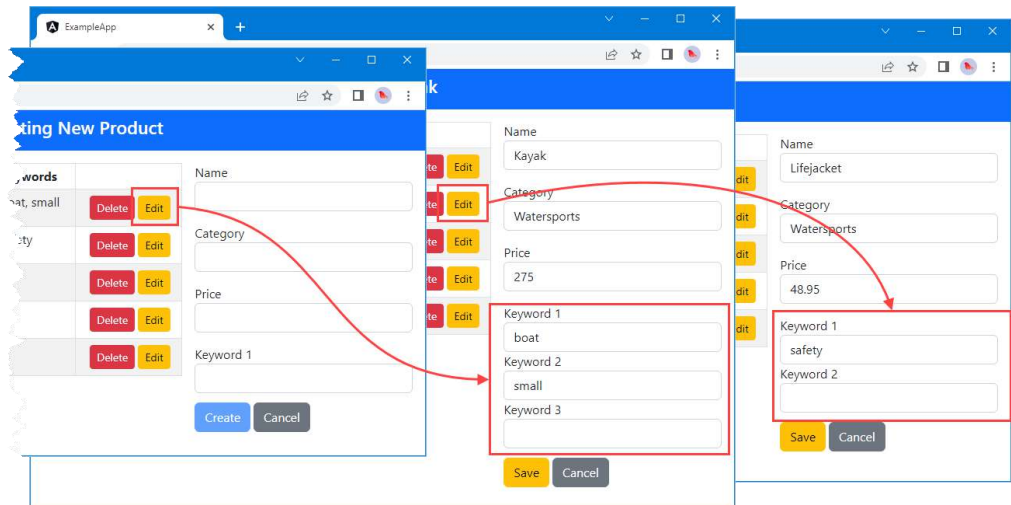


Figure 22.2. Using a form array

22.2.2 Adding and removing form controls

To complete the support for multiple keywords, I am going to allow the user to add and remove controls. Listing 22.6 adds methods to the control class.

Listing 22.6. Adding methods in the form.component.ts file in the src/app/core folder

```
import { Component } from "@angular/core";
import { FormArray, FormControl, FormGroup, NgForm, 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 } from "../sharedState.service";
import { toObservable } from "@angular/core/rxjs-interop";

@Component({
  selector: "paForm",
  templateUrl: "form.component.html",
  styleUrls: ["form.component.css"]
})
export class FormComponent {

  // ...statements omitted for brevity...

  addKeywordControl() {
    this.keywordGroup.push(this.createKeywordFormControl());
  }

  removeKeywordControl(index: number) {
```

```

        this.keywordGroup.removeAt(index);
    }
}

```

The new methods use the `FormArray` features described in table 22.2 to add and remove `FormGroup` objects. Listing 22.7 adds elements to the template that will invoke the new component methods and allow the user to manage the number of `keywords` fields.

Listing 22.7. Adding elements in the `form.component.html` file in the `src/app/core` folder

```

...
<div formGroupName="keywords">
  <button class="btn btn-sm btn-primary my-2"
    (click)="addKeywordControl()" type="button">
    Add Keyword
  </button>
  <div class="form-group" *ngFor="let c of keywordGroup.controls;
    let i = index; let count = count">
    <label>Keyword {{ i + 1 }}</label>
    <div class="input-group">
      <input class="form-control"
        [formControlName]="i" [value]="c.value" />
      <button class="btn btn-danger" type="button"
        *ngIf="count > 1" (click)="removeKeywordControl(i)">
        Delete
      </button>
    </div>
  </div>
</div>
...

```

I use the `count` variable exported by the `ngForm` directive to display a Delete button only when there are multiple controls in the form array. The number of keyword fields will be initially determined by the selected `Product` object, after which the user can add and remove fields, as shown in figure 22.3.

The screenshot shows a web browser window with the address bar displaying 'localhost:4200'. The page title is 'Editing Lifejacket'. It features a table with 5 columns: ID, Name, Category, Price, and Keywords. The table contains 5 rows of product data. To the right of the table is a form for editing a product. The form has input fields for Name, Category, and Price. Below these is a section for keywords, labeled 'Add Keyword', which contains two rows for 'Keyword 1' and 'Keyword 2', each with a text input and a 'Delete' button. At the bottom of the form are 'Save' and 'Cancel' buttons. A red box highlights the keyword section, and a red arrow points from the 'Keywords' column of the table to this section.

ID	Name	Category	Price	Keywords	
1	Kayak	Watersports	\$275.00	boat, small	Delete Edit
2	Lifejacket	Watersports	\$48.95	safety	Delete Edit
3	Soccer Ball	Soccer	\$19.50		Delete Edit
4	Corner Flags	Soccer	\$34.95		Delete Edit
5	Thinking Cap	Chess	\$16.00		Delete Edit

Form fields:

- Name: Lifejacket
- Category: Watersports
- Price: 48.95
- Add Keyword button
- Keyword 1: safety (Delete button)
- Keyword 2: (Delete button)
- Save button
- Cancel button

Figure 22.3. Adding and removing form controls in a form array

22.2.3 Validating dynamically created form controls

Validation for the controls in a `FormArray` is similar to validating the controls in a `FormGroup`, as shown in listing 22.8.

Listing 22.8. Adding validation in the `form.component.ts` file in the `src/app/core` folder

```
...
createKeywordFormControl() {
  return new FormControl("", {
    validators: Validators.pattern("[A-Za-z ]+$")
  });
}
...
```

The advantage of using a method to create `FormControl` objects for the `FormArray` is that I can define the validation policy in a single place. Listing 22.9 displays validation messages to the user.

Listing 22.9. Displaying validation messages in the `form.component.html` file in the `src/app/core` folder

```
...
<div formGroupName="keywords">
```



```

<button class="btn btn-sm btn-primary my-2"
  (click)="addKeywordControl()" type="button">
  Add Keyword
</button>
<div class="form-group" *ngFor="let c of keywordGroup.controls;
  let i = index; let count = count">
  <label>Keyword {{ i + 1 }}</label>
  <div class="input-group">
    <input class="form-control"
      [formControlName]="i" [value]="c.value" />
    <button class="btn btn-danger" type="button"
      *ngIf="count > 1" (click)="removeKeywordControl(i)">
      Delete
    </button>
  </div>
  <ul class="text-danger list-unstyled mt-1">
    <li *validationErrors="productForm;
      control:'keywords.' + i; label: 'keyword'; let err">
      {{ err }}
    </li>
  </ul>
</div>
</div>
...

```

The path to the control uses the position in the array, rather than a name, like this:

```

...
<li *validationErrors="productForm; control:'keywords.' + i;
  label: 'keyword'; let err">
...

```

It is important to ensure you specify the correct position; otherwise, you will display validation messages for a different control. The user is presented with a validation error if a disallowed character is entered into a keyword field, as shown in figure 22.4.

Add Keyword

Keyword 1

boat2

The keyword contains illegal characters

Create Cancel

Figure 22.4. Validation for a form array control

22.2.4 Filtering the FormArray values

When dealing with variable numbers of controls in a `FormArray`, the user may not enter values in all of the controls, which can cause a problem when processing the contents of the form. Figure 22.5 illustrates the problem.

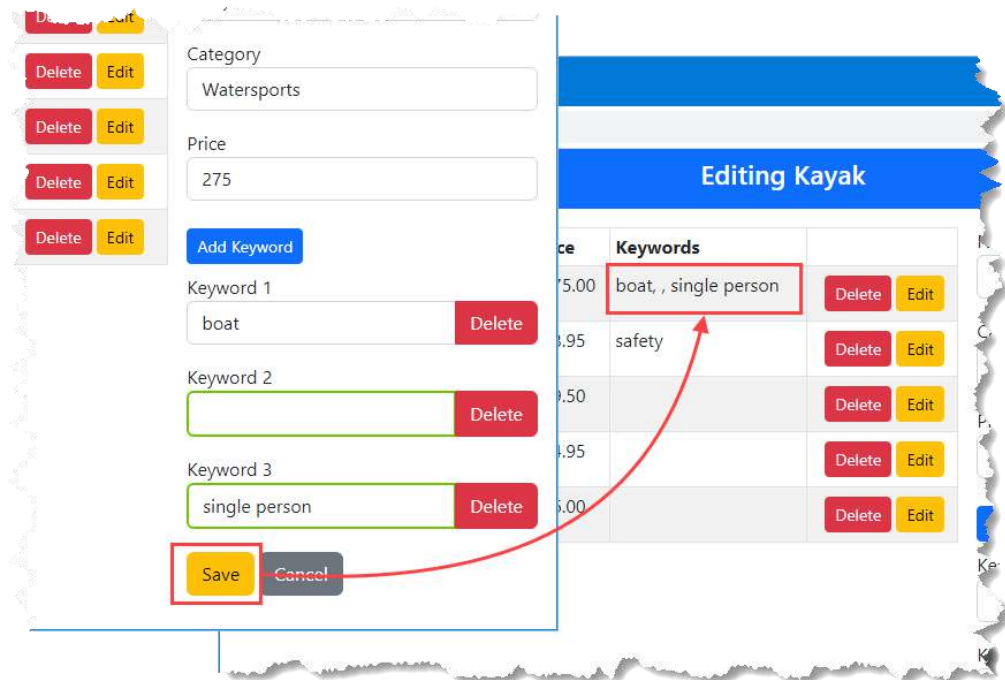


Figure 22.5. The effect of empty fields in form array controls

I left one of the `keyword` fields empty when I submitted the form, which means that an empty string has been included in the array of values assigned to the `Product` model object's `keywords` field.

I could prevent this problem using the `required` validator, but this requires the user to remove any empty controls before submitting the form, which would be an awkward interruption to their workflow.

My preference is to give the user some flexibility and create a custom class that will filter out unwanted values. Add a file named `filteredFormArray.ts` to the `src/app/core` folder with the contents shown in listing 22.10.

Listing 22.10. The contents of the `filteredFormArray.ts` file in the `src/app/core` folder

```
import { FormArray } from "@angular/forms";
```

```

export type ValueFilter = (value: any) => boolean;

export class FilteredFormArray extends FormArray {
  filter: ValueFilter | undefined = (val) => val == "" || val == null;

  _updateValue() {
    (this as {value: any}).value =
      this.controls.filter((control) =>
        (control.enabled || this.disabled)
        && !this.filter?.(control.value)
      ).map((control) => control.value);
  }
}

```

The `FilteredFormArray` class defines an `_updateValue` method, which applies a filter function that, by default, excludes empty string and `null` values.

The code in listing 22.10 is on the edge of what I would consider acceptable meddling with the Angular API. You won't see the `_updateValue` method in the API documentation for the `FormArray` class because it is part of the internal API, which was originally defined as an abstract method in the `AbstractControl` class and then overridden in the `FormArray` class. These methods are marked as internal, and I located them by looking at the Angular source code to figure out how these classes set the `value` property.

There are two issues with using methods like this. The first is that internal methods are subject to change or removal without notice, which means that future releases of Angular may remove the `_updateValue` method and break the code in listing 22.10.

The second issue is that the Angular packages are compiled using a TypeScript setting that excludes internal methods from the type declaration files that are used during project development. This means that the TypeScript compiler doesn't know that the `FormArray` class defines an `_updateValue` method and won't allow the use of the `override` or the `super` keywords. For this reason, I have had to copy the original code from the `FormArray` class and integrate support for filtering, rather than just calling the `FormArray` implementation of the method and filtering the result.

I am comfortable with these issues when it comes to small changes in functionality. You must make your own assessment of the issues and decide whether relying on internal features is reasonable for your projects.

But, even if you are not comfortable using this approach in your projects, this example does let me illustrate that, once again, there is nothing magical about the way that Angular works. In this case, I relied on the fact that Angular applications are compiled into pure JavaScript and that the JavaScript rules for locating a method apply, even if TypeScript has been configured to exclude the method from the type declaration files.

Listing 22.11 replaces the standard `FormArray` object with one that filters values.

Listing 22.11. Using a customized form array in the `form.component.ts` file in the `src/app/core` folder

```

import { Component } from "@angular/core";
import { FormArray, FormControl, FormGroup, NgForm, 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 } from "../sharedState.service";
import { toObservable } from "@angular/core/rxjs-interop";
import { FilteredFormArray } from "../filteredFormArray";

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

  keywordGroup = new FilteredFormArray([
    this.createKeywordFormControl()
  ]);

  // ...statements omitted for brevity...
}

```

The use of the filter prevents empty values from being included in the keywords array, as shown in figure 22.6.

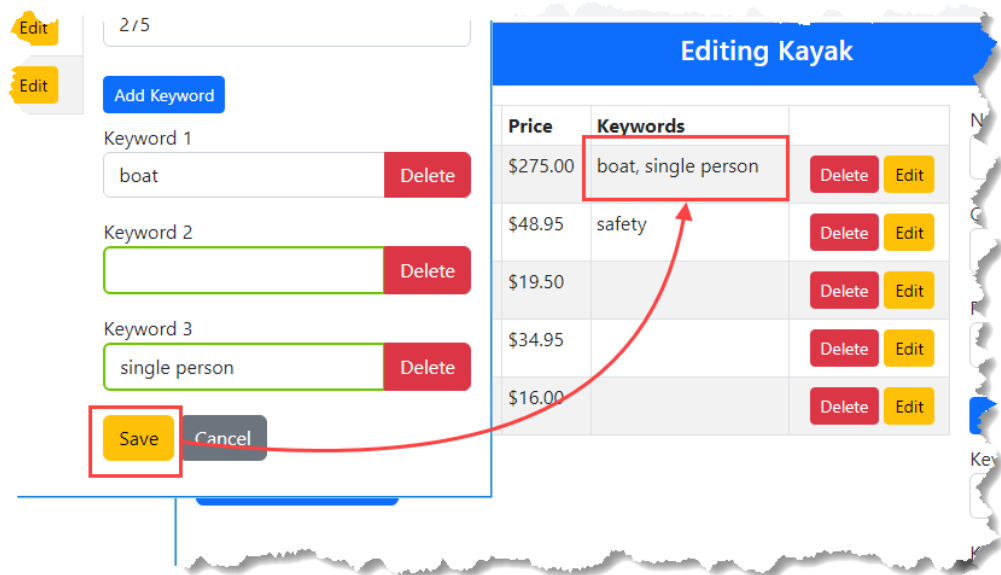


Figure 22.6. Filtering values

22.3 Creating custom form validation

Angular supports custom form validators, which can be used to enforce a validation policy that is specific to the application, rather than the general-purpose validation that the built-in validators provide. A good way to understand how custom validation works is to re-create the functionality provided by one of the built-in validators.

Create the `src/app/validation` folder and add to it a file named `limit.ts` with the contents shown in listing 22.12.

Listing 22.12. The contents of the `limit.ts` file in the `src/app/validation` folder

```
import { AbstractControl, ValidationErrors, ValidatorFn }
  from "@angular/forms";

export class LimitValidator {

  static Limit(limit:number) : ValidatorFn {
    return (control: AbstractControl) : ValidationErrors | null => {
      let val = parseFloat(control.value);
      if (isNaN(val) || val > limit) {
        return {"limit": {"limit": limit, "actualValue": val}};
      }
      return null;
    }
  }
}
```

Custom validators are functions that implement the `ValidatorFn` interface, which describes a factory function that creates functions that perform validation. The factory function accepts parameters that allow validation to be configured and returns functions that accept an `AbstractControl` parameter and return a `ValidationErrors | null` result:

```
...
static Limit(limit:number) : ValidatorFn {
  return (control: AbstractControl) : ValidationErrors | null => {
...

```

The factory function in this example is named `Limit`, and it defines a parameter named `limit` that specifies a maximum acceptable value, similar to the way that the built-in `min` validator works.

Listing 22.13 adds support for translating the validation results produced by the custom validator into messages that can be displayed to the user.

Listing 22.13. Adding support for a new validator in the `validationHelper.pipe.ts` file in the `src/app/core` folder

```
import { Pipe } from "@angular/core";
import { FormControl, ValidationErrors } from "@angular/forms";

@Pipe({
  name: "validationFormat"
})
export class ValidationHelper {

  transform(source: any, name: any) : string[] {
```

```

        if (source instanceof FormControl) {
            return this.format((source as FormControl).errors, name)
        }
        return this.format(source as ValidationErrors, name)
    }

    format(errors: ValidationErrors | null, name: string): string[] {
        let messages: string[] = [];
        for (let errorName in errors) {
            switch (errorName) {
                case "required":
                    messages.push(`You must enter a ${name}`);
                    break;
                case "minlength":
                    messages.push(`A ${name} must be at least
                        ${errors['minlength'].requiredLength}
                        characters`);
                    break;
                case "pattern":
                    messages.push(`The ${name} contains
                        illegal characters`);
                    break;
                case "limit":
                    messages.push(`The ${name} must be less than
                        ${errors['limit'].limit}`);
                    break;
            }
        }
        return messages;
    }
}

```

Custom validators are applied in the same way as those built into Angular, as shown in listing 22.14.

Listing 22.14. Using a custom validator in the form.component.ts file in the src/app/core folder

```

import { Component } from "@angular/core";
import { FormArray, FormControl, FormGroup, NgForm, 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 } from "../sharedState.service";
import { toObservable } from "@angular/core/rxjs-interop";
import { FilteredFormArray } from "../filteredFormArray";
import { LimitValidator } from "../validation/limit";

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

```

```

editing: boolean = false;

keywordGroup = new FilteredFormArray([
  this.createKeywordFormControl()
]);

productForm: FormGroup = new FormGroup({
  name: new FormControl("", {
    validators: [
      Validators.required,
      Validators.minLength(3),
      Validators.pattern("[A-Za-z ]+$")
    ],
    updateOn: "change"
  }),
  category: new FormControl("", { validators: Validators.required }),
  price: new FormControl("", {
    validators: [
      Validators.required,
      Validators.pattern("[0-9\\.]+$"),
      LimitValidator.Limit(300)
    ]
  }),
  keywords: this.keywordGroup
});

// ...statements omitted for brevity...
}

```

To see the effect of the custom validator, enter a value in the Price field that exceeds the limit set in listing 22.14, as shown in figure 22.7.

Price

500

The price must be less than 300

Figure 22.7. Using a custom validator

22.3.1 Creating a directive for a custom validator

A directive is required to apply a custom validator to a template element when the reactive forms API isn't used. Add a file named `hiLOW.ts` to the `src/app/validation` folder with the content shown in listing 22.15.

Listing 22.15. The contents of the `hiLOW.ts` file in the `src/app/validation` folder

```

import { Directive, Input, SimpleChanges } from "@angular/core";
import { AbstractControl, NG_VALIDATORS, ValidationErrors,
  Validator, ValidatorFn } from "@angular/forms";

```

```

export class HiLowValidator {

  static HiLow(high:number, low: number) : ValidatorFn {
    return (control: AbstractControl) : ValidationErrors | null => {
      let val = parseFloat(control.value);
      if (isNaN(val) || val > high || val < low) {
        return {"hiLow": {
          "high": high, "low": low, "actualValue": val}};
      }
      return null;
    }
  }
}

@Directive({
  selector: 'input[high][low]',
  providers: [{provide: NG_VALIDATORS,
    useExisting: HiLowValidatorDirective,
    multi: true}]
})
export class HiLowValidatorDirective implements Validator {

  @Input()
  high: number | string | undefined

  @Input()
  low: number | string | undefined

  validator?: (control: AbstractControl) => ValidationErrors | null;

  ngOnChanges(changes: SimpleChanges): void {
    if ("high" in changes || "low" in changes) {
      let hival = typeof(this.high) == "string"
        ? parseInt(this.high) : this.high;
      let loval = typeof(this.low) == "string"
        ? parseInt(this.low) : this.low;
      this.validator = HiLowValidator.HiLow(hival
        ?? Number.MAX_VALUE, loval ?? 0);
    }
  }

  validate(control: AbstractControl): ValidationErrors | null {
    return this.validator?.(control) ?? null;
  }
}

```

The `HiLowValidator` class defines a `HiLow` factory function that can be used with reactive forms. This listing also defines the `HiLowValidatorDirective` class, which implements the `Validator` interface and the `validate` method it defines. The `ngOnChanges` method is used to create a new validator when the input value changes, which is used in the `validate` method to assess the contents of a control.

Validation directives use the `providers` property to register the validator as a service, like this:

```
...
```



```

providers: [{provide: NG_VALIDATORS,
             useExisting: HiLowValidatorDirective, multi: true}]
...

```

This configuration setting adds the directive to the set of validation directives applied by Angular when verifying a value. Without this configuration, Angular will not use the directive for validation. Listing 22.16 registers the validation directive so that it can be used in templates.

Listing 22.16. Registering a directive in the core.module.ts file in the src/app/core folder

```

import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule, NG_VALIDATORS, ReactiveFormsModule }
  from "@angular/forms";
import { ModelModule } from "../model/model.module";
import { TableComponent } from "../table.component";
import { FormComponent } from "../form.component";
import { SharedState } from "../sharedState.service";
import { ValidationHelper } from "../validationHelper.pipe";
import { ValidationErrorsDirective } from "../validationErrors.directive";
import { HiLowValidatorDirective } from "../validation/hilow";

@NgModule({
  imports: [BrowserModule, FormsModule, ModelModule,
    ReactiveFormsModule],
  declarations: [TableComponent, FormComponent, ValidationHelper,
    ValidationErrorsDirective, HiLowValidatorDirective],
  exports: [ModelModule, TableComponent, FormComponent],
  providers: [SharedState]
})
export class CoreModule { }

```

Listing 22.17 adds support for producing a validation message that can be displayed to the user.

Listing 22.17. Adding a validation message in the validationHelper.pipe.ts file in the src/app/core folder

```

import { Pipe } from "@angular/core";
import { FormControl, ValidationErrors } from "@angular/forms";

@Pipe({
  name: "validationFormat"
})
export class ValidationHelper {

  transform(source: any, name: any) : string[] {
    if (source instanceof FormControl) {
      return this.format((source as FormControl).errors, name)
    }
    return this.format(source as ValidationErrors, name)
  }

  format(errors: ValidationErrors | null, name: string): string[] {
    let messages: string[] = [];
    for (let errorName in errors) {

```

```

switch (errorName) {
  case "required":
    messages.push(`You must enter a ${name}`);
    break;
  case "minlength":
    messages.push(`A ${name} must be at least
      ${errors['minlength'].requiredLength}
      characters`);
    break;
  case "pattern":
    messages.push(`The ${name} contains
      illegal characters`);
    break;
  case "limit":
    messages.push(`The ${name} must be less than
      ${errors['limit'].limit}`);
    break;
  case "hilow":
    messages.push(`The ${name} must be between
      ${errors['hilow'].low} and
      ${errors['hilow'].high}`);
    break;
}
}
return messages;
}
}

```

Finally, listing 22.18 applies the validation directive in a template.

Listing 22.18. Applying a directive in the form.component.html file in the src/app/core folder

```

...
<div class="form-group">
  <label>Price</label>
  <input class="form-control" formControlName="price"
    [high]=300 [low]=10 />
  <ul class="text-danger list-unstyled mt-1">
    <li *validationErrors="productForm; control:'price'; let err">
      {{ err }}
    </li>
  </ul>
</div>
...

```

As you enter values into the Price field, the `HiLowValidatorDirective` uses the `HiLow` factory function for validation, as shown in figure 22.8.

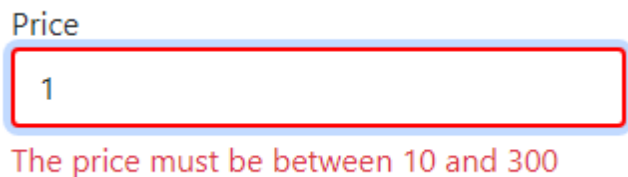


Figure 22.8. Applying a custom validator with a directive

22.3.2 Validating across multiple fields

Custom validators can be used to enforce policies that apply to multiple fields. This type of validator is applied to the `FormGroup` or `FormArray` that is the parent to the controls that are validated. Add a file named `unique.ts` to the `src/app/validation` folder with the contents shown in listing 22.19.

Listing 22.19. The contents of the `unique.ts` file in the `src/app/validation` folder

```
import { AbstractControl, FormArray, ValidationErrors, ValidatorFn }
  from "@angular/forms";

export class UniqueValidator {

  static unique() : ValidatorFn {
    return (control: AbstractControl) : ValidationErrors | null => {
      if (control instanceof FormArray) {
        let badElems = control.controls.filter((child, index) => {
          return control.controls.filter((c, i2) => i2 !== index)
            .some(target => target.value !== ""
              && target.value === child.value);
        });
        if (badElems.length > 0) {
          return {"unique": {}};
        }
      }
      return null;
    }
  }
}
```

The validator function looks for duplicate values in the array of controls managed by the `FormArray` and produces an error if there are duplicates. Listing 22.20 applies the validator to the `FormArray` in the component class.

Listing 22.20. Applying a validator in the `form.component.ts` file in the `src/app/core` folder

```
import { Component } from "@angular/core";
import { FormArray, FormControl, FormGroup, NgForm, 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 } from "../sharedState.service";
import { toObservable } from "@angular/core/rxjs-interop";
import { FilteredFormArray } from "../filteredFormArray";
import { LimitValidator } from "../validation/limit";
import { UniqueValidator } from "../validation/unique";

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

  keywordGroup = new FilteredFormArray([
    this.createKeywordFormControl()
  ], {
    validators: UniqueValidator.unique()
  });

  // ...statements omitted for brevity...
}

```

Listing 22.21 adds a validation message that can be presented to the user.

Listing 22.21. Adding a validation message in the validationHelper.pipe.ts file in the src/app/core folder

```

import { Pipe } from "@angular/core";
import { FormControl, ValidationErrors } from "@angular/forms";

@Pipe({
  name: "validationFormat"
})
export class ValidationHelper {

  transform(source: any, name: any) : string[] {
    if (source instanceof FormControl) {
      return this.format((source as FormControl).errors, name)
    }
    return this.format(source as ValidationErrors, name)
  }

  format(errors: ValidationErrors | null, name: string): string[] {
    let messages: string[] = [];
    for (let errorName in errors) {
      switch (errorName) {
        case "required":
          messages.push(`You must enter a ${name}`);
          break;
        case "minlength":
          messages.push(`A ${name} must be at least
            ${errors['minlength'].requiredLength}
            characters`);

```

```

        break;
    case "pattern":
        messages.push(`The ${name} contains
            illegal characters`);
        break;
    case "limit":
        messages.push(`The ${name} must be less than
            ${errors['limit'].limit}`);
        break;
    case "hilow":
        messages.push(`The ${name} must be between
            ${errors['hilow'].low} and
            ${errors['hilow'].high}`);
        break;
    case "unique":
        messages.push(`The ${name} must be unique`);
        break;
    }
}
return messages;
}
}

```

The final step is to display validation messages for the form array in the template, as shown in listing 22.22.

Listing 22.22. Displaying validation messages in the form.component.html file in the src/app/core folder

```

...
<div formGroupName="keywords">
  <button class="btn btn-sm btn-primary my-2"
    (click)="addKeywordControl()" type="button">
    Add Keyword
  </button>
  <ul class="text-danger list-unstyled mt-1">
    <li *validationErrors="productForm; control: 'keywords';
      label: 'keywords' let err">
      {{ err }}
    </li>
  </ul>
  <div class="form-group" *ngFor="let c of keywordGroup.controls;
    let i = index; let count = count">
    <label>Keyword {{ i + 1 }}</label>
    <div class="input-group">
      <input class="form-control"
        [formControlName]="i" [value]="c.value" />
      <button class="btn btn-danger" type="button"
        *ngIf="count > 1" (click)="removeKeywordControl(i)">
        Delete
      </button>
    </div>
    <ul class="text-danger list-unstyled mt-1">
      <li *validationErrors="productForm;
        control: 'keywords.' + i; label: 'keyword'; let err">
        {{ err }}
      </li>
    </ul>
  </div>

```

```

        </ul>
      </div>
    </div>
    ...

```

To see the effect, click the Edit button for the `Kayak` product and change the value of the second keyword field to `boat`. The validator will detect the duplicate value and display a validation message, as shown in figure 22.9.

Figure 22.9. Validating across fields

The validator works as expected, but there is an important mismatch between the validation message and the color coding applied to the individual elements, which I will improve upon in the next section.

IMPROVING CROSS-FIELD VALIDATION

Improving the cross-field validation experience can be done, but it requires careful navigation around the way that Angular expects groups of form controls to behave. Unlike an earlier example in this chapter, no internal methods are used, but the code relies on the `setTimeout` function to trigger changes after the current update cycle to perform updates without creating an infinite update loop.

The problem is that Angular expects changes to propagate up through the structure of form controls so that the user edits a field, which triggers validation in the `FormControl`, and then in its enclosing `FormGroup` or `FormArray`, working its way to the top-level `FormGroup`. To

achieve the effect I want, I have to push updates in the opposite direction so that a change in validation status in the `FormArray` triggers validation updates in the enclosed `FormControl` objects. Listing 22.23 updates the `unique` custom validator so that it alters the validation status of contained `FormControl` elements that contain the same value.

Listing 22.23. Improving validation in the `unique.ts` file in the `src/app/validation` folder

```
import { AbstractControl, FormArray, ValidationErrors, ValidatorFn }
  from "@angular/forms";

export class UniqueValidator {

  static uniquechild(control: AbstractControl)
    : ValidationErrors | null {
    return control.parent?.hasError("unique")
      ? {"unique-child": {}} : null;
  }

  static unique() : ValidatorFn {
    return (control: AbstractControl) : ValidationErrors | null => {
      let badElems: AbstractControl[] = [];
      let goodElems: AbstractControl[] = [];
      if (control instanceof FormArray) {
        control.controls.forEach((child, index) => {
          if (control.controls.filter((c, i2) => i2 !== index)
            .some(target => target.value !== ""
              && target.value === child.value)) {
            badElems.push(child);
          } else {
            goodElems.push(child);
          }
        })
        setTimeout(() => {
          badElems.forEach(c => {
            if (!c.hasValidator(this.uniquechild)) {
              c.markAsDirty();
              c.addValidators(this.uniquechild)
              c.updateValueAndValidity({onlySelf: true,
                emitEvent: false});
            }
          })
          goodElems.forEach(c => {
            if (c.hasValidator(this.uniquechild)) {
              c.removeValidators(this.uniquechild);
            }
            c.updateValueAndValidity({ onlySelf: true,
              emitEvent: false})
          })
        }, 0);
      }
      return badElems.length > 0 ? {"unique": {}} : null;
    }
  }
}
```

I add a validator to child controls that have duplicate values, which will ensure they are marked in red. The additional code in listing 22.23 takes care of adding and removing the validator and triggering validation updates when there are changes, which I do through the `updateValueAndValidity` method. This method, which I have described in table 22.4 for quick reference, updates a control's `value` property and performs validation. This method is defined by the `AbstractControl` class, which means that it can be used on `FormControl`, `FormGroup`, and `FormArray` objects.

Table 22.4. The `AbstractControl` method for manual updates

Name	Description
<code>updateValueAndValidity(opts)</code>	This method causes a control to update its value and perform validation. The optional argument can be used to restrict propagating the change up the form hierarchy by setting the <code>onlySelf</code> property to <code>true</code> and preventing events by setting the <code>emitEvent</code> property to <code>false</code> .

The changes in listing 22.23 ensure that individual controls with the form array are marked as invalid when they contain duplicate values, as shown in figure 22.10.

Add Keyword

The keywords must be unique

Keyword 1

boat

Delete

Keyword 2

boat

Delete

Save

Cancel

Figure 22.10. Improving the cross-field validation experience

22.3.3 Performing validation asynchronously

Asynchronous validation is useful for complex validation tasks or where the amount of time taken to perform validation is subject to delay, such as when a call to a remote HTTP service is required.

Add a file named `prohibited.ts` to the `src/app/validation` folder with the contents shown in listing 22.24.

Listing 22.24. The contents of the `prohibited.ts` file in the `src/app/validation` folder

```
import { AbstractControl, AsyncValidatorFn, ValidationErrors }
  from "@angular/forms";
import { Observable, Subject } from "rxjs";

export class ProhibitedValidator {

  static prohibitedTerms: string[] = ["ski", "swim"]

  static prohibited(): AsyncValidatorFn {
    return (control: AbstractControl): Promise<ValidationErrors | null>
      | Observable<ValidationErrors | null> => {
      let subject = new Subject<ValidationErrors | null>();
      setTimeout(() => {
        let match = false;
        this.prohibitedTerms.forEach(word => {
          if ((control.value as string)
            .toLowerCase().indexOf(word) > -1) {
            subject.next({"prohibited": { prohibited: word}})
            match = true;
          }
        });
        if (!match) {
          subject.next(null);
        }
        subject.complete();
      }, 1000);
      return subject;
    }
  }
}
```

Asynchronous validators produce their results through a `Promise`, which is useful when using non-Angular packages, or an `Observable`, which is useful when using the Angular features provided for making HTTP requests. For simplicity, the validator in listing 22.24 simulates an asynchronous operation using the `setTimeout` function and compares the value it receives with a list of prohibited terms.

When performing asynchronous validation, it is important to produce a `ValidationErrors` object or, if there are no errors, `null`, so that Angular knows that the validation process is complete. Listing 22.25 introduces a new validation message that can be displayed to the user.

Listing 22.25. Adding a message in the `validationHelper.pipe.ts` file in the `src/app/core` folder

```

...
format(errors: ValidationErrors | null, name: string): string[] {
  let messages: string[] = [];
  for (let errorName in errors) {
    switch (errorName) {
      case "required":
        messages.push(`You must enter a ${name}`);
        break;
      case "minlength":
        messages.push(`A ${name} must be at least
          ${errors['minlength'].requiredLength}
          characters`);
        break;
      case "pattern":
        messages.push(`The ${name} contains
          illegal characters`);
        break;
      case "limit":
        messages.push(`The ${name} must be less than
          ${errors['limit'].limit}`);
        break;
      case "hilow":
        messages.push(`The ${name} must be between
          ${errors['hilow'].low} and
          ${errors['hilow'].high}`);
        break;
      case "unique":
        messages.push(`The ${name} must be unique`);
        break;
      case "prohibited":
        messages.push(`The ${name} may not contain
          "${errors["prohibited"].prohibited}"`);
        break;
    }
  }
  return messages;
}
...

```

Listing 22.26 applies the asynchronous validator to a `FormControl`, which is done using the `asyncValidators` property.

Listing 22.26. Applying a validator in the `form.component.ts` file in the `src/app/core` folder

```

import { Component } from "@angular/core";
import { FormArray, FormControl, FormGroup, NgForm, 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 } from "../sharedState.service";
import { toObservable } from "@angular/core/rxjs-interop";
import { FilteredFormArray } from "../filteredFormArray";
import { LimitValidator } from "../validation/limit";
import { UniqueValidator } from "../validation/unique";

```

```

import { ProhibitedValidator } from "../validation/prohibited";

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

  keywordGroup = new FilteredFormArray([
    this.createKeywordFormControl()
  ], {
    validators: UniqueValidator.unique()
  });

  productForm: FormGroup = new FormGroup({
    name: new FormControl("", {
      validators: [
        Validators.required,
        Validators.minLength(3),
        Validators.pattern("^[A-Za-z ]+$")
      ],
      updateOn: "change"
    }),
    category: new FormControl("", {
      validators: Validators.required,
      asyncValidators: ProhibitedValidator.prohibited()
    }),
    price: new FormControl("", {
      validators: [
        Validators.required,
        Validators.pattern("^[0-9\\.]+$"),
        LimitValidator.Limit(300)
      ]
    }),
    keywords: this.keywordGroup
  });

  // ...statements omitted for brevity...
}

```

While waiting for an asynchronous validation result, Angular puts the `FormControl` object into the pending state, which adds the HTML element to the `ng-pending` class. Listing 22.27 defines a new CSS style that will be applied to pending elements.

Listing 22.27. Adding a style in the `form.component.css` file in the `src/app/core` folder

```

input.ng-dirty.ng-invalid { border: 2px solid #ff0000 }
input.ng-dirty.ng-valid { border: 2px solid #6bc502 }
input.ng-pending { border: 2px solid #ffc107 }

```

To see the asynchronous validator working, start typing into the Category field. The HTML element will be displayed with an amber border during validation, and an error will be displayed if the text you enter contains the terms *ski* or *swim*, as shown in figure 22.11.

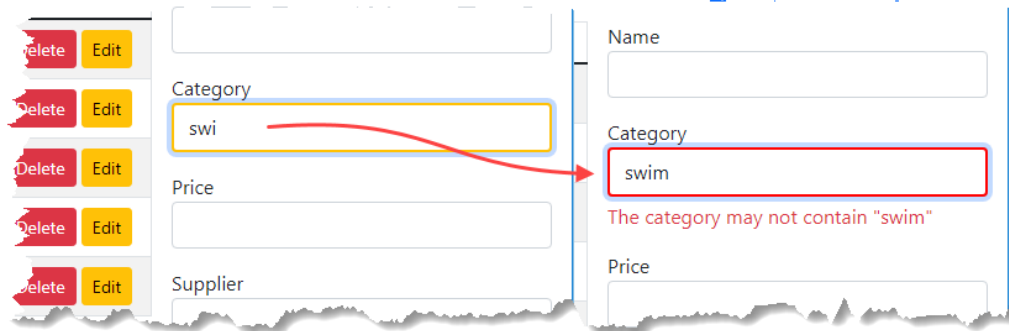


Figure 22.11. Using an asynchronous validator

There are two points of note when using an asynchronous validator. The first point is that asynchronous validation is performed only when no synchronous validator has returned an error, which can create a confusing sequence of validation messages unless the interaction between validators has been thought through.

The second point is that asynchronous validation is performed after every change to the form control (as long as there are no synchronous validation errors). This can result in a large number of validation operations, which may be slow or expensive to perform. If this is a concern, then you can change the update frequency using the `updateOn` option described in chapter 21.

22.4 Summary

In this chapter, I described the Angular forms API features for creating controls dynamically using a `FormArray` and explained the different ways in which custom validation can be performed, including the use of asynchronous validators.

- Angular provides an API for creating form components dynamically.
- The `FormGroup` class is used to represent dynamically created components.
- The API includes support for validating data, which can be done synchronously or asynchronously.

In the next chapter, I describe the features that Angular provides for making HTTP requests.

23

Making HTTP Requests

This chapter covers

- Using the built-in Angular features for HTTP requests
- Receiving data from HTTP requests
- Setting headers in HTTP requests
- Handling HTTP request errors

All the examples since chapter 9 have relied on static data that has been hardwired into the application. In this chapter, I demonstrate how to use asynchronous HTTP requests, often called Ajax requests, to interact with a web service to get real data into an application. Table 23.1 puts HTTP requests in context.

Table 23.1. Putting Asynchronous HTTP Requests in Context

Question	Answer
What are they?	Asynchronous HTTP requests are HTTP requests sent by the browser on behalf of the application. The term <i>asynchronous</i> refers to the fact that the application continues to operate while the browser is waiting for the server to respond.
Why are they useful?	Asynchronous HTTP requests allow Angular applications to interact with web services so that persistent data can be loaded into the application and changes can be sent to the server and saved.
How are they used?	Requests are made using the <code>HttpClient</code> class, which is delivered as a service through dependency injection. This class provides an Angular-friendly wrapper around the browser's <code>XMLHttpRequest</code> feature.

Are there any pitfalls or limitations?	Using the Angular HTTP feature requires the use of Reactive Extensions <code>Observable</code> objects.
Are there any alternatives?	You can work directly with the browser's <code>XMLHttpRequest</code> object if you prefer, and some applications—those that don't need to deal with persistent data—can be written without making HTTP requests at all.

Table 23.2 summarizes the chapter.

Table 23.2. Chapter Summary

Problem	Solution	Listing
Sending HTTP requests in an Angular application	Use the <code>Http</code> service	1–7
Performing REST operations	Use the HTTP method and URL to specify an operation and a target for that operation	8–10
Making cross-origin requests	Use the <code>HttpClient</code> service to support CORS automatically	11
Including headers in a request	Set the <code>headers</code> property in the <code>Request</code> object	12–13
Responding to an HTTP error	Create an error handler class	14, 15

23.1 Preparing the example project

This chapter uses the `exampleApp` project created in chapter 20 and modified in the chapters that followed. For this chapter, I rely on a server that responds to HTTP requests with JSON data. Run the command shown in listing 23.1 in the `exampleApp` folder to add the `json-server` package to the project.

TIP You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/manningbooks/pro-angular-16>. See chapter 1 for how to get help if you have problems running the examples.

Listing 23.1. Adding a package to the project

```
npm install json-server@0.17.3
```

I added an entry in the `scripts` section of the `package.json` file to run the `json-server` package, as shown in listing 23.2.

Listing 23.2. Adding a script entry in the `package.json` file in the `exampleApp` folder

```
...
"scripts": {
  "ng": "ng",
```

```

"start": "ng serve",
"build": "ng build",
"watch": "ng build --watch --configuration development",
"test": "ng test",
"json": "json-server --p 3500 restData.js"
},
...

```

23.1.1 Configuring the model feature module

The `@angular/common/http` JavaScript module contains an Angular module called `HttpClientModule`, which must be imported into the application in either the root module or one of the feature modules before HTTP requests can be created. In listing 23.3, I imported the module to the `model` module, which is the natural place in the example application because I will be using HTTP requests to populate the model with data.

Listing 23.3. Importing a module in the `model.module.ts` file in the `src/app/model` folder

```

import { NgModule } from "@angular/core";
import { StaticDataSource } from "../static.datasource";
import { Model } from "../repository.model";
import { HttpClientModule } from "@angular/common/http";

@NgModule({
  imports: [HttpClientModule],
  providers: [Model, StaticDataSource]
})
export class ModelModule { }

```

23.1.2 Creating the data file

To provide the `json-server` package with some data, I added a file called `restData.js` to the `exampleApp` folder and added the code shown in listing 23.4.

Listing 23.4. The contents of the `restData.js` file in the `exampleApp` folder

```

module.exports = function () {
  var data = {
    products: [
      { id: 1, name: "Kayak", category: "Watersports", price: 275,
        keywords: ["boat", "small"] },
      { id: 2, name: "Lifejacket", category: "Watersports",
        price: 48.95, keywords: ["safety"] },
      { id: 3, name: "Soccer Ball", category: "Soccer",
        price: 19.50 },
      { id: 4, name: "Corner Flags", category: "Soccer",
        price: 34.95 },
      { id: 5, name: "Stadium", category: "Soccer", price: 79500 },
      { id: 6, name: "Thinking Cap", category: "Chess", price: 16 },
      { id: 7, name: "Unsteady Chair", category: "Chess",
        price: 29.95 },
      { id: 8, name: "Human Chess Board", category: "Chess",
        price: 75 },
      { id: 9, name: "Bling Bling King", category: "Chess",
        price: 1200 }
    ]
  }
}

```

```

    }
    return data
  }
}

```

The `json-server` package can work with JSON or JavaScript files. If you use a JSON file, then its contents will be modified to reflect change requests made by clients. I have chosen the JavaScript option, which allows data to be generated programmatically and means that restarting the process will return to the original data.

23.1.3 Running the example project

Open a new command prompt, navigate to the `exampleApp` folder, and run the following command to start the data server:

```
npm run json
```

This command will start the `json-server`, which will listen for HTTP requests on port 3500. Open a new browser window and navigate to `http://localhost:3500/products/2`. The server will respond with the following data:

```

{
  "id": 2,
  "name": "Lifejacket",
  "category": "Watersports",
  "price": 48.95,
  "keywords": [
    "safety"
  ]
}

```

Leave the `json-server` running and use a separate command prompt to start the Angular development tools by running the following command in the `exampleApp` folder:

```
ng serve
```

Use the browser to navigate to `http://localhost:4200` to see the content illustrated in figure 23.1.

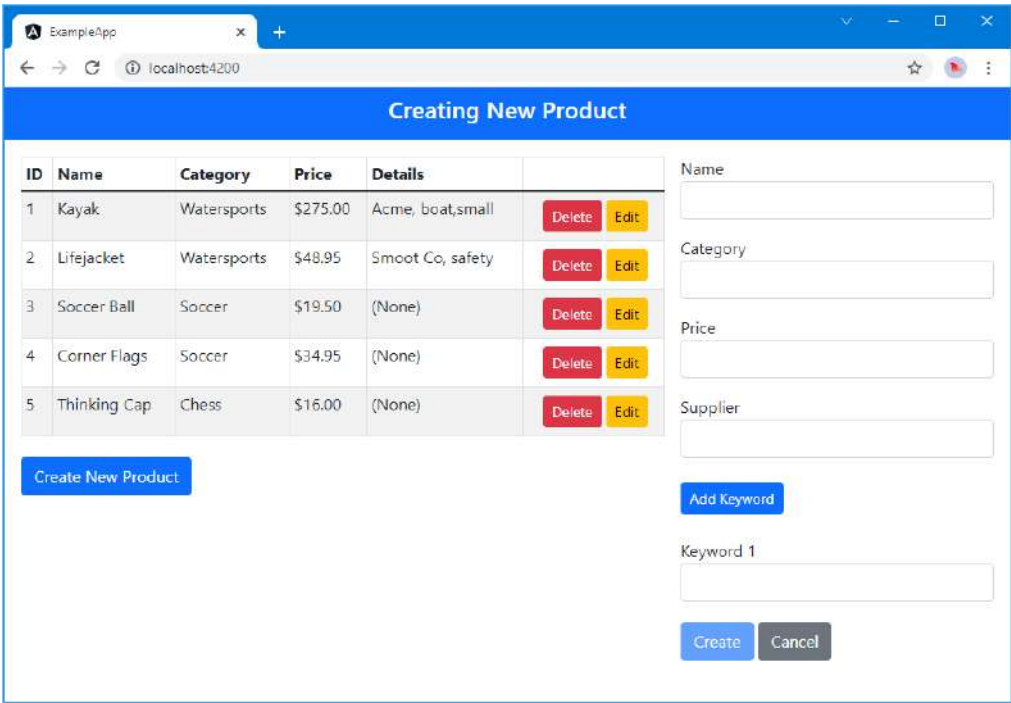


Figure 23.1. Running the example application

23.2 Understanding RESTful web services

The most common approach for delivering data to an application is to use the Representational State Transfer pattern, known as REST, to create a data web service. There is no detailed specification for REST, which leads to a lot of different approaches that fall under the RESTful banner. There are, however, some unifying ideas that are useful in web application development.

The core premise of a RESTful web service is to embrace the characteristics of HTTP so that request methods—also known as *verbs*—specify an operation for the server to perform, and the request URL specifies one or more data objects to which the operation will be applied.

As an example, here is a URL that might refer to a specific product in the example application:

```
http://localhost:3500/products/2
```

The first segment of the URL—`products`—is used to indicate the collection of objects that will be operated on and allows a single server to provide multiple services, each with separate data. The second segment—`2`—selects an individual object within the `products` collection. In the example, it is the value of the `id` property that uniquely identifies an object and that will be used in the URL, in this case, specifying the `Lifejacket` object.

The HTTP verb or method used to make the request tells the RESTful server what operation should be performed on the specified object. When you tested the RESTful server in the previous section, the browser sent an HTTP GET request, which the server interprets as an instruction to retrieve the specified object and send it to the client. It is for this reason that the browser displayed a JSON representation of the `Lifejacket` object.

Table 23.3 shows the most common combination of HTTP methods and URLs and explains what each of them does when they are sent to a RESTful server.

Table 23.3. Common HTTP verbs and their effect in a RESTful web service

Verb	URL	Description
GET	/products	This combination retrieves all the objects in the <code>products</code> collection.
GET	/products/2	This combination retrieves the object whose <code>id</code> is 2 from the <code>products</code> collection.
POST	/products	This combination is used to add a new object to the <code>products</code> collection. The request body contains a JSON representation of the new object.
PUT	/products/2	This combination is used to replace the object in the <code>products</code> collection whose <code>id</code> is 2. The request body contains a JSON representation of the replacement object.
PATCH	/products/2	This combination is used to update a subset of the properties of the object in the <code>products</code> collection whose <code>id</code> is 2. The request body contains a JSON representation of the properties to update and the new values.
DELETE	/products/2	This combination is used to delete the product whose <code>id</code> is 2 from the <code>products</code> collection.

Caution is required because there can be considerable differences in the way that some RESTful web services work, caused by differences in the frameworks used to create them and the preferences of the development team. It is important to confirm how a web service uses verbs and what is required in the URL and request body to perform operations.

Some common variations include web services that won't accept any request bodies that contain `id` values (to ensure they are generated uniquely by the server's data store) or any web services that don't support all of the verbs (it is common to ignore `PATCH` requests and only accept updates using the `PUT` verb).

23.3 Replacing the static data source

The best place to start with HTTP requests is to replace the static data source in the example application with one that retrieves data from the RESTful web service. This will provide a

foundation for describing how Angular supports HTTP requests and how they can be integrated into an application.

23.3.1 Creating the new data source service

To create a new data source, I added a file called `rest.datasource.ts` in the `src/app/model` folder and added the statements shown in listing 23.5.

Listing 23.5. The contents of the `rest.datasource.ts` file in the `src/app/model` folder

```
import { Injectable, Signal } from "@angular/core";
import { HttpClient } from "@angular/common/http";
import { Product } from "../product.model";
import { Observable } from "rxjs";

export const REST_URL = `http://${location.hostname}:3500/products`;

@Injectable()
export class RestDataSource {

  constructor(private http: HttpClient) { }

  getData(): Observable<Product[]> {
    return this.http.get<Product[]>(REST_URL);
  }
}
```

This is a simple-looking class, but there are some important features at work, which I described in the sections that follow.

SETTING UP THE HTTP REQUEST

Angular provides the ability to make asynchronous HTTP requests through the `HttpClient` class, which is defined in the `@angular/common/http` JavaScript module and is provided as a service in the `HttpClientModule` feature module. The data source declared a dependency on the `HttpClient` class using its constructor, like this:

```
...
constructor(private http: HttpClient) { }
...
```

The `HttpClient` object received through the constructor is used to make an HTTP GET request in the data source's `getData` method, like this:

```
...
getData(): Observable<Product[]> {
  return this.http.get<Product[]>(REST_URL);
}
...
```

The `HttpClient` class defines a set of methods for making HTTP requests, each of which uses a different HTTP verb, as described in table 23.4.

TIP The methods in table 23.4 accept an optional configuration object, as demonstrated in the “Configuring request headers” section.

Table 23.4. The `HttpClient` methods

Name	Description
<code>get(url)</code>	This method sends a GET request to the specified URL.
<code>post(url, body)</code>	This method sends a POST request using the specified object as the body.
<code>put(url, body)</code>	This method sends a PUT request using the specified object as the body.
<code>patch(url, body)</code>	This method sends a PATCH request using the specified object as the body.
<code>delete(url)</code>	This method sends a DELETE request to the specified URL.
<code>head(url)</code>	This method sends a HEAD request, which has the same effect as a GET request except that the server will return only the headers and not the request body.
<code>options(url)</code>	This method sends an OPTIONS request to the specified URL.
<code>request(method, url, options)</code>	This method can be used to send a request with any verb, as described in the “Consolidating HTTP Requests” section.

PROCESSING THE RESPONSE

The methods described in table 23.4 accept a type parameter, which the `HttpClient` class uses to parse the response received from the server. The RESTful web server returns JSON data, which has become the de facto standard used by web services, and the `HttpClient` object will automatically convert the response into an `Observable` that yields an instance of the type parameter when it completes. This means that if you call the `get` method, for example, with a `Product[]` type parameter, then the response from the `get` method will be an `Observable<Product[]>` that represents the eventual response from the HTTP request.

```
...
getData(): Observable<Product[]> {
    return this.http.get<Product[]>(REST_URL);
}
...
```

CAUTION The methods in table 23.4 prepare an HTTP request, but it isn’t sent to the server until the `Observable` object’s `subscribe` method is invoked. Be careful, though, because the request will be sent once per call to the `subscribe` method, which makes it easy to inadvertently send the same request multiple times.

23.3.2 Configuring the data source

The next step is to configure a provider for the new data source and to create a value-based provider to configure it with a URL to which requests will be sent. Listing 23.6 shows the changes to the `model.module.ts` file.

Listing 23.6. Configuring the data source in the `model.module.ts` file in the `src/app/model` folder

```
import { NgModule } from "@angular/core";
import { StaticDataSource } from "../static.datasource";
import { Model } from "../repository.model";
import { HttpClientModule } from "@angular/common/http";
import { RestDataSource } from "../rest.datasource";

@NgModule({
  imports: [HttpClientModule],
  providers: [Model, RestDataSource]
})
export class ModelModule { }
```

The `HttpClient` service is defined in the `HttpClientModule` module. The new provider enables the `RestDataSource` class as a service. I removed the provider for the `StaticDataSource` class, which is no longer required.

23.3.3 Using the REST data source

The final step is to update the repository class so that it declares a dependency on the new data source and uses it to get the application data, as shown in listing 23.7.

Listing 23.7. Using the new data source in the `repository.model.ts` file in the `src/app/model` folder

```
import { Injectable, Signal, computed, signal } from "@angular/core";
import { Product } from "../product.model";
//import { StaticDataSource } from "../static.datasource";
import { RestDataSource } from "../rest.datasource";

@Injectable()
export class Model {
  private products = signal<Product[]>([]);
  private locator = (p: Product, id?: number) => p.id == id;

  constructor(private dataSource: RestDataSource) {
    //this.products.set(this.dataSource.getData());
    this.dataSource.getData()
      .subscribe(data => this.products.set(data));
  }

  // ...methods omitted for brevity...
}
```

The constructor dependency has changed so that the repository will receive a `RestDataSource` object when it is created. Within the constructor, the data source's `getData` method is called, and the `subscribe` method is used to receive the data objects that are returned from the server and use them to update the `products` signal.

When you save the changes, the browser will reload the application, and the new data source will be used. An asynchronous HTTP request will be sent to the RESTful web service, which will return the larger set of data objects shown in figure 23.2.

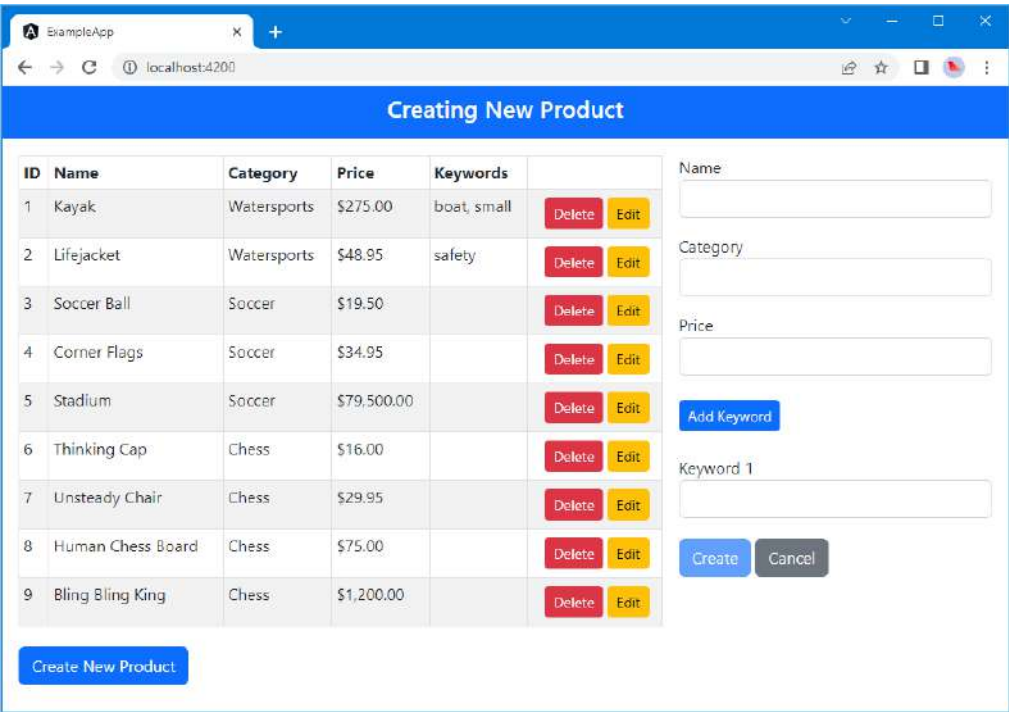


Figure 23.2. Getting the application data

23.3.4 Saving and deleting data

The data source can get data from the server, but it also needs to send data the other way, persisting changes that the user makes to objects in the model and storing new objects that are created. Listing 23.8 adds methods to the data source class to send HTTP requests to save or update objects using the Angular `HttpClient` class.

Listing 23.8. Sending data in the `rest.datasource.ts` file in the `src/app/model` folder

```
import { Injectable, Signal } from "@angular/core";
import { HttpClient } from "@angular/common/http";
import { Product } from "../product.model";
import { Observable } from "rxjs";

export const REST_URL = `http://${location.hostname}:3500/products`;

@Injectable()
export class RestDataSource {

  constructor(private http: HttpClient) { }

  getData(): Observable<Product[]> {
    return this.http.get<Product[]>(REST_URL);
  }
}
```

```

    }

    saveProduct(product: Product): Observable<Product> {
        return this.http.post<Product>(REST_URL, product);
    }

    updateProduct(product: Product): Observable<Product> {
        return this.http
            .put<Product>(`${REST_URL}/${product.id}`, product);
    }

    deleteProduct(id: number): Observable<Product> {
        return this.http.delete<Product>(`${REST_URL}/${id}`);
    }
}

```

The `saveProduct`, `updateProduct`, and `deleteProduct` methods follow the same pattern: they call one of the `HttpClient` class methods and return an `Observable<Product>` as the result.

When saving a new object, the ID of the object is generated by the server so that it is unique and clients don't inadvertently use the same ID for different objects. In this situation, the POST method is used, and the request is sent to the `/products` URL. When updating or deleting an existing object, the ID is already known, and a PUT request is sent to a URL that includes the ID. So, a request to update the object whose ID is 2, for example, is sent to the `/products/2` URL. Similarly, to remove that object, a DELETE request would be sent to the same URL.

What these methods have in common is that the server is the authoritative data store, and the response from the server contains the official version of the object that has been saved by the server. It is this object that is returned as the result of these methods, provided through the `Observable<Product>`.

Listing 23.9 shows the corresponding changes in the repository class that take advantage of the new data source features.

Listing 23.9. Using the data source features in the `repository.model.ts` file in the `src/app/model` folder

```

import { Injectable, Signal, computed, signal } from "@angular/core";
import { Product } from "../product.model";
import { RestDataSource } from "../rest.datasource";

@Injectable()
export class Model {
    private products = signal<Product[]>([]);
    private locator = (p: Product, id?: number) => p.id == id;

    constructor(private dataSource: RestDataSource) {
        this.dataSource.getData()
            .subscribe(data => this.products.set(data));
    }

    get Products(): Signal<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 == undefined) {
            this.dataSource.saveProduct(product)
                .subscribe(p =>
                    this.products.mutate(prods => prods.push(p)));
        } else {
            this.dataSource.updateProduct(product).subscribe(() => {
                this.products.mutate(prods => {
                    let index = prods.findIndex(p =>
                        this.locator(p, product.id));
                    prods.splice(index, 1, product);
                });
            });
        }
    }

    deleteProduct(id: number) {
        this.dataSource.deleteProduct(id).subscribe(() => {
            this.products.mutate(prods => {
                let index = prods.findIndex(p => this.locator(p, id));
                if (index > -1) {
                    prods.splice(index, 1);
                }
            });
        });
    }

    // private generateID(): number {
    //     let candidate = 100;
    //     while (this.getProduct(candidate) != null) {
    //         candidate++;
    //     }
    //     return candidate;
    // }
}

```

The changes use the data source to send updates to the server and use the results to update the signal so that the modified data is displayed by the rest of the application. To test the changes, click the Edit button for the Kayak product and change its name to Green Kayak. Click the Save button, and the browser will send an HTTP PUT request to the server, which will return a modified object that is added to the repository's `products` array and is displayed in the table, as shown in figure 23.3.

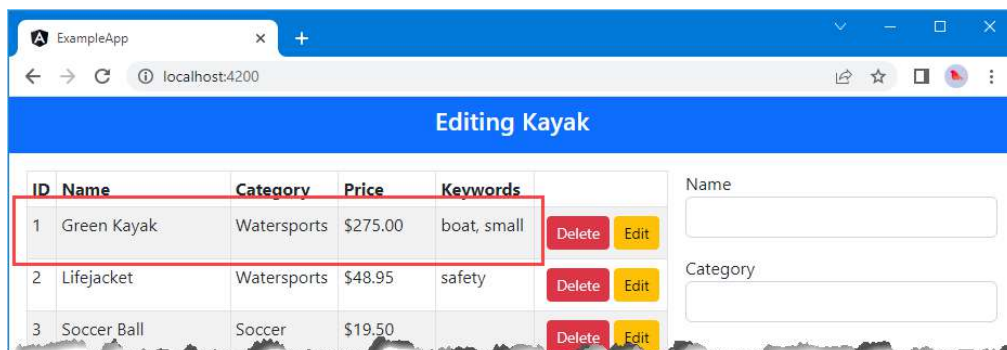


Figure 23.3. Sending a PUT request to the server

You can check that the server has stored the changes by using the browser to request `http://localhost:3500/products/1`, which will produce the following representation of the object:

```
{
  "id": 1,
  "name": "Green Kayak",
  "category": "Watersports",
  "price": 275,
  "keywords": [
    "boat",
    "small"
  ]
}
```

23.4 Consolidating HTTP requests

Each of the methods in the data source class duplicates the same basic pattern of sending an HTTP request using a verb-specific `HttpClient` method. This means that any change to the way that HTTP requests are made has to be repeated in four different places, ensuring that the requests that use the GET, POST, PUT, and DELETE verbs are all correctly updated and performed consistently.

The `HttpClient` class also defines the `request` method, which allows the HTTP verb to be specified as an argument. Listing 23.10 uses the `request` method to consolidate the HTTP requests in the data source class.

Listing 23.10. Consolidating HTTP requests in the `rest.datasource.ts` file in the `src/app/model` folder

```
import { Injectable, Signal } from "@angular/core";
import { HttpClient } from "@angular/common/http";
import { Product } from "../product.model";
import { Observable } from "rxjs";

export const REST_URL = `http://${location.hostname}:3500/products`;
```

```

@Inject()
export class RestDataSource {

  constructor(private http: HttpClient) { }

  getData(): Observable<Product[]> {
    return this.sendRequest<Product[]>("GET", REST_URL);
  }

  saveProduct(product: Product): Observable<Product> {
    return this.sendRequest<Product>("POST", REST_URL, product);
  }

  updateProduct(product: Product): Observable<Product> {
    return this.sendRequest<Product>("PUT",
      `${REST_URL}/${product.id}`, product);
  }

  deleteProduct(id: number): Observable<Product> {
    return this.sendRequest<Product>("DELETE", `${REST_URL}/${id}`);
  }

  private sendRequest<T>(verb: string, url: string, body?: Product)
    : Observable<T> {
    return this.http.request<T>(verb, url, {
      body: body
    });
  }
}

```

The `request` method accepts the HTTP verb, the URL for the request, and an optional object that is used to configure the request. The configuration object is used to set the request body using the `body` property, and the `HttpClient` will automatically take care of encoding the body object and including a serialized representation of it in the request.

Table 23.5 describes the most useful properties that can be specified to configure an HTTP request made using the `request` method.

Table 23.5. Useful request method configuration object properties

Name	Description
<code>headers</code>	This property returns an <code>HttpHeaders</code> object that allows the request headers to be specified, as described in the “Configuring Request Headers” section.
<code>body</code>	This property is used to set the request body. The object assigned to this property will be serialized as JSON when the request is sent.
<code>withCredentials</code>	When <code>true</code> , this property is used to include authentication cookies when making cross-site requests. This setting must be used only with servers that include the <code>Access-Control-Allow-Credentials</code> header in

	responses, as part of the Cross-Origin Resource Sharing (CORS) specification. See the “Making Cross-Origin Requests” section for details.
<code>responseType</code>	This property is used to specify the type of response expected from the server. The default value is <code>json</code> , indicating the JSON data format.

23.5 Making cross-origin requests

By default, browsers enforce a security policy that allows JavaScript code to make asynchronous HTTP requests only within the same *origin* as the document that contains them. This policy is intended to reduce the risk of cross-site scripting (CSS) attacks, where the browser is tricked into executing malicious code. The details of this attack are beyond the scope of this book, but the article available at http://en.wikipedia.org/wiki/Cross-site_scripting provides a good introduction to the topic.

For Angular developers, the same-origin policy can be a problem when using web services because they are typically outside of the origin that contains the application’s JavaScript code. Two URLs are considered to be in the same origin if they have the same protocol, host, and port and have different origins if this is not the case. The URL for the HTML file that contains the example application’s JavaScript code is `http://localhost:4200/index.html`. Table 23.6 summarizes how similar URLs have the same or different origins, compared with the application’s URL.

Table 23.6. URLs and their origins

URL	Origin Comparison
<code>http://localhost:4200/otherfile.html</code>	Same origin
<code>http://localhost:4200/app/main.js</code>	Same origin
<code>https://localhost:4200/index.html</code>	Different origin; protocol differs
<code>http://localhost:3500/products</code>	Different origin; port differs
<code>http://angular.io/index.html</code>	Different origin; host differs

As the table shows, the URL for the RESTful web service, `http://localhost:3500/products`, has a different origin because it uses a different port from the main application.

HTTP requests made using the Angular `HttpClient` class will automatically use Cross-Origin Resource Sharing to send requests to different origins. With CORS, the browser includes headers in the asynchronous HTTP request that provide the server with the origin of the JavaScript code. The response from the server includes headers that tell the browser whether it is willing to accept the request. The details of CORS are outside the scope of this book, but there is a good introduction to the topic at https://en.wikipedia.org/wiki/Cross-origin_resource_sharing.

For the Angular developer, CORS is something that is taken care of automatically, just as long as the server that receives asynchronous HTTP requests supports the specification. The

`json-server` package that has been providing the RESTful web service for the examples supports CORS and will accept requests from any origin, which is why the examples have been working. If you want to see CORS in action, use the browser's F12 developer tools to watch the network requests that are made when you edit or create a product. You may see a request made using the `OPTIONS` verb, known as the *preflight request*, which the browser uses to check that it is allowed to make the POST or PUT request to the server. This request and the subsequent request that sends the data to the server will contain an `Origin` header, and the response will contain one or more `Access-Control-Allow` headers, through which the server sets out what it is willing to accept from the client.

All of this happens automatically, and the only configuration option is the `withCredentials` property that was described in table 23.5. When this property is `true`, the browser will include authentication cookies, and headers from the origin will be included in the request to the server.

23.6 Configuring request headers

If you are using a commercial RESTful web service, you will often have to set a request header to provide an API key so that the server can associate the request with your application for access control and billing. You can set this kind of header—or any other header—by configuring the configuration object that is passed to the `request` method, as shown in listing 23.11.

Listing 23.11. Setting a header in the `rest.datasource.ts` file in the `src/app/model` folder

```
import { Injectable, Signal } from "@angular/core";
import { HttpClient, HttpHeaders } from "@angular/common/http";
import { Product } from "../product.model";
import { Observable } from "rxjs";

export const REST_URL = `http://${location.hostname}:3500/products`;

@Injectable()
export class RestDataSource {

  constructor(private http: HttpClient) { }

  getData(): Observable<Product[]> {
    return this.sendRequest<Product[]>("GET", REST_URL);
  }

  saveProduct(product: Product): Observable<Product> {
    return this.sendRequest<Product>("POST", REST_URL, product);
  }

  updateProduct(product: Product): Observable<Product> {
    return this.sendRequest<Product>("PUT",
      `${REST_URL}/${product.id}`, product);
  }

  deleteProduct(id: number): Observable<Product> {
    return this.sendRequest<Product>("DELETE", `${REST_URL}/${id}`);
  }
}
```

```

    }

    private sendRequest<T>(verb: string, url: string, body?: Product)
      : Observable<T> {
      return this.http.request<T>(verb, url, {
        body: body,
        headers: new HttpHeaders({
          "Access-Key": "<secret>",
          "Application-Name": "exampleApp"
        })
      });
    }
  }
}

```

The `headers` property is set to an `HttpHeaders` object, which can be created using a map object of properties that correspond to header names and the values that should be used for them. If you use the browser's F12 developer tools to inspect the asynchronous HTTP requests, you will see that the two headers specified in the listing are sent to the server along with the standard headers that the browser creates, like this:

```

...
Accept: */*
Accept-Encoding: gzip, deflate, sdch, br
Accept-Language: en-US,en;q=0.8
access-key: <secret>
application-name: exampleApp
Connection: keep-alive
...

```

If you have more complex demands for request headers, then you can use the methods defined by the `HttpHeaders` class, as described in table 23.7.

Table 23.7. The `HttpHeaders` methods

Name	Description
<code>keys()</code>	Returns all the header names in the collection
<code>get(name)</code>	Returns the first value for the specified header
<code>getAll(name)</code>	Returns all the values for the specified header
<code>has(name)</code>	Returns <code>true</code> if the collection contains the specified header
<code>set(header, value)</code>	Returns a new <code>HttpHeaders</code> object that replaces all existing values for the specified header with a single value
<code>set(header, values)</code>	Returns a new <code>HttpHeaders</code> object that replaces all existing values for the specified header with an array of values
<code>append(name, value)</code>	Appends a value to the list of values for the specified header
<code>delete(name)</code>	Removes the specified header from the collection

23.7 Handling errors

At the moment, there is no error handling in the application, which means that Angular doesn't know what to do if there is a problem with an HTTP request. To make it easy to generate an error, I have added a button to the product table that will lead to an HTTP request to delete an object that doesn't exist at the server, as shown in listing 23.12.

Listing 23.12. Adding a button in the `table.component.html` file in the `src/app/core` folder

```
<table class="table table-sm table-bordered table-striped">
  <thead>
    <tr>
      <th>ID</th><th>Name</th><th>Category</th>
      <th>Price</th><th>Keywords</th><th></th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let item of Products()">
      <td>{{item.id}}</td>
      <td>{{item.name}}</td>
      <td>{{item.category}}</td>
      <td>{{item.price | currency:"USD" }}</td>
      <td>{{ item.keywords?.join(", ")}}</td>
      <td class="text-center">
        <button class="btn btn-danger btn-sm m-1"
          (click)="deleteProduct(item.id)">
          Delete
        </button>
        <button class="btn btn-warning btn-sm"
          (click)="editProduct(item.id)">
          Edit
        </button>
      </td>
    </tr>
  </tbody>
</table>
<button class="btn btn-primary mt-1" (click)="createProduct()">
  Create New Product
</button>
<button class="btn btn-danger mt-1 mx-1" (click)="deleteProduct(-1)">
  Generate HTTP Error
</button>
```

The `button` element invokes the component's `deleteProduct` method with an argument of `-1`. The component will ask the repository to delete this object, which will lead to an HTTP DELETE request being sent to `/products/-1`, which does not exist. If you open the browser's JavaScript console and click the Generate HTTP Error button, you will see the response from the server displayed, like this:

```
DELETE http://localhost:3500/products/-1 404 (Not Found)
```

Improving this situation means detecting this kind of error when it occurs and notifying the user, who won't typically be looking at the JavaScript console. A real application might also respond to errors by logging them so they can be analyzed later, but I am going to keep things simple and just display an error message.

23.7.1 Generating user-ready messages

The first step in handling errors is to convert the HTTP exception into something that can be displayed to the user. The default error message, which is the one written to the JavaScript console, contains too much information to display to the user. Users don't need to know the URL that the request was sent to; just having a sense of the kind of problem that has occurred will be enough.

The best way to transform error messages is to use the `catchError` method. The `catchError` method is used with the `pipe` method to receive any errors that occur within an `Observable` sequence, as shown in listing 23.13.

Listing 23.13. Transforming errors in the `rest.datasource.ts` file in the `src/app/model` folder

```
import { Injectable, Signal } from "@angular/core";
import { HttpClient, HttpHeaders } from "@angular/common/http";
import { Product } from "../product.model";
import { Observable, catchError } from "rxjs";

export const REST_URL = `http://${location.hostname}:3500/products`;

@Injectable()
export class RestDataSource {

  constructor(private http: HttpClient) { }

  getData(): Observable<Product[]> {
    return this.sendRequest<Product[]>("GET", REST_URL);
  }

  saveProduct(product: Product): Observable<Product> {
    return this.sendRequest<Product>("POST", REST_URL, product);
  }

  updateProduct(product: Product): Observable<Product> {
    return this.sendRequest<Product>("PUT",
      `${REST_URL}/${product.id}`, product);
  }

  deleteProduct(id: number): Observable<Product> {
    return this.sendRequest<Product>("DELETE", `${REST_URL}/${id}`);
  }

  private sendRequest<T>(verb: string, url: string, body?: Product)
    : Observable<T> {
    return this.http.request<T>(verb, url, {
      body: body,
      headers: new HttpHeaders({
        "Access-Key": "<secret>",
        "Application-Name": "exampleApp"
      })
    })
    .pipe(catchError((error: Response) => {
      throw `Network Error: ${error.statusText} (${error.status})`
    }));
  }
}
```

```

    }));
  }
}

```

The function passed to the `catchError` method is invoked when there is an error and receives the `Response` object that describes the outcome, which in this case is used to generate an error message that contains the HTTP status code and status text from the response.

If you save the changes and then click the Generate HTTP Error button again, the error message will still be written to the browser's JavaScript console but will have changed to the format produced by the `catchError` method.

```
EXCEPTION: Network Error: Not Found (404)
```

23.7.2 Handling the errors

The errors have been transformed but not handled, which is why they are still being reported as exceptions in the browser's JavaScript console. There are two ways in which the errors can be handled. The first is to provide an error-handling function to the `subscribe` method for the `Observable` objects created by the `HttpClient` object. This is a useful way to localize the error and provide the repository with the opportunity to retry the operation or try to recover in some other way.

The second approach is to replace the built-in Angular error-handling feature, which responds to any unhandled errors in the application and, by default, writes them to the console. It is this feature that writes out the messages shown in the previous sections.

For the example application, I want to override the default error handler with one that uses the message service. I created a file called `errorHandler.ts` in the `src/app/messages` folder and used it to define the class shown in listing 23.14.

Listing 23.14. The contents of the `errorHandler.ts` file in the `src/app/messages` folder

```

import { ErrorHandler, Injectable, NgZone } from "@angular/core";
import { MessageService } from "../message.service";
import { Message } from "../message.model";

@Injectable()
export class MessageErrorHandler implements ErrorHandler {

  constructor(private messageService: MessageService,
              private ngZone: NgZone) {

  }

  handleError(error: any) {
    let msg = error instanceof Error
      ? error.message : error.toString();
    this.ngZone.run(() => this.messageService
      .reportMessage(new Message(msg, true)), 0);
  }
}

```

The `ErrorHandler` class is defined in the `@angular/core` module and responds to errors through a `handleError` method. The class shown in the listing replaces the default implementation of this method with one that uses the `MessageService` to report an error.

Redefining the error handler presents a problem. I want to display a message to the user, which requires the Angular change detection process to be triggered. But the message is produced by a service, and Angular doesn't keep track of the state of services as it does for components and directives. To resolve this issue, I defined an `NgZone` constructor parameter and used its `run` method to create the error message:

```
...
    this.ngZone.run(() => this.messageService
        .reportMessage(new Message(msg, true)), 0);
...
```

The `run` method executes the function it receives and then triggers the Angular change detection process. For this example, the result is that the new message will be displayed to the user. Without the use of the `NgZone` object, the error message would be created but would not be displayed to the user until the next time the Angular detection process runs.

To replace the default `ErrorHandler`, I used a class provider in the message module, as shown in listing 23.15. The service configuration replaces the existing implementation of the `ErrorHandler` service with the custom `MessageErrorHandler` class.

Listing 23.15. Configuring an error handler in the `message.module.ts` file in the `src/app/messages` folder

```
import { NgModule, ErrorHandler } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { MessageComponent } from "../message.component";
import { MessageService } from "../message.service";
import { MessageErrorHandler } from "../errorHandler";

@NgModule({
  imports: [BrowserModule],
  declarations: [MessageComponent],
  exports: [MessageComponent],
  providers: [MessageService,
    { provide: ErrorHandler, useClass: MessageErrorHandler }],
})
export class MessageModule { }
```

The error handling function uses the `MessageService` to report an error message to the user. Once these changes have been saved, clicking the Generate HTTP Error button produces an error that the user can see, as shown in figure 23.4.

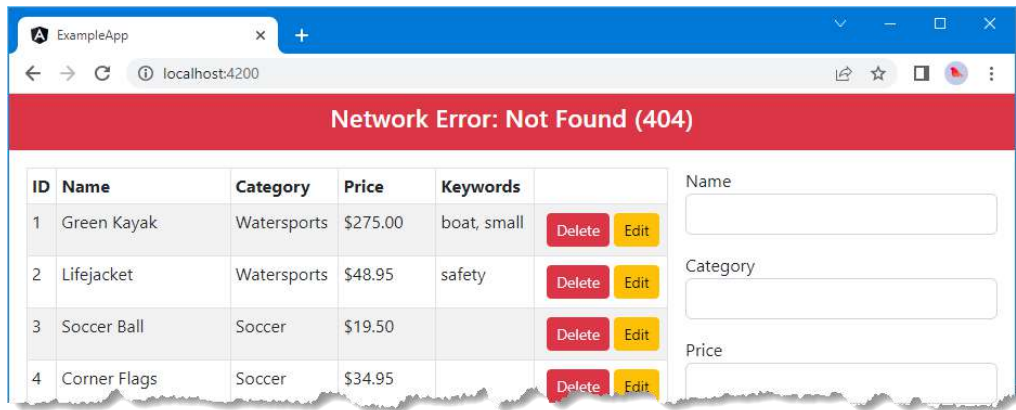


Figure 23.4. Handling an HTTP error

23.8 Summary

In this chapter, I explained how to make asynchronous HTTP requests in Angular applications. I introduced RESTful web services and the methods provided by the Angular `HttpClient` class that can be used to interact with them. I explained how the browser restricts requests to different origins and how Angular supports CORS.

- The `HttpClient` class makes HTTP requests and produces results using observables.
- The `HttpClient` class provides methods for commonly-used HTTP methods and general-purpose methods for sending any request.
- Cross-origin requests are performed automatically but require corresponding server-side support.
- The standard Angular error handling won't display HTTP errors to the user and required additional integration into applications.

In the next chapter, I introduce the URL routing feature, which allows for navigating complex applications.

24

Routing and navigation: part 1

This chapter covers

- Using the Angular routing system to select components based on the current URL
- Defining and using routes
- Using directives to enable navigation to different URLs
- Inspecting the active route within components
- Receiving events that describe route changes

The Angular routing feature allows applications to change the components and templates that are displayed to the user by responding to changes to the browser's URL. This allows complex applications to be created that adapt the content they present openly and flexibly, with minimal coding. To support this feature, data bindings and services can be used to change the browser's URL, allowing the user to navigate around the application.

Routing is useful as the complexity of a project increases because it allows the structure of an application to be defined separately from the components and directives, meaning that changes to the structure can be made in the routing configuration and do not have to be applied to the individual components.

In this chapter, I demonstrate how the basic routing system works and apply it to the example application. In chapters 25 and 26, I explain the more advanced routing features. Table 24.1 puts routing in context.

Table 24.1. Putting routing and navigation in context

Question	Answer
What is it?	Routing uses the browser's URL to manage the content displayed to the user.

Why is it useful?	Routing allows the structure of an application to be kept apart from the components and templates in the application. Changes to the structure of the application are made in the routing configuration rather than in individual components and directives.
How is it used?	The routing configuration is defined as a set of fragments that are used to match the browser's URL and to select a component whose template is displayed as the content of an HTML element called <code>router-outlet</code> .
Are there any pitfalls or limitations?	The routing configuration can become unmanageable, especially if the URL schema is being defined gradually on an ad hoc basis.
Are there any alternatives?	You don't have to use the routing feature. You could achieve similar results by creating a component whose view selects the content to display to the user with the <code>ngIf</code> or <code>ngSwitch</code> directive, although this approach becomes more difficult than using routing as the size and complexity of an application increases.

Table 24.2 summarizes the chapter.

Table 24.2. Chapter summary

Problem	Solution	Listing
Using URL navigation to select the content shown to users	Use URL routing	1–4
Navigating using an HTML element	Apply the <code>routerLink</code> attribute	5–7
Responding to route changes	Use the routing services to receive notifications	8
Including information in URLs	Use route parameters	9–17
Navigating using code	Use the <code>Router</code> service	18
Receiving notifications of routing activity	Handle routing events	19–23

24.1 Preparing the example project

This chapter uses the `exampleApp` project created in chapter 23. No changes are required for this chapter.

TIP You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/manningbooks/pro-angular-16>. See chapter 1 for how to get help if you have problems running the examples.

Open a new command prompt, navigate to the `exampleApp` folder, and run the following command to start the server that provides the RESTful web server:

```
npm run json
```

Open a separate command prompt, navigate to the `exampleApp` folder, and run the following command to start the Angular development tools:

```
ng serve
```

Open a new browser window and navigate to `http://localhost:4200` to see the content shown in figure 24.1.

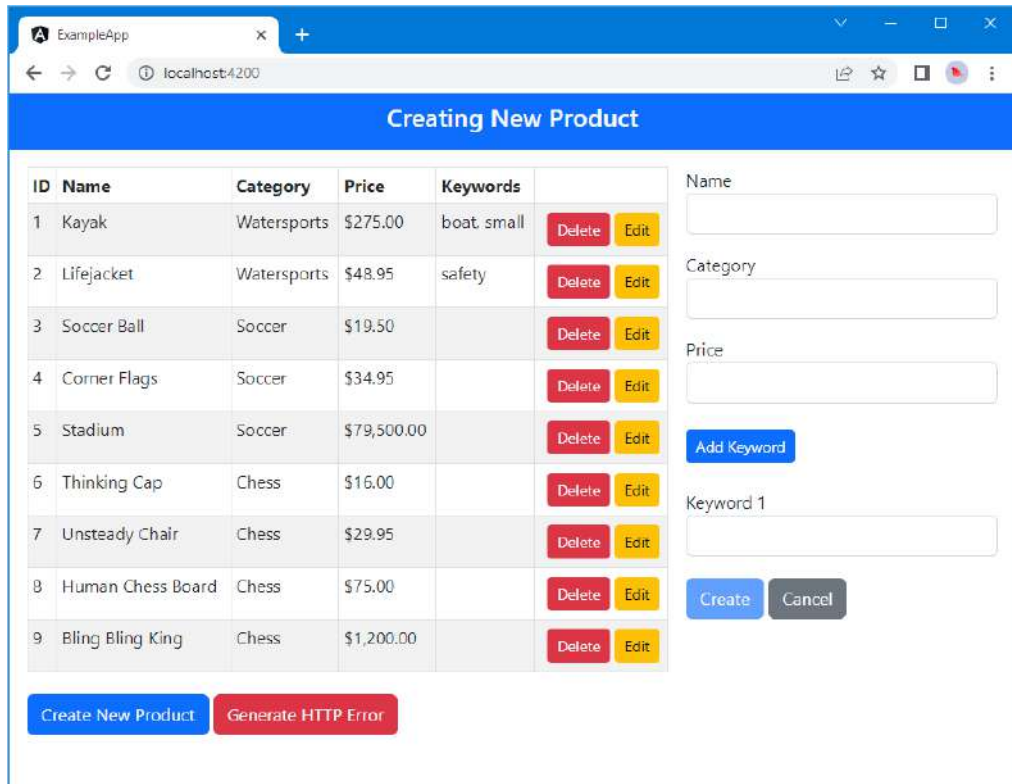


Figure 24.1. Running the example application

24.2 Getting started with routing

At the moment, all the content in the application is visible to the user all of the time. For the example application, this means that both the table and the form are always visible, and it is up to the user to keep track of which part of the application they are using for the task at hand.

That’s fine for a simple application, but it becomes unmanageable in a complex project, which can have many areas of functionality that would be overwhelming if they were all displayed at once.

URL routing adds structure to an application using a natural and well-understood aspect of web applications: the URL. In this section, I am going to introduce URL routing by applying it to the example application so that either the table or the form is visible, with the active component being chosen based on the user’s actions. This will provide a good basis for explaining how routing works and set the foundation for more advanced features.

24.2.1 Creating a routing configuration

The first step when applying routing is to define the *routes*, which are mappings between URLs and the components that will be displayed to the user. Routing configurations are conventionally defined in a file called `app.routing.ts`, defined in the `src/app` folder. I created this file and added the statements shown in listing 24.1.

Listing 24.1. The contents of the `app.routing.ts` file in the `src/app` folder

```
import { Routes, RouterModule } from "@angular/router";
import { tableComponent } from "../core/table.component";
import { FormComponent } from "../core/form.component";

const routes: Routes = [
  { path: "form/edit", component: FormComponent },
  { path: "form/create", component: FormComponent },
  { path: "", component: tableComponent }]

export const routing = RouterModule.forRoot(routes);
```

The `Routes` class defines a collection of routes, each of which tells Angular how to handle a specific URL. This example uses the most basic properties, where the `path` specifies the URL and the `component` property specifies the component that will be displayed to the user.

The `path` property is specified relative to the rest of the application, which means that the configuration in listing 24.1 sets up the routes shown in table 24.3.

Table 24.3. The routes created in the example

URL	Displayed Component
<code>http://localhost:4200/form/edit</code>	<code>FormComponent</code>
<code>http://localhost:4200/form/create</code>	<code>FormComponent</code>
<code>http://localhost:4200/</code>	<code>TableComponent</code>

The routes are packaged into a module using the `RouterModule.forRoot` method. The `forRoot` method produces a module that includes the routing service. There is also a `forChild` method that doesn’t include the service and that is demonstrated in chapter 25, where I explain how to create routes for feature modules.

Although the `path` and `component` properties are the most commonly used when defining routes, there is a range of additional properties that can be used to define routes with advanced features. These properties are described in table 24.4, along with details of where they are described.

Table 24.4. The Routes properties used to define routes

Name	Description
<code>path</code>	This property specifies the path for the route.
<code>component</code>	This property specifies the component that will be selected when the active URL matches the <code>path</code> .
<code>pathMatch</code>	This property tells Angular how to match the current URL to the <code>path</code> property. There are two allowed values: <code>full</code> , which requires the <code>path</code> value to completely match the URL, and <code>prefix</code> , which allows the <code>path</code> value to match the URL, even if the URL contains additional segments that are not part of the <code>path</code> value. This property is required when using the <code>redirectTo</code> property, as demonstrated in chapter 25.
<code>redirectTo</code>	This property is used to create a route that redirects the browser to a different URL when activated. See chapter 25 for details.
<code>children</code>	This property is used to specify child routes, which display additional components in nested <code>router-outlet</code> elements contained in the template of the active component, as demonstrated in chapter 25.
<code>outlet</code>	This property is used to support multiple outlet elements, as described in chapter 26.
<code>resolve</code>	This property is used to define work that must be completed before a route can be activated, as described in chapter 26.
<code>canActivate</code>	This property is used to control when a route can be activated, as described in chapter 26.
<code>canActivateChild</code>	This property is used to control when a child route can be activated, as described in chapter 26.
<code>canDeactivate</code>	This property is used to control when a route can be deactivated so that a new route can be activated, as described in chapter 26.
<code>loadChildren</code>	This property is used to configure a module that is loaded only when it is needed, as described in chapter 26.
<code>canLoad</code>	This property is used to control when an on-demand module can be loaded.

Understanding route ordering

The order in which routes are defined is significant. Angular compares the URL to which the browser has navigated with the path property of each route in turn until it finds a match. This means that the most specific routes should be defined first, with the routes that follow decreasing in specificity. This isn't a big deal for the routes in listing 24.1, but it becomes significant when using route parameters (described in the "Using Route Parameters" section of this chapter) or adding child routes (described in chapter 25).

If you find that your routing configuration doesn't result in the behavior you expect, then the order in which the routes have been defined is the first thing to check.

24.2.2 Creating the routing component

When using routing, the root component is dedicated to managing the navigation between different parts of the application. This is the typical purpose of the `app.component.ts` file that was added to the project by the `ng new` command when it was created. This component is a vehicle for its template, which is the `app.component.html` file in the `src/app` folder. In listing 24.2, I have replaced the default contents.

Listing 24.2. Replacing the contents of the `app.component.html` file in the `src/app` file

```
<pMessages></pMessages>
<router-outlet></router-outlet>
```

The `pMessages` element displays any messages and errors in the application. For routing, it is the `router-outlet` element—known as the *outlet*—that is important because it tells Angular that this is where the component matched by the routing configuration should be displayed.

24.2.3 Updating the root module

The next step is to update the root module so that the new root component is used to bootstrap the application, as shown in listing 24.3, which also imports the module that contains the routing configuration.

Listing 24.3. Enabling routing 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 { ModelModule } from "../model/model.module";
import { CoreModule } from "../core/core.module";
import { TableComponent } from "../core/table.component";
import { FormComponent } from "../core/form.component";
import { MessageModule } from "../messages/message.module";
import { MessageComponent } from "../messages/message.component";
import { AppComponent } from './app.component';
import { routing } from './app.routing';
```



```

@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule, FormsModule, CoreModule, MessageModule,
    routing],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

24.2.4 Completing the Configuration

The final step is to update the `index.html` file, as shown in listing 24.4.

Listing 24.4. Configuring routing in the `index.html` file in the `src` folder

```

<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>ExampleApp</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="m-1">
  <app-root></app-root>
</body>
</html>

```

The `app` element applies the new root component, whose template contains the `router-outlet` element. When you save the changes and the browser reloads the application, you will see just the product table, as illustrated by figure 24.2. The default URL for the application corresponds to the route that shows the product table.

TIP You may need to stop the Angular development tools, start them again using the `ng serve` command, and then reload the browser for this example.

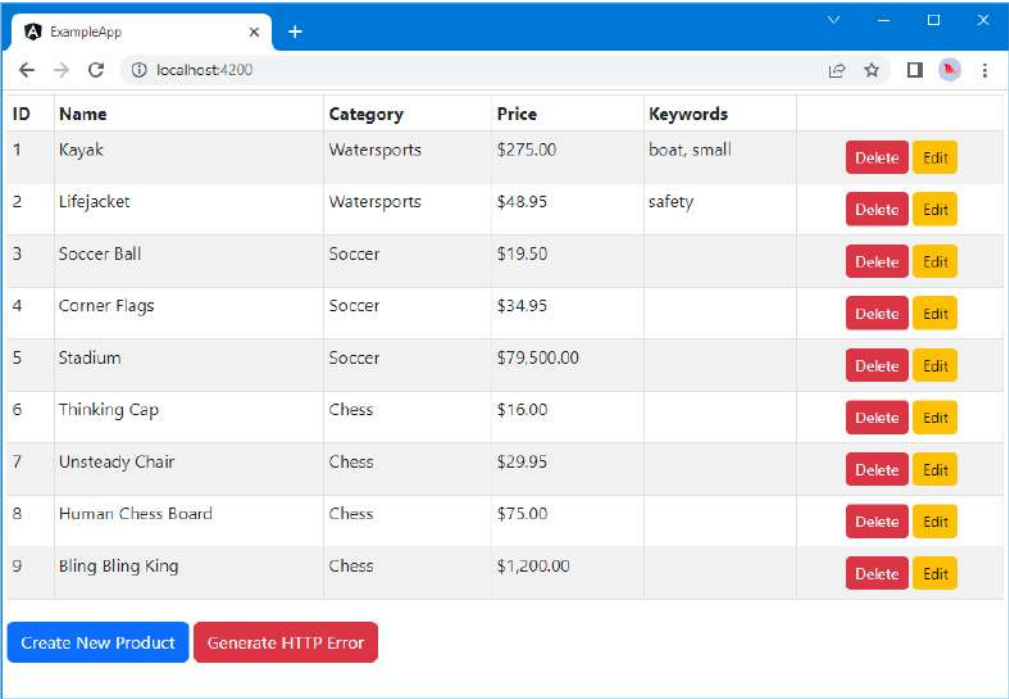


Figure 24.2. Using routing to display components to the user

24.2.5 Adding navigation links

The basic routing configuration is in place, but there is no way to navigate around the application: nothing happens when you click the Create New Product or Edit button.

The next step is to add links to the application that will change the browser’s URL and, in doing so, trigger a routing change that will display a different component to the user. Listing 24.5 adds these links to the table component’s template.

Listing 24.5. Adding links in the table.component.html file in the src/app/core folder

```
<table class="table table-sm table-bordered table-striped">
  <thead>
    <tr>
      <th>ID</th><th>Name</th><th>Category</th>
      <th>Price</th><th>Keywords</th><th></th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let item of Products()">
      <td>{{item.id}}</td>
      <td>{{item.name}}</td>
      <td>{{item.category}}</td>
      <td>{{item.price | currency:"USD" }}</td>
```

```

<td>{{ item.keywords?.join(", ")}}</td>
<td class="text-center">
  <button class="btn btn-danger btn-sm m-1"
    (click)="deleteProduct(item.id)">
    Delete
  </button>
  <button class="btn btn-warning btn-sm"
    (click)="editProduct(item.id)"
    routerLink="/form/edit">
    Edit
  </button>
</td>
</tr>
</tbody>
</table>
<button class="btn btn-primary mt-1" (click)="createProduct()"
  routerLink="/form/create">
  Create New Product
</button>
<button class="btn btn-danger mt-1 mx-1" (click)="deleteProduct(-1)">
  Generate HTTP Error
</button>

```

The `routerLink` attribute applies a directive from the routing package that performs the navigation change. This directive can be applied to any element, although it is typically applied to `button` and `anchor (a)` elements. The expression for the `routerLink` directive applied to the Edit buttons tells Angular to target the `/form/edit` route.

```

...
<button class="btn btn-warning btn-sm" (click)="editProduct(item.id)"
  routerLink="/form/edit">
  Edit
</button>
...

```

The same directive applied to the Create New Product button tells Angular to target the `/create` route.

```

...
<button class="btn btn-primary m-1" (click)="createProduct()"
  routerLink="/form/create">
  Create New Product
</button>
...

```

The routing links added to the table component's template will allow the user to navigate to the form. The addition to the form component's template shown in listing 24.6 will allow the user to navigate back again using the Cancel button.

Listing 24.6. Adding a link in the `form.component.html` file in the `src/app/core` folder

```

...
<div class="mt-2">
  <button type="submit" class="btn btn-primary"
    [class.btn-warning]="editing"
    [disabled]="form.invalid">
    {{editing ? "Save" : "Create"}}
  </button>
  <button type="reset" class="btn btn-secondary m-1" routerLink="/">

```

```

        Cancel
      </button>
    </div>
    ...

```

The value assigned to the `routerLink` attribute targets the route that displays the product table. Listing 24.7 updates the feature module that contains the template so that it imports the `RouterModule`, which is the Angular module that contains the directive that selects the `routerLink` attribute.

Listing 24.7. Enabling the directive in the `core.module.ts` file in the `src/app/core` folder

```

import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule, NG_VALIDATORS, ReactiveFormsModule }
  from "@angular/forms";
import { ModelModule } from "../model/model.module";
import { TableComponent } from "../table.component";
import { FormComponent } from "../form.component";
import { SharedState } from "../sharedState.service";
import { ValidationHelper } from "../validationHelper.pipe";
import { ValidationErrorsDirective } from "../validationErrors.directive";
import { HiLowValidatorDirective } from "../validation/hilow";
import { RouterModule } from "@angular/router";

@NgModule({
  imports: [BrowserModule, FormsModule, ModelModule,
    ReactiveFormsModule, RouterModule],
  declarations: [TableComponent, FormComponent, ValidationHelper,
    ValidationErrorsDirective, HiLowValidatorDirective],
  exports: [ModelModule, TableComponent, FormComponent],
  providers: [SharedState]
})
export class CoreModule { }

```

24.2.6 Understanding the effect of routing

Restart the Angular development tools, and you will be able to navigate around the application using the Edit, Create New Product, and Cancel buttons, as shown in figure 24.3.

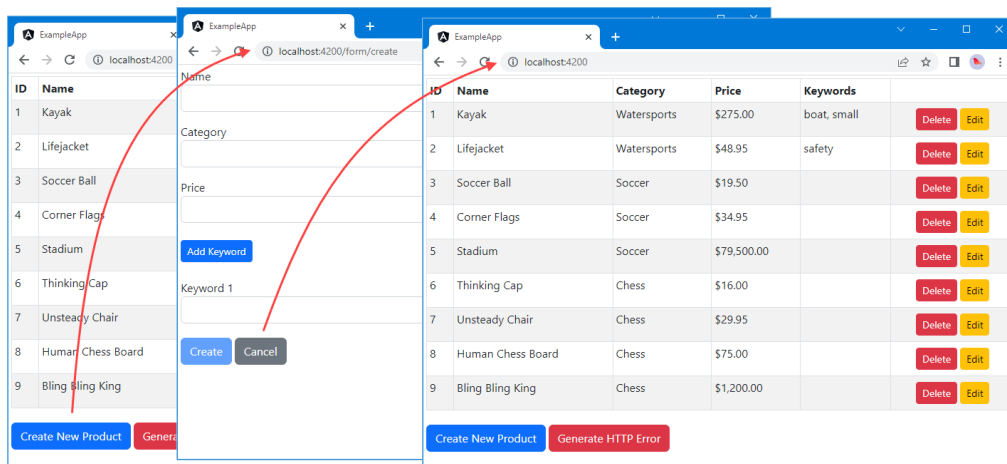


Figure 24.3. Using routes to navigate around the application

Not all the features in the application work yet, but this is a good time to explore the effect of adding routing to the application. Enter the root URL for the application (<http://localhost:4200>) and then click the Create New Product button. When you clicked the button, the Angular routing system changed the URL that the browser displays to this:

`http://localhost:4200/form/create`

If you watch the requests made by the application in the F12 development tools during the transition, you will notice that no requests are sent to the server for new content. This change is done entirely within the Angular application and does not produce any new HTTP requests.

The new URL is processed by the Angular routing system, which can match the new URL to this route from the `app.routing.ts` file.

```
{ path: "form/create", component: FormComponent },
```

The routing system takes into account the base element in the `index.html` file when it matches the URL to a route. The base element is configured with an `href` value of `/` that is combined with the path in the route to make a match when the URL is `/form/create`.

The `component` property tells the Angular routing system that it should display the `FormComponent` to the user. A new instance of the `FormComponent` class is created, and its template content is used as the content for the `router-outlet` element in the root component's template.

If you click the Cancel button below the form, then the process is repeated, but this time, the browser returns to the root URL for the application, which is matched by the route whose path component is the empty string.

```
{ path: "", component: TableComponent }
```

This route tells Angular to display the table `Component` to the user. A new instance of the `TableComponent` class is created, and its template is used as the content of the `router-outlet` element, displaying the model data to the user.

This is the essence of routing: the browser's URL changes, which causes the routing system to consult its configuration to determine which component should be displayed to the user. Lots of options and features are available, but this is the core purpose of routing, and you won't go too far wrong if you keep this in mind.

The perils of changing the URL manually

The `routerLink` directive sets the URL using a JavaScript API that tells the browser that this is a change relative to the current document and not a change that requires an HTTP request to the server.

If you enter a URL that matches the routing system into the browser window, you will see an effect that looks like the expected change but is something else entirely. Keep an eye on the network requests in the F12 development tools while manually entering the following URL into the browser:

```
http://localhost:4200/form/create
```

Rather than handling the change within the Angular application, the browser sends an HTTP request to the server, which reloads the application. Once the application is loaded, the routing system inspects the browser's URL, matches one of the routes in the configuration, and then displays the `FormComponent`.

The reason this works is that the development HTTP server will return the contents of the `index.html` file for URLs that don't correspond to files on the disk. As an example, request this URL:

```
http://localhost:4200/this/does/not/exist
```

The browser will display an error because the request has provided the browser with the contents of the `index.html` file, which it has used to load and start the example Angular application. When the routing system inspects the URL, it finds no matching route and creates an error.

There are two important points to note. The first is that when you test your application's routing configuration, you should check the HTTP requests that the browser is making because you will sometimes see the right result for the wrong reasons. On a fast machine, you may not even realize that the application has been reloaded and restarted by the browser.

Second, you must remember that the URL must be changed using the `routerLink` directive (or one of the similar features provided by the router module) and not manually, using the browser's URL bar.

Finally, since users won't know about the difference between programmatic and manual URL changes, your routing configuration should be able to deal with URLs that don't correspond to routes, as described in chapter 25.

24.3 Completing the routing implementation

Adding routing to the application is a good start, but a lot of the application features just don't work. For example, clicking an Edit button displays the form, but it isn't populated, and it doesn't show the color cue that indicates editing. In the sections that follow, I use features provided by the routing system to finish wiring up the application so that everything works as expected.

24.3.1 Handling route changes in components

The form component isn't working properly because it isn't being notified that the user has clicked a button to edit a product. This problem occurs because the routing system creates new instances of component classes only when it needs them, which means the `FormComponent` object is created only after the Edit button is clicked. If you click the Cancel button under the form and then click an Edit button in this table again, a second instance of the `FormComponent` will be created.

This leads to a timing issue in the way that the product component and the table component communicate, via a `Reactive Extensions Subject`. A `Subject` only passes events to subscribers that arrive after the `subscribe` method has been called. The introduction of routing means that the `FormComponent` object is created after the event describing the edit operation has already been sent.

This problem could be solved by replacing the `Subject` with a `BehaviorSubject`, which sends the most recent event to subscribers when they call the `subscribe` method. But a more elegant approach—especially since this is a chapter on the routing system—is to use the URL to collaborate between components.

Angular provides a service that components can receive to get details of the current route. The relationship between the service and the types that it provides access to may seem complicated at first, but it will make sense as you see how the examples unfold and some of the different ways that routing can be used.

The class on which components declare a dependency is called `ActivatedRoute`. For this section, it defines one important property, which is described in table 24.5. There are other properties, too, which are described later in the chapter but which you can ignore for the moment.

Table 24.5. The `ActivatedRoute` property

Name	Description
<code>snapshot</code>	This property returns an <code>ActivatedRouteSnapshot</code> object that describes the current route.

The `snapshot` property returns an instance of the `ActivatedRouteSnapshot` class, which provides information about the route that led to the current component being displayed to the user using the properties described in table 24.6.

Table 24.6. The Basic ActivatedRouteSnapshot properties

Name	Description
url	This property returns an array of <code>UrlSegment</code> objects, each of which describes a single segment in the URL that matched the current route.
params	This property returns a <code>Params</code> object, which describes the URL parameters, indexed by name.
queryParams	This property returns a <code>Params</code> object, which describes the URL query parameters, indexed by name.
fragment	This property returns a <code>string</code> containing the URL fragment.

The `url` property is the one that is most important for this example because it allows the component to inspect the segments of the current URL and extract information from them. The `url` property returns an array of `UrlSegment` objects, which provide the properties described in table 24.7.

Table 24.7. The `UrlSegment` properties

Name	Description
path	This property returns a <code>string</code> that contains the segment value.
parameters	This property returns an indexed collection of parameters, as described in the “Using Route Parameters” section.

To determine what route has been activated by the user, the form component can declare a dependency on `ActivatedRoute` and then use the object it receives to inspect the segments of the URL, as shown in listing 24.8.

Listing 24.8. Inspecting the route in the `form.component.ts` file in the `src/app/core` folder

```
import { Component } from "@angular/core";
import { FormArray, FormControl, FormGroup, NgForm, 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 } from "../sharedState.service";
import { toObservable } from "@angular/core/rxjs-interop";
import { FilteredFormArray } from "../filteredFormArray";
import { LimitValidator } from "../validation/limit";
import { UniqueValidator } from "../validation/unique";
import { ProhibitedValidator } from "../validation/prohibited";
import { ActivatedRoute } from "@angular/router";

@Component({
  selector: "paForm",
  templateUrl: "form.component.html",
```



```

        styleUrls: ["form.component.css"]
    })
    export class FormComponent {
        product: Product = new Product();
        editing: boolean = false;

        // ...form structure omitted for brevity...

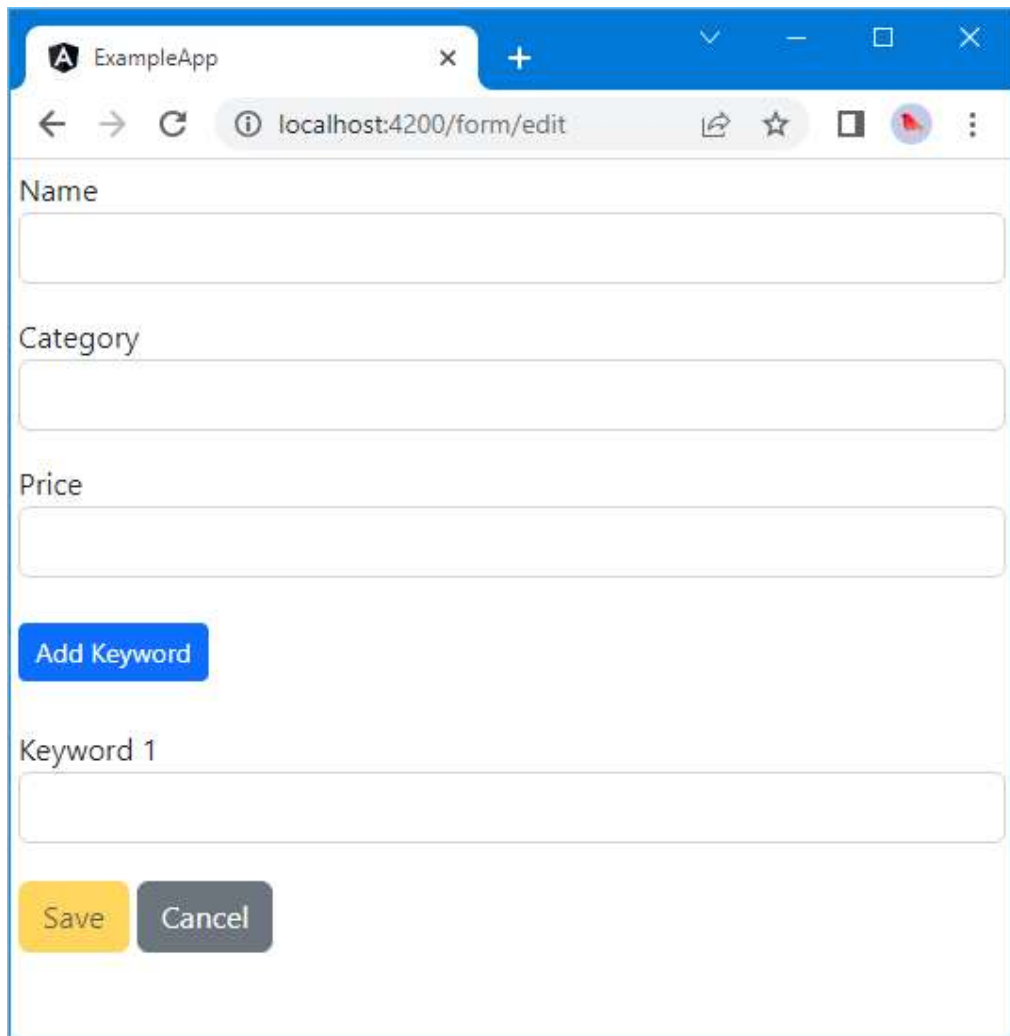
        constructor(private model: Model, activeRoute: ActivatedRoute) {
            this.editing = activeRoute.snapshot.url[1].path == "edit";
        }

        submitForm() {
            if (this.productForm.valid) {
                Object.assign(this.product, this.productForm.value);
                this.model.saveProduct(this.product);
                this.product = new Product();
                this.productForm.reset();
            }
        }

        // ...other methods omitted for brevity...
    }

```

The component no longer uses the shared state service to receive events. Instead, it inspects the second segment of the active route's URL to set the value of the `editing` property, which determines whether it should display its create or edit mode. If you click an Edit button in the table, you will now see the correct coloring displayed, as shown in figure 24.4, although the fields are not yet populated with data.



The screenshot shows a web browser window with the title 'ExampleApp'. The address bar displays 'localhost:4200/form/edit'. The form contains the following elements:

- A text input field labeled 'Name'.
- A text input field labeled 'Category'.
- A text input field labeled 'Price'.
- A blue button labeled 'Add Keyword'.
- A text input field labeled 'Keyword 1'.
- Two buttons at the bottom: a yellow 'Save' button and a grey 'Cancel' button.

Figure 24.4. Using the active route in a component

24.3.2 Using route parameters

When I set up the routing configuration for the application, I defined two routes that targeted the form component, like this:

```
...  
{ path: "form/edit", component: FormComponent },  
{ path: "form/create", component: FormComponent },  
...
```

When Angular is trying to match a route to a URL, it looks at each segment in turn and checks to see that it matches the URL that is being navigated to. Both of these URLs are made up of *static segments*, which means they have to match the navigated URL exactly before Angular will activate the route.

Angular routes can be more flexible and include *route parameters*, which allow any value for a segment to match the corresponding segment in the navigated URL. This means routes that target the same component with similar URLs can be consolidated into a single route, as shown in listing 24.9.

Listing 24.9. Consolidating routes in the app.routing.ts file in the src/app folder

```
import { Routes, RouterModule } from "@angular/router";
import { TableComponent } from "../core/table.component";
import { FormComponent } from "../core/form.component";

const routes: Routes = [
  // { path: "form/edit", component: FormComponent },
  // { path: "form/create", component: FormComponent },
  { path: "form/:mode", component: FormComponent },
  { path: "", component: TableComponent }]

export const routing = RouterModule.forRoot(routes);
```

The second segment of the modified URL defines a route parameter, denoted by the colon (the `:` character) followed by a name. In this case, the route parameter is called `mode`. This route will match any URL that has two segments where the first segment is `form`, as summarized in table 24.8. The content of the second segment will be assigned to a parameter called `mode`.

Table 24.8. URL matching with the route parameter

URL	Result
http://localhost:4200/form	No match—too few segments
http://localhost:4200/form/create	Matches, with <code>create</code> assigned to the <code>mode</code> parameter
http://localhost:4200/form/london	Matches, with <code>london</code> assigned to the <code>mode</code> parameter
http://localhost:4200/product/edit	No match—the first segment is not <code>form</code>
http://localhost:4200/form/edit/1	No match—too many segments

Using route parameters makes it simpler to handle routes programmatically because the value of the parameter can be obtained using its name, as shown in listing 24.10.

Listing 24.10. Reading a route parameter in the form.component.ts file in the src/app/core folder

...

```

    constructor(private model: Model, activeRoute: ActivatedRoute) {
        this.editing = activeRoute.snapshot.params["mode"] == "edit";
    }
    ...

```

The component doesn't need to know the structure of the URL to get the information it needs. Instead, it can use the `params` property provided by the `ActivatedRouteSnapshot` class to get a collection of the parameter values, indexed by name. The component gets the value of the `mode` parameter and uses it to set the `editing` property.

RECEIVING ROUTE DATA WITH INPUT PROPERTIES

Values from the current route can be received using input properties, which can simplify the code in a component. Listing 24.11 replaces the constructor with an input property and an implementation of the `ngOnInit` method, which is called after the route data has been used to populate the inputs.

Listing 24.11. Using an input property in the `form.component.ts` file in the `src/app/core` folder

```

import { Component, Input } from "@angular/core";
import { FormArray, FormControl, FormGroup, NgForm, 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 } from "../sharedState.service";
import { toObservable } from "@angular/core/rxjs-interop";
import { FilteredFormArray } from "../filteredFormArray";
import { LimitValidator } from "../validation/limit";
import { UniqueValidator } from "../validation/unique";
import { ProhibitedValidator } from "../validation/prohibited";
import { ActivatedRoute } from "@angular/router";

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

    // ...form structure omitted for brevity...

    constructor(private model: Model) {
        // this.editing = activeRoute.snapshot.params["mode"] == "edit";
    }

    ngOnInit() {
        this.editing = this.mode == "edit";
    }

    @Input()
    mode?: string;

```

```

    // ...methods omitted for brevity...
  }

```

The value of the `mode` property is set using the value from the current route and the `ngOnInit` method reads the value and uses it to set the `editing` property. The extra step is required because there is a mismatch between the data type received from the route, which is a string, and the value of the `editing` property, which is boolean.

A configuration change is required to enable this feature, as shown in listing 24.12.

Listing 24.12. Enabling Input properties in the `app.routing.ts` file in the `src/app` folder

```

import { Routes, RouterModule } from "@angular/router";
import { TableComponent } from "../core/table.component";
import { FormComponent } from "../core/form.component";

const routes: Routes = [
  { path: "form/:mode", component: FormComponent },
  { path: "", component: TableComponent }]

export const routing = RouterModule.forRoot(routes, {
  bindToComponentInputs: true
});

```

The additional argument to the `forRoot` method is a configuration object that defines a `bindToComponentInputs` property set to `true`. (This is the only configuration option required for this chapter, but there are other configuration setters, described at <https://angular.io/api/router/RouterModule>).

There is no change in the way the application behaves, and the only change is that Angular takes responsibility for extracting the value from the route.

USING MULTIPLE ROUTE PARAMETERS

To tell the form component which product has been selected when the user clicks an Edit button, I need to use a second route parameter. Since Angular matches URLs based on the number of segments they contain, this means I need to split up the routes that target the form component again, as shown in listing 24.13. This cycle of consolidating and then expanding routes is typical of most development projects as you increase the amount of information that is included in routed URLs to add functionality to the application.

Listing 24.13. Adding a route in the `app.routing.ts` file in the `src/app` folder

```

import { Routes, RouterModule } from "@angular/router";
import { TableComponent } from "../core/table.component";
import { FormComponent } from "../core/form.component";

const routes: Routes = [
  { path: "form/:mode/:id", component: FormComponent },
  { path: "form/:mode", component: FormComponent },
  { path: "", component: TableComponent }]

export const routing = RouterModule.forRoot(routes, {
  bindToComponentInputs: true
});

```

The new route will match any URL that has three segments where the first segment is `form`. To create URLs that target this route, I need to use a different approach for the `routerLink` expressions in the template because I need to generate the third segment dynamically for each Edit button in the product table, as shown in listing 24.14.

Listing 24.14. Generating dynamic URLs in the `table.component.html` file in the `src/app/core` folder

```
...
<td class="text-center">
  <button class="btn btn-danger btn-sm m-1"
    (click)="deleteProduct(item.id)">
    Delete
  </button>
  <button class="btn btn-warning btn-sm"
    (click)="editProduct(item.id)"
    [routerLink]="['/form', 'edit', item.id]">
    Edit
  </button>
</td>
...
```

The `routerLink` attribute is now enclosed in square brackets, telling Angular that it should treat the attribute value as a data binding expression. The expression is set out as an array, with each element containing the value for one segment. The first two segments are literal strings and will be included in the target URL without modification. The third segment will be evaluated to include the `id` property value for the current `Product` object being processed by the `ngIf` directive, just like the other expressions in the template. The `routerLink` directive will combine the individual segments to create a URL such as `/form/edit/2`.

Listing 24.15 shows how the form component gets the value of the new route parameter and uses it to select the product that is to be edited.

Listing 24.15. Using the new route parameter in the `form.component.ts` file in the `src/app/core` folder

```
...
ngOnInit() {
  this.editing = this.mode == "edit";
  if (this.id != null) {
    let idVal = parseInt(this.id);
    Object.assign(this.product,
      this.model.getProduct(idVal) || new Product());
    this.productForm.patchValue(this.product);
  }
}

@Input()
mode?: string;

@Input()
id?: string;
...
```

When the user clicks an Edit button, the routing URL that is activated tells the form component that an edit operation is required and specifies the product is to be modified, allowing the form to be populated correctly, as shown in figure 24.5.

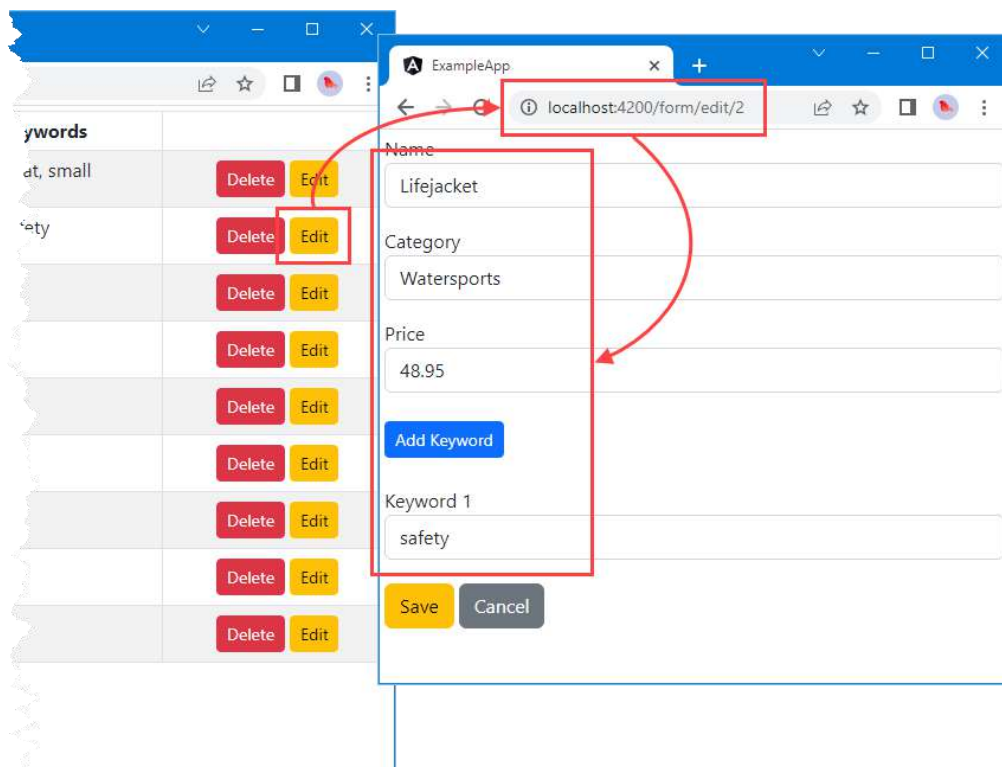


Figure 24.5. Using URLs segments to provide information

Understanding direct user navigation

The introduction of routing has revealed a problem with the way that data is obtained from the web service. If the user starts by requesting `http://localhost:4200` and clicks one of the Edit buttons, then the application works as expected and the form is correctly populated with data.

But if the user navigates directly to the URL for editing a product, such as `http://localhost:4200/form/edit/2`, then the form is never populated with data. This is because the `RestDataSource` class has been written to assume that individual `Product` objects will be accessed only by clicking an Edit button, which can be done only once the data has been received from the web service.

In chapter 26, I explain how you can stop routes from being activated until a specific condition is true, such as the arrival of the data.

USING OPTIONAL ROUTE PARAMETERS

Optional route parameters allow URLs to include information to provide hints or guidance to the rest of the application, but this is not essential for the application to work.

This type of route parameter is expressed using URL matrix notation, which isn't part of the specification for URLs but which browsers support nonetheless. Here is an example of a URL that has optional route parameters:

```
http://localhost:4200/form/edit/2;name=Lifejacket;price=48.95
```

The optional route parameters are separated by semicolons (the `;` character), and this URL includes optional parameters called `name` and `price`.

As a demonstration of how to use optional parameters, listing 24.16 shows the addition of an optional route parameter that includes the object to be edited as part of the URL.

Listing 24.16. An optional route parameter in the `table.component.html` file in the `src/app/core` folder

```
...
<button class="btn btn-warning btn-sm"
        (click)="editProduct(item.id)"
        [routerLink]="['/form', 'edit', item.id,
                     {name: item.name, category: item.category,
                      price: item.price}]">
  Edit
</button>
...
```

The optional values are expressed as literal objects, where property names identify the optional parameter. In this example, there are `name`, `category`, and `price` properties, and their values are set.

Listing 24.17 shows how the form component checks to see whether the optional parameters are present. If they have been included in the URL, then the parameter values are used to avoid a request to the data model.

Listing 24.17. Receiving optional parameters in the `form.component.ts` file in the `src/app/core` folder

```
...
@Input("name")
optionalName?: string;

@Input("category")
optionalCategory?: string;

@Input("price")
optionalPrice?: string;

ngOnInit() {
```



```

this.editing = this.mode == "edit";
if (this.id != null) {
  let idVal = parseInt(this.id);
  Object.assign(this.product,
    this.model.getProduct(idVal) || new Product());
  this.product.name = this.optionalName ?? this.product.name;
  this.product.category = this.optionalCategory
    ?? this.product.category;
  if (this.optionalPrice != undefined) {
    this.product.price = Number.parseFloat(this.optionalPrice);
  }
  this.productForm.patchValue(this.product);
}
}
...

```

The optional parameters in listing 24.16 will produce a URL like this one for the Edit buttons:

```
http://localhost:4200/form/edit/5;name=Stadium;category=Soccer;price=79500
```

Optional route parameters are accessed in the same way as required parameters, and it is the responsibility of the component to check to see whether they are present and to proceed anyway if they are not part of the URL. In this case, the component uses the optional parameter values to override the values from the repository, which you can see by requesting this URL:

```
http://localhost:4200/form/edit/3;name=Soccer%20Ball;category=Football;price=19.5
```

The supplied values are used to populate the form, as shown in figure 24.6. This is an example of direct navigation and, as noted in the sidebar, the content is displayed before the repository is populated with data from the web service.

The screenshot shows a web browser window titled 'ExampleApp'. The address bar displays the URL: `localhost:4200/form/edit/3;name=Soccer%20Ball;category=Football;price=19.5`. The form contains the following elements:

- Name:** A text input field containing 'Soccer Ball'.
- Category:** A text input field containing 'Football'.
- Price:** A text input field containing '19.5'.
- Add Keyword:** A blue button.
- Keyword 1:** A text input field.
- Save:** A yellow button.
- Cancel:** A grey button.

Figure 24.6. Using optional route parameters

24.3.3 Navigating in code

Using the `routerLink` attribute makes it easy to set up navigation in templates, but applications will often need to initiate navigation on behalf of the user within a component or directive.

To give access to the routing system to building blocks such as directives and components, Angular provides the `Router` class, which is available as a service through dependency injection and whose most useful methods and properties are described in table 24.9.

Table 24.9. Selected Router methods and properties

Name	Description
<code>navigated</code>	This boolean property returns <code>true</code> if there has been at least one navigation event and <code>false</code> otherwise.
<code>url</code>	This property returns the active URL.
<code>isActive(url, exact)</code>	This method returns <code>true</code> if the specified URL is the URL defined by the active route. The <code>exact</code> argument specified whether all the segments in the specified URL must match the current URL for the method to return <code>true</code> .
<code>events</code>	This property returns an <code>Observable<Event></code> that can be used to monitor navigation changes. See the “Receiving Navigation Events” section for details.
<code>navigateByUrl(url, extras)</code>	This method navigates to the specified URL. The result of the method is a <code>Promise</code> , which resolves with <code>true</code> when the navigation is successful and <code>false</code> when it is not, and which is rejected when there is an error.
<code>navigate(commands, extras)</code>	This method navigates using an array of segments. The <code>extras</code> object can be used to specify whether the change of URL is relative to the current route. The result of the method is a <code>Promise</code> , which resolves with <code>true</code> when the navigation is successful and <code>false</code> when it is not, and which is rejected when there is an error.

The `navigate` and `navigateByUrl` methods make it easy to perform navigation inside a building block such as a component. Listing 24.18 shows the use of the `Router` in the form component to redirect the application back to the table after a product has been created or updated.

Listing 24.18. Navigating in the `form.component.ts` file in the `src/app/core` folder

```
import { Component, Input } from "@angular/core";
import { FormArray, FormControl, FormGroup, NgForm, 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 } from "../sharedState.service";
import { toObservable } from "@angular/core/rxjs-interop";
import { FilteredFormArray } from "../filteredFormArray";
import { LimitValidator } from "../validation/limit";
import { UniqueValidator } from "../validation/unique";
import { ProhibitedValidator } from "../validation/prohibited";
import { Router } from "@angular/router";

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

  // ...form structure omitted for brevity...

  constructor(private model: Model, private router: Router) {}

  // ...properties and methods omitted for brevity...

  submitForm() {
    if (this.productForm.valid) {
      Object.assign(this.product, this.productForm.value);
      this.model.saveProduct(this.product);
      this.product = new Product();
      this.productForm.reset();
      this.router.navigateByUrl("/");
    }
  }

  // ...methods omitted for brevity...
}

```

The component receives the `Router` object as a constructor argument and uses it in the `submitForm` method to navigate back to the application's root URL. The statements that have been commented out in the `submitForm` method are no longer required because the routing system will destroy the form component once it is no longer on display, which means that resetting the form's state is not required.

The result is that clicking the Save or Create button in the form will cause the application to display the product table, as shown in figure 24.7.

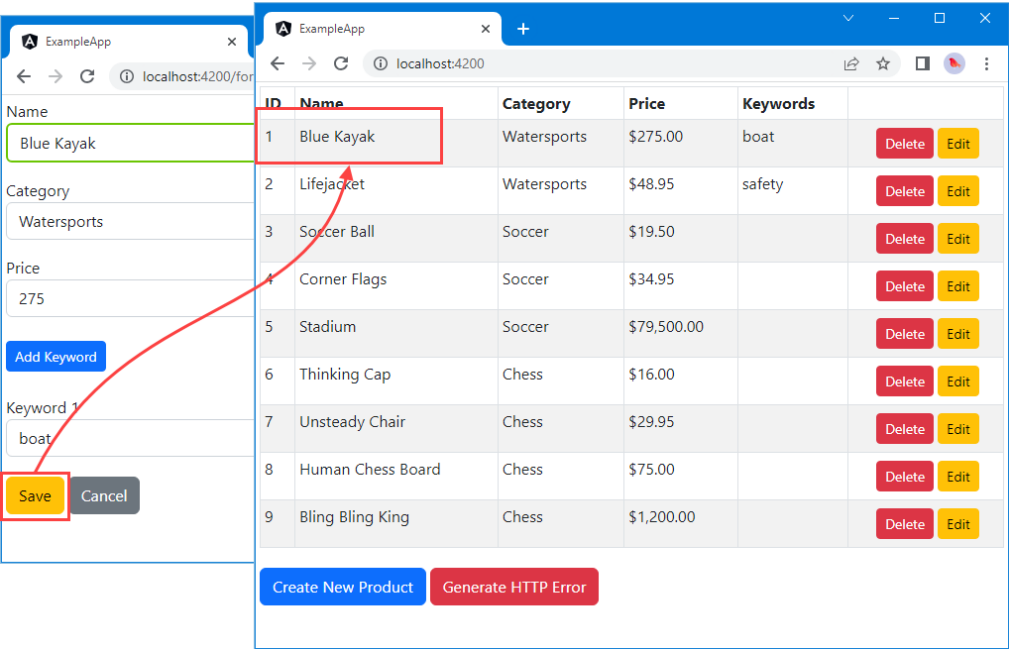


Figure 24.7. Navigating programmatically

24.3.4 Receiving navigation events

In many applications, there will be components or directives that are not directly involved in the application's navigation but that still need to know when navigation occurs. The example application contains an example in the message component, which displays notifications and errors to the user. This component always displays the most recent message, even when that information is stale and unlikely to be helpful to the user. To see the problem, click the Generate HTTP Error button and then click the Create New Product button or one of the Edit buttons; the error message remains on display even though you have navigated elsewhere in the application, as shown in figure 24.8.

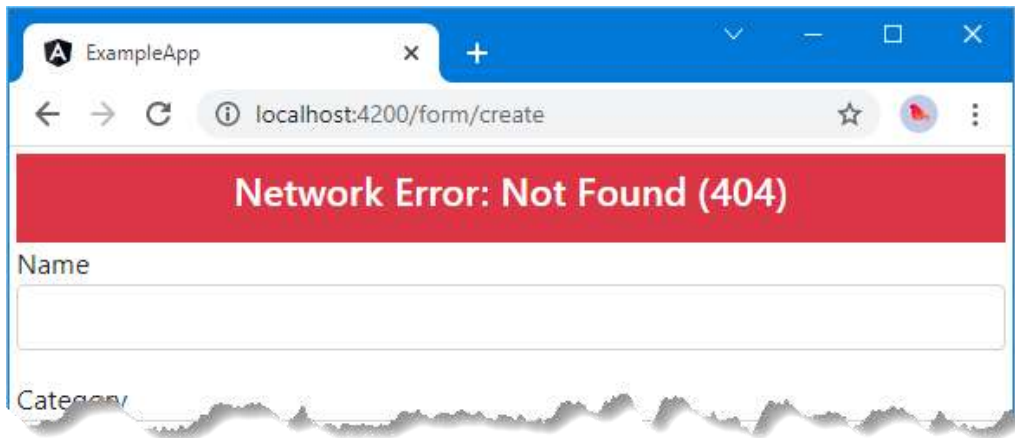


Figure 24.8. An outdated error message

The `events` property defined by the `Router` class returns an `Observable<Event>`, which emits a sequence of `Event` objects describing changes from the routing system. Table 24.10 describes the most useful events.

Table 24.10. Useful events provided by the Router Observer

Name	Description
<code>NavigationStart</code>	This event is sent when the navigation process starts.
<code>RoutesRecognized</code>	This event is sent when the routing system matches the URL to a route.
<code>NavigationEnd</code>	This event is sent when the navigation process completes successfully.
<code>NavigationError</code>	This event is sent when the navigation process produces an error.
<code>NavigationCancel</code>	This event is sent when the navigation process is canceled.
<code>NavigationError</code>	This event is sent when an error arises during navigation.
<code>Scroll</code>	This event is sent when the router scrolls the content to position content during navigation.

All the event classes define an `id` property, which returns a number that is incremented for each navigation, and a `url` property, which returns the target URL. The `RoutesRecognized` and `NavigationEnd` events also define a `urlAfterRedirects` property, which returns the URL that has been navigated to.

To address the issue with the messaging system, listing 24.19 subscribes to the `Observer` provided by the `Router.events` property and clears the message displayed to the user when the `NavigationEnd` or `NavigationCancel` event is received.

Listing 24.19. Responding to Events in the message.component.ts file in the src/app/messages folder

```
import { Component, Signal, computed, signal } from "@angular/core";
import { MessageService } from "../message.service";
import { Message } from "../message.model";
import { NavigationCancel, NavigationEnd, Router } from "@angular/router";

@Component({
  selector: "paMessages",
  templateUrl: "message.component.html",
})
export class MessageComponent {
  lastMessage!: Signal<Message | undefined>;

  constructor(messageService: MessageService, router: Router) {

    let clearedLen = signal(-1);

    router.events.subscribe(ev => {
      if (ev instanceof NavigationCancel
        || ev instanceof NavigationEnd) {
        clearedLen.set(messageService.messages().length);
      }
    });

    this.lastMessage = computed(() => {
      if (messageService.messages().length > clearedLen()) {
        return messageService.messages()
          [messageService.messages().length - 1];
      }
      return undefined;
    });
  }
}
```

A corresponding change is required in the template, as shown in listing 24.20 because messages are cleared using undefined values.

Listing 24.20. Dealing with undefined values in the message.component.html file in the src/app/messages folder

```
<div *ngIf="lastMessage()"
  class="bg-primary text-white p-2 text-center"
  [class.bg-danger]="lastMessage()?.error">
  <h4>{{lastMessage()?.text}}</h4>
</div>
```

The result of these changes is that messages are shown to the user only until the next navigation event, as shown in figure 24.9.

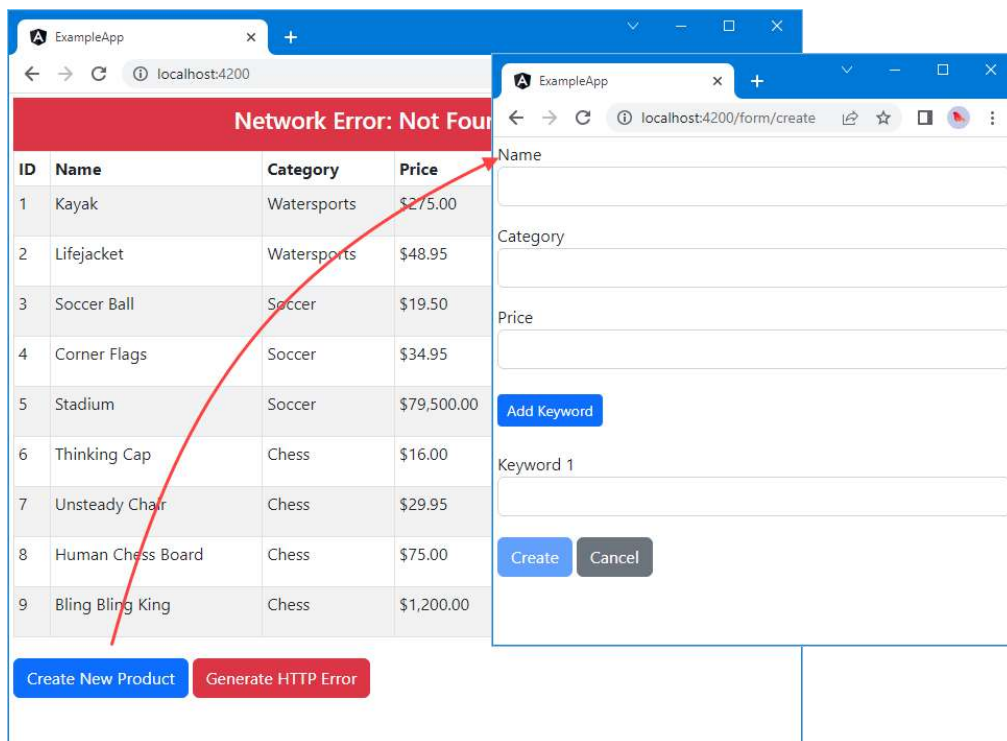


Figure 24.9. Responding to navigation events

24.3.5 Removing the event bindings and supporting code

One of the benefits of using the routing system is that it can simplify applications, replacing event bindings and the methods they invoke with navigation changes. The final change to complete the routing implementation is to remove the last traces of the previous mechanism that was used to coordinate between components. Listing 24.21 removes the event bindings from the table component's template, which were used to respond when the user clicked the Create New Product or Edit button. (The event binding for the Delete buttons is still required because this feature does not relate to navigation.)

Listing 24.21. Removing event bindings in the table.component.html file in the src/app/core folder

```
<table class="table table-sm table-bordered table-striped">
  <thead>
    <tr>
      <th>ID</th><th>Name</th><th>Category</th>
      <th>Price</th><th>Keywords</th><th></th>
    </tr>
  </thead>
```

```

<tbody>
  <tr *ngFor="let item of Products()">
    <td>{{item.id}}</td>
    <td>{{item.name}}</td>
    <td>{{item.category}}</td>
    <td>{{item.price | currency:"USD" }}</td>
    <td>{{ item.keywords?.join(", ")}}</td>
    <td class="text-center">
      <button class="btn btn-danger btn-sm m-1"
        (click)="deleteProduct(item.id)">
        Delete
      </button>
      <button class="btn btn-warning btn-sm"
        [routerLink]="['/form', 'edit', item.id]">
        Edit
      </button>
    </td>
  </tr>
</tbody>
</table>
<button class="btn btn-primary mt-1" routerLink="/form/create">
  Create New Product
</button>
<button class="btn btn-danger mt-1 mx-1" (click)="deleteProduct(-1)">
  Generate HTTP Error
</button>

```

Listing 24.22 shows the corresponding changes in the component, which remove the methods that the event bindings invoked and remove the dependency on the service that was used to signal when a product should be edited or created.

Listing 24.22. Removing event handling code in the `table.component.ts` file in the `src/app/core` folder

```

import { Component, Signal } from "@angular/core";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";
//import { MODES, SharedState } from "../sharedState.service";

@Component({
  selector: "paTable",
  templateUrl: "table.component.html"
})
export class TableComponent {

  constructor(private model: Model) { }

  getProduct(key: number): Product | undefined {
    return this.model.getProduct(key);
  }

  get Products(): Signal<Product[]> {
    return this.model.Products;
  }

  deleteProduct(key?: number) {
    if (key != undefined) {

```



```

        this.model.deleteProduct(key);
    }
}

// editProduct(key?: number) {
//     this.state.update(MODES.EDIT, key)
// }

// createProduct() {
//     this.state.update(MODES.CREATE);
// }
}

```

The service used for coordination by the components is no longer required, and listing 24.23 disables it from the core module.

Listing 24.23. Removing the shared state service in the core.module.ts file in the src/app/core folder

```

import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule, NG_VALIDATORS, ReactiveFormsModule }
    from "@angular/forms";
import { ModelModule } from "../model/model.module";
import { TableComponent } from "../table.component";
import { FormComponent } from "../form.component";
//import { SharedState } from "../sharedState.service";
import { ValidationHelper } from "../validationHelper.pipe";
import { ValidationErrorsDirective } from "../validationErrors.directive";
import { HiLowValidatorDirective } from "../validation/hilow";
import { RouterModule } from "@angular/router";

@NgModule({
    imports: [BrowserModule, FormsModule, ModelModule,
        ReactiveFormsModule, RouterModule],
    declarations: [TableComponent, FormComponent, ValidationHelper,
        ValidationErrorsDirective, HiLowValidatorDirective],
    exports: [ModelModule, TableComponent, FormComponent],
    //providers: [SharedState]
})
export class CoreModule { }

```

The result is that the coordination between the table and form components is handled entirely through the routing system, which is now responsible for displaying the components and managing the navigation between them.

24.4 Summary

In this chapter, I introduced the Angular routing feature and demonstrated how to navigate to a URL in an application to select the content that is displayed to the user. I showed you how to create navigation links in templates, how to perform navigation in a component or directive, and how to respond to navigation changes programmatically.

- Routing allows components to be selected based on the current URL.
- Angular provides directives for navigation within the routing system.

- Components can inspect the current route using a service.
- Routes can be defined with one or more route parameters that can match multiple URLs.

In the next chapter, I continue describing the Angular routing system.

25

Routing and navigation: part 2

This chapter

- Using wildcards to define routes that match multiple URLs
- Using redirections to map from one URL to another
- Performing navigation within a component
- Using CSS styles to reflect changes in routes
- Using child routes to define common route configurations

In the previous chapter, I introduced the Angular URL routing system and explained how it can be used to control the components that are displayed to the user. The routing system has a lot of features, which I continue to describe in this chapter and chapter 26. The emphasis in this chapter is on creating more complex routes, including routes that will match any URL, routes that redirect the browser to other URLs, routes that navigate within a component, and routes that select multiple components. Table 25.1 summarizes the chapter.

Table 25.1. Chapter summary

Problem	Solution	Listing
Matching multiple URLs with a single route	Use routing wildcards	1–8
Redirecting one URL to another	Use a redirection route	9
Navigating within a component	Use a relative URL	10–11
Styling an element when a specific route is active	Use the <code>routerLinkActive</code> attribute	12–14

Using the routing system to display nested components	Define child routes and use the <code>router-outlet</code> element	15–20
---	--	-------

25.1 Preparing the example project

For this chapter, I will continue using the `exampleApp` project that was created in chapter 20 and has been modified in each subsequent chapter. To prepare for this chapter, I have added two methods to the repository class, as shown in listing 25.1.

TIP You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/manningbooks/pro-angular-16>. See chapter 1 for how to get help if you have problems running the examples.

Listing 25.1. Adding methods in the `repository.model.ts` file in the `src/app/model` folder

```
import { Injectable, Signal, computed, signal } from "@angular/core";
import { Product } from "../product.model";
import { RestDataSource } from "../rest.datasource";

@Injectable()
export class Model {
  private products = signal<Product[]>([]);
  private locator = (p: Product, id?: number) => p.id == id;

  constructor(private dataSource: RestDataSource) {
    this.dataSource.getData()
      .subscribe(data => this.products.set(data));
  }

  get Products(): Signal<Product[]> {
    return this.products;
  }

  getProduct(id: number): Product | undefined {
    return this.products().find(p => this.locator(p, id));
  }

  getNextProductId(id?: number): number {
    let nextId = 0;
    let index = this.products().findIndex(p => this.locator(p, id));
    if (index > -1) {
      nextId = this.products()[this.products().length > index + 1
        ? index + 1 : 0].id ?? 0;
    } else {
      nextId = id || 0;
    }
    return nextId;
  }

  getPreviousProductId(id?: number): number {
    let nextId = 0;
    let index = this.products().findIndex(p => this.locator(p, id));
    if (index > -1) {
```

```

        nextId = this.products()[index > 0
            ? index - 1 : this.products().length - 1].id ?? 0;
    } else {
        nextId = id || 0;
    }
    return nextId;
}

saveProduct(product: Product) {
    if (product.id == 0 || product.id == undefined) {
        this.dataSource.saveProduct(product)
            .subscribe(p =>
                this.products.mutate(prods => prods.push(p)));
    } else {
        this.dataSource.updateProduct(product).subscribe(() => {
            this.products.mutate(prods => {
                let index = prods.findIndex(p =>
                    this.locator(p, product.id));
                prods.splice(index, 1, product);
            });
        });
    }
}

deleteProduct(id: number) {
    this.dataSource.deleteProduct(id).subscribe(() => {
        this.products.mutate(prods => {
            let index = prods.findIndex(p => this.locator(p, id));
            if (index > -1) {
                prods.splice(index, 1);
            }
        });
    });
}
}

```

The new methods accept an ID value, locate the corresponding product, and then return the IDs of the next and previous objects in the array that the repository uses to collect the data model objects. I will use this feature later in the chapter to allow the user to page through the set of objects in the data model.

To simplify the example, listing 25.2 removes the statements in the form component that receive the details of the product to edit using optional route parameters.

Listing 25.2. Removing optional parameters in the form.component.ts file in the src/app/core folder

```

import { Component, Input, signal } from "@angular/core";
import { FormArray, FormControl, FormGroup, NgForm, 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 } from "../sharedState.service";
import { toObservable } from "@angular/core/rxjs-interop";
import { FilteredFormArray } from "../filteredFormArray";

```

```

import { LimitValidator } from "../validation/limit";
import { UniqueValidator } from "../validation/unique";
import { ProhibitedValidator } from "../validation/prohibited";
import { ActivatedRoute } from "@angular/router";
import { Router } from "@angular/router";

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

  // ...form structure omitted for brevity...

  constructor(private model: Model, private router: Router) {}

  // @Input("name")
  // optionalName?: string;

  // @Input("category")
  // optionalCategory?: string;

  // @Input("price")
  // optionalPrice?: string;

  nextId = signal(0);
  previousId = signal(0);

  ngOnInit() {
    this.editing = this.mode == "edit";
    if (this.id != null) {
      let idVal = parseInt(this.id);
      Object.assign(this.product,
        this.model.getProduct(idVal) || new Product());
      // this.product.name = this.optionalName ?? this.product.name;
      // this.product.category = this.optionalCategory
      // ?? this.product.category;
      // if (this.optionalPrice != undefined) {
      //   this.product.price
      //     = Number.parseFloat(this.optionalPrice);
      // }
      this.productForm.patchValue(this.product);
      this.nextId.set(this.model.getNextProductId(idVal));
      this.previousId.set(this.model.getPreviousProductId(idVal));
    }
  }

  @Input()
  mode?: string;

  @Input()
  id?: string;

  // ...methods omitted for brevity...

```

```
}
```

25.1.1 Adding components to the project

I need to add some components to the application to demonstrate some of the features covered in this chapter. These components are simple because I am focusing on the routing system, rather than adding useful features to the application. I created a file called `productCount.component.ts` in the `src/app/core` folder and used it to define the component shown in listing 25.3.

TIP You can omit the `selector` attribute from the `@Component` decorator if a component is going to be displayed only through the routing system. I tend to add it anyway so that I can apply the component using an HTML element as well.

Listing 25.3. The `productCount.component.ts` file in the `src/app/core` folder

```
import { Component, Signal, computed } from "@angular/core";
import { Model } from "../model/repository.model";

@Component({
  selector: "paProductCount",
  template: `<div class="bg-info text-white p-2">
    There are {{ count() }} products
  </div>`
})
export class ProductCountComponent {
  count: Signal<number>;

  constructor(private model: Model) {
    this.count = computed(() => {
      return this.model.Products().length;
    });
  }
}
```

This component uses an inline template to display the number of products in the data model, which is updated when the data model changes. Next, I added a file called `categoryCount.component.ts` in the `src/app/core` folder and defined the component shown in listing 25.4.

Listing 25.4. The `categoryCount.component.ts` file in the `src/app/core` folder

```
import { Component, Signal, computed } from "@angular/core";
import { Model } from "../model/repository.model";

@Component({
  selector: "paCategoryCount",
  template: `<div class="bg-primary p-2 text-white">
    There are {{ count() }} categories
  </div>`
})
export class CategoryCountComponent {
  count: Signal<number>;
```

```

    constructor(private model: Model) {
      this.count = computed(() => {
        return this.model.Products()
          .map(p => p.category)
          .filter((category, index, array) =>
            array.indexOf(category) == index)
          .length;
      });
    }
  }
}

```

This component uses a differ to track changes in the data model and count the number of unique categories, which is displayed using a simple inline template. For the final component, I added a file called `notFound.component.ts` in the `src/app/core` folder and used it to define the component shown in listing 25.5.

Listing 25.5. The `notFound.component.ts` file in the `src/app/core` folder

```

import { Component } from "@angular/core";

@Component({
  selector: "paNotFound",
  template: `<h3 class="bg-danger text-white p-2">
    Sorry, something went wrong
  </h3>
  <button class="btn btn-primary" routerLink="/">
    Start Over
  </button>`
})
export class NotFoundComponent {}

```

This component displays a static message that will be shown when something goes wrong with the routing system. Listing 25.6 adds the new components to the core module.

Listing 25.6. Declaring components in the `core.module.ts` file in the `src/app/core` folder

```

import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule, NG_VALIDATORS, ReactiveFormsModule }
  from "@angular/forms";
import { ModelModule } from "../model/model.module";
import { TableComponent } from "../table.component";
import { FormComponent } from "../form.component";
import { ValidationHelper } from "../validationHelper.pipe";
import { ValidationErrorsDirective } from "../validationErrors.directive";
import { HiLowValidatorDirective } from "../validation/hilow";
import { RouterModule } from "@angular/router";
import { ProductCountComponent } from "../productCount.component";
import { CategoryCountComponent } from "../categoryCount.component";
import { NotFoundComponent } from "../notFound.component";

@NgModule({
  imports: [BrowserModule, FormsModule, ModelModule,
    ReactiveFormsModule, RouterModule],
  declarations: [TableComponent, FormComponent, ValidationHelper,
    ValidationErrorsDirective, HiLowValidatorDirective,
    ProductCountComponent, CategoryCountComponent,

```



```

    NotFoundComponent],
    exports: [ModelModule, TableComponent, FormComponent]
  })
  export class CoreModule { }

```

Open a new command prompt, navigate to the `exampleApp` folder, and run the following command to start the server that provides the RESTful web server:

```
npm run json
```

Open a separate command prompt, navigate to the `exampleApp` folder, and run the following command to start the Angular development tools:

```
ng serve
```

Open a new browser window and navigate to `http://localhost:4200` to see the content shown in figure 25.1.

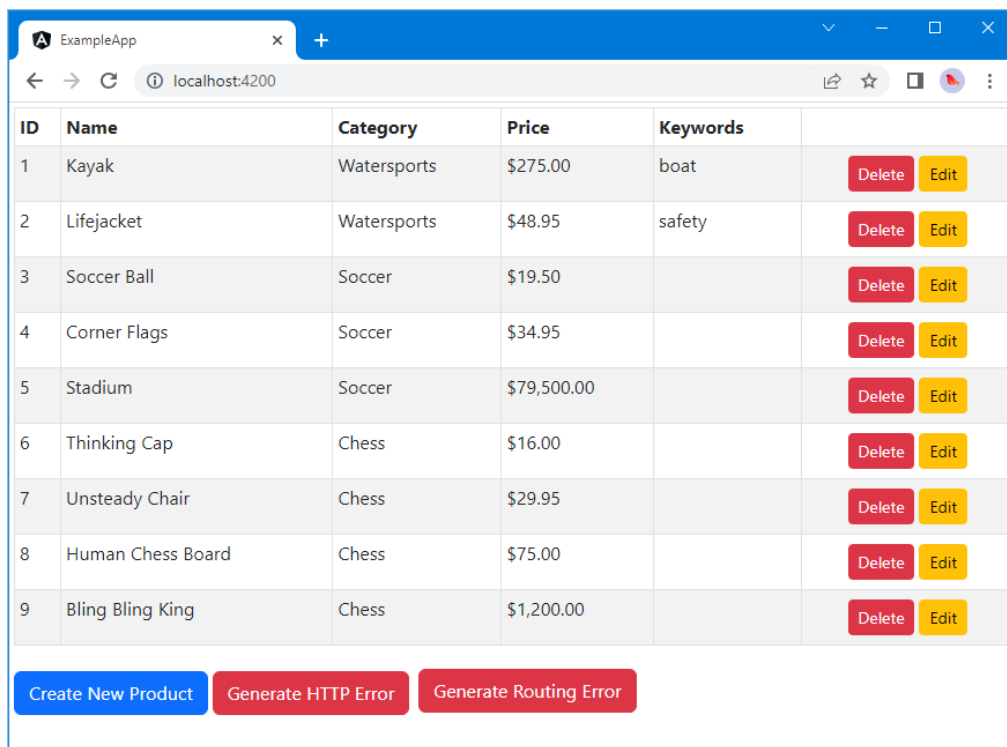


Figure 25.1. Running the example application

25.2 Using wildcards and redirections

The routing configuration in an application can quickly become complex and contain redundancies and oddities to cater to the structure of an application. Angular provides two useful tools that can help simplify routes and also deal with problems when they arise, as described in the following sections.

25.2.1 Using wildcards in routes

The Angular routing system supports a special path, denoted by two asterisks (the `**` characters), that allows routes to match any URL. The basic use of the wildcard path is to deal with navigation that would otherwise create a routing error. Listing 25.7 adds a button to the table component's template that navigates to a route that hasn't been defined by the application's routing configuration.

Listing 25.7. Adding a button in the `table.component.html` file in the `src/app/core` folder

```
<table class="table table-sm table-bordered table-striped">
  <thead>
    <tr>
      <th>ID</th><th>Name</th><th>Category</th>
      <th>Price</th><th>Keywords</th><th></th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let item of Products()">
      <td>{{item.id}}</td>
      <td>{{item.name}}</td>
      <td>{{item.category}}</td>
      <td>{{item.price | currency:"USD" }}</td>
      <td>{{ item.keywords?.join(", ")}}</td>
      <td class="text-center">
        <button class="btn btn-danger btn-sm m-1"
          (click)="deleteProduct(item.id)">
          Delete
        </button>
        <button class="btn btn-warning btn-sm"
          [routerLink]='["/form", "edit", item.id]'">
          Edit
        </button>
      </td>
    </tr>
  </tbody>
</table>
<button class="btn btn-primary mt-1" routerLink="/form/create">
  Create New Product
</button>
<button class="btn btn-danger mt-1 mx-1" (click)="deleteProduct(-1)">
  Generate HTTP Error
</button>
<button class="btn btn-danger m-1" routerLink="/does/not/exist">
  Generate Routing Error
</button>
```

Clicking the button will ask the application to navigate to the URL `/does/not/exist`, for which there is no route configured. When a URL doesn't match a URL, an error is thrown, which is then picked up and processed by the error handling class, which leads to a warning being displayed by the message component, as shown in figure 25.2.

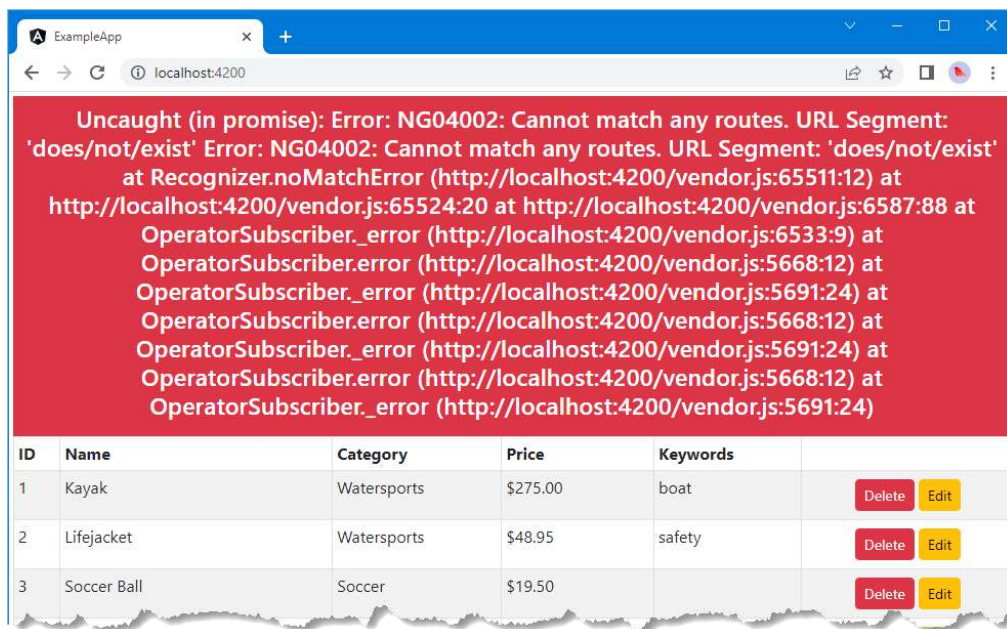


Figure 25.2. The default navigation error

This isn't a useful way to deal with an unknown route because the user won't know what routes are and may not realize that the application was trying to navigate to the problem URL.

A better approach is to use the wildcard route to handle navigation for URLs that have not been defined and select a component that will present a more useful message to the user, as illustrated in listing 25.8.

Listing 25.8. Adding a wildcard route in the app.routing.ts file in the src/app folder

```
import { Routes, RouterModule } from "@angular/router";
import { TableComponent } from "../core/table.component";
import { FormComponent } from "../core/form.component";
import { NotFoundComponent } from "../core/notFound.component";

const routes: Routes = [
  { path: "form/:mode/:id", component: FormComponent },
  { path: "form/:mode", component: FormComponent },
  { path: "", component: TableComponent },
  { path: "**", component: NotFoundComponent }
];

export const routing = RouterModule.forRoot(routes, {
  bindToComponentInputs: true
});
```

The new route in the listing uses the wildcard to select the `NotFoundComponent`, which displays the message shown in figure 25.3 when the Generate Routing Error button is clicked.

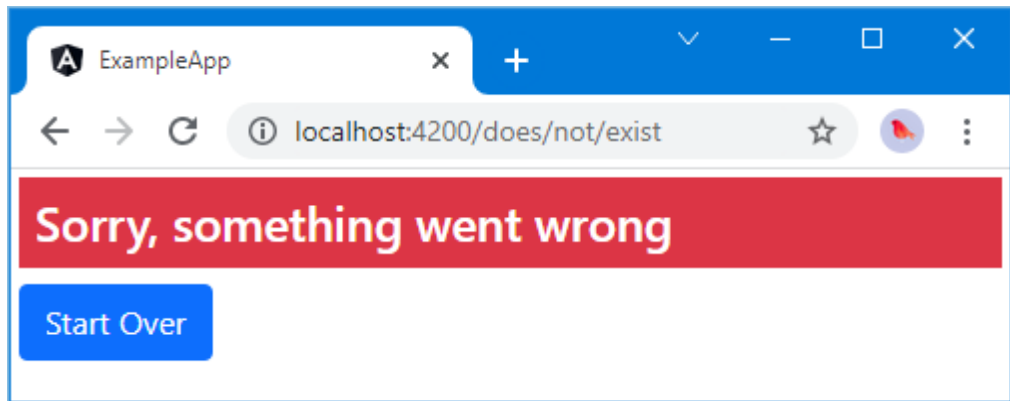


Figure 25.3. Using a wildcard route

Clicking the Start Over button navigates to the `/` URL, which will select the table component for display.

25.2.2 Using redirections in routes

Routes do not have to select components; they can also be used as aliases that redirect the browser to a different URL. Redirections are defined using the `redirectTo` property in a route, as shown in listing 25.9.

Listing 25.9. Using route redirection in the `app.routing.ts` file in the `src/app` folder

```
import { Routes, RouterModule } from "@angular/router";
import { TableComponent } from "../core/table.component";
import { FormComponent } from "../core/form.component";
import { NotFoundComponent } from "../core/notFound.component";

const routes: Routes = [
  { path: "form/:mode/:id", component: FormComponent },
  { path: "form/:mode", component: FormComponent },
  { path: "does", redirectTo: "/form/create", pathMatch: "prefix" },
  { path: "table", component: TableComponent },
  { path: "", redirectTo: "/table", pathMatch: "full" },
  { path: "**", component: NotFoundComponent }
];

export const routing = RouterModule.forRoot(routes, {
  bindToComponentInputs: true
});
```

The `redirectTo` property is used to specify the URL that the browser will be redirected to. When defining redirections, the `pathMatch` property must also be specified, using one of the values described in table 25.2.

Table 25.2. The `pathMatch` values

Name	Description
prefix	This value configures the route so that it matches URLs that start with the specified path, ignoring any subsequent segments.
full	This value configures the route so that it matches only the URL specified by the path property.

The first route added in listing 25.9 specifies a `pathMatch` value of `prefix` and a path of `does`, which means it will match any URL whose first segment is `does`, such as the `/does/not/exist` URL that is navigated to by the Generate Routing Error button. When the browser navigates to a URL that has this prefix, the routing system will redirect it to the `/form/create` URL, as shown in figure 25.4.

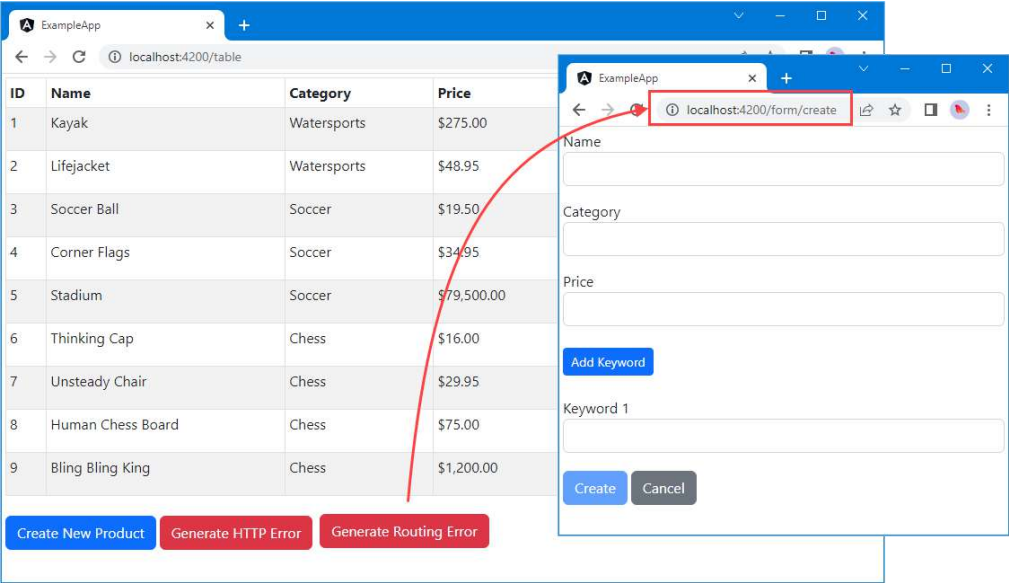


Figure 25.4. Performing a route redirection

The other routes in listing 25.9 redirect the empty path to the `/table` URL, which displays the table component. This is a common technique that makes the URL schema more obvious because it matches the default URL (`http://localhost:4200/`) and redirects it to something more meaningful and memorable to the user (`http://localhost:4200/table`). In this case, the `pathMatch` property value is `full`, although this has no effect since it has been applied to the empty path.

25.3 Navigating within a component

The examples in the previous chapter navigated between different components so that clicking a button in the table component navigates to the form component and vice versa.

This isn't the only kind of navigation that's possible; you can also navigate within a component. To demonstrate, listing 25.10 adds buttons to the form component that allow the user to edit the previous or next data objects.

Listing 25.10. Adding buttons to the form.component.html file in the src/app/core folder

```
<div *ngIf="editing" class="p-2">
  <button class="btn btn-secondary m-1"
    [routerLink]="['/form', 'edit', previousId()]">
    Previous
  </button>
  <button class="btn btn-secondary"
    [routerLink]="['/form', 'edit', nextId()]">
    Next
  </button>
</div>

<form [formGroup]="productForm" #form="ngForm"
  (ngSubmit)="submitForm()" (reset)="resetForm()">

  <!-- ...elements omitted for brevity... -->

</form>
```

These buttons have bindings for the `routerLink` directive with expressions that target the previous and next objects in the data model, using the signals whose values are set using the methods added to the repository at the start of the chapter. This means that if you click the Edit button in the table for the lifejacket, for example, the Next button will navigate to the URL that edits the soccer ball, and the Previous button will navigate to the URL for the kayak. (You must always start from the table view, otherwise, there will be no data available to determine the next or previous product).

25.3.1 Responding to ongoing routing changes

Although the URL changes when the Previous or Next buttons are clicked, there is no change in the data displayed to the user. Angular tries to be efficient during navigation, and it knows that the URLs that the Previous and Next buttons navigate to are handled by the same component that is currently displayed to the user. Rather than create a new instance of the component, it simply processes the route and updates the value of the input properties.

The simplest way to reflect the navigation changes is to change the lifecycle method used by the component to select the product to display, as shown in listing 25.11.

Listing 25.11. Changing lifecycle method in the form.component.ts file in the src/app/core folder

```
...
//ngOnInit() {
ngOnChanges() {
```

```

    this.editing = this.mode == "edit";
    if (this.id != null) {
      let idVal = parseInt(this.id);
      Object.assign(this.product,
        this.model.getProduct(idVal) || new Product());
      this.productForm.patchValue(this.product);
      this.nextId.set(this.model.getNextProductId(idVal));
      this.previousId.set(this.model.getPreviousProductId(idVal));
    }
    ...
  }

```

Using the `ngOnChanges` method ensures that changes in the route select a new product to be displayed.

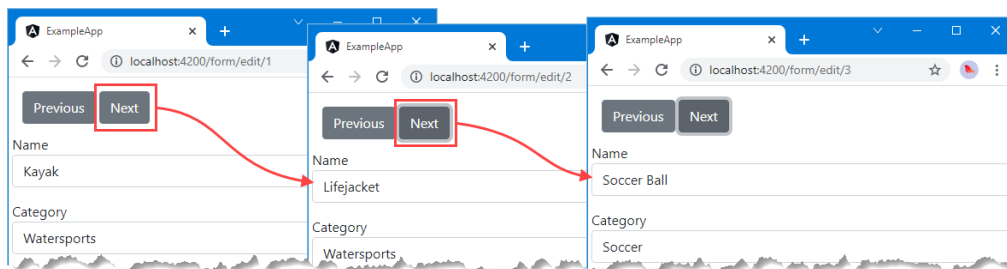


Figure 25.5. Responding to route changes

25.3.2 Styling links for active routes

A common use for the routing system is to display multiple navigation elements alongside the content that they select. To demonstrate, listing 25.12 adds a new route to the application that will allow the table component to be targeted with a URL that contains a category filter.

Listing 25.12. Defining a route in the `app.routing.ts` file in the `src/app` folder

```

import { Routes, RouterModule } from "@angular/router";
import { TableComponent } from "../core/table.component";
import { FormComponent } from "../core/form.component";
import { NotFoundComponent } from "../core/notFound.component";

const routes: Routes = [
  { path: "form/:mode/:id", component: FormComponent },
  { path: "form/:mode", component: FormComponent },
  { path: "does", redirectTo: "/form/create", pathMatch: "prefix" },
  { path: "table/:category", component: TableComponent },
  { path: "table", component: TableComponent },
  { path: "", redirectTo: "/table", pathMatch: "full" },
  { path: "**", component: NotFoundComponent }
];

export const routing = RouterModule.forRoot(routes, {
  bindToComponentInputs: true
});

```

Listing 25.13 updates the `TableComponent` class so that it uses the routing system to get details of the active route. The `category` input property is used to filter the products in the data model.

Listing 25.13. Adding category filter support in the `table.component.ts` file in the `src/app/core` folder

```
import { Component, Input, Signal, computed } from "@angular/core";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";
//import { MODES, SharedState } from "../sharedState.service";

@Component({
  selector: "paTable",
  templateUrl: "table.component.html"
})
export class TableComponent {

  constructor(private model: Model) { }

  @Input()
  category?: string

  getProduct(key: number): Product | undefined {
    return this.model.getProduct(key);
  }

  get Products(): Signal<Product[]> {
    return computed(() => {
      return this.model.Products().filter(p =>
        this.category == null || p.category == this.category);
    });
  }

  get Categories(): Signal<string[]> {
    return computed(() => {
      return this.model.Products()
        .map(p => p.category)
        .filter((c, index, arr) => c != undefined
          && arr.indexOf(c) == index) as string[];
    })
  }

  deleteProduct(key?: number) {
    if (key != undefined) {
      this.model.deleteProduct(key);
    }
  }
}
```

There is also a new `Categories` property that will be used in the template to generate the set of categories for filtering. The final step is to add the HTML elements to the template that will allow the user to apply a filter, as shown in listing 25.14.

Listing 25.14. Adding filter elements in the table.component.html file in the src/app/core folder

```

<div class="container-fluid">
  <div class="row">
    <div class="col-auto">
      <div class="d-grid gap-2">
        <button class="btn btn-secondary"
          routerLink="/" routerLinkActive="bg-primary">
          All
        </button>
        <button *ngFor="let category of Categories()"
          class="btn btn-secondary"
          [routerLink]="['/table', category]"
          routerLinkActive="bg-primary">
          {{category}}
        </button>
      </div>
    </div>
    <div class="col">
      <table class="table table-sm table-bordered table-striped">
        <thead>
          <tr>
            <th>ID</th><th>Name</th><th>Category</th>
            <th>Price</th><th>Keywords</th><th></th>
          </tr>
        </thead>
        <tbody>
          <tr *ngFor="let item of Products()">
            <td>{{item.id}}</td>
            <td>{{item.name}}</td>
            <td>{{item.category}}</td>
            <td>{{item.price | currency:"USD" }}</td>
            <td>{{ item.keywords?.join(", ")}}</td>
            <td class="text-center">
              <button class="btn btn-danger btn-sm m-1"
                (click)="deleteProduct(item.id)">
                Delete
              </button>
              <button class="btn btn-warning btn-sm"
                [routerLink]="['/form', 'edit', item.id]">
                Edit
              </button>
            </td>
          </tr>
        </tbody>
      </table>
    </div>
  </div>
</div>

<div class="p-2 text-center">
  <button class="btn btn-primary mt-1" routerLink="/form/create">
    Create New Product
  </button>
  <button class="btn btn-danger mt-1 mx-1" (click)="deleteProduct(-1)">
    Generate HTTP Error
  </button>
</div>

```

```

</button>
<button class="btn btn-danger m-1" routerLink="/does/not/exist">
  Generate Routing Error
</button>
</div>

```

The important part of this example is the use of the `routerLinkActive` attribute, which is used to specify a CSS class that the element will be assigned to when the URL specified by the `routerLink` attribute matches the active route.

The listing specifies a class called `bg-primary`, which changes the appearance of the button and makes the selected category more obvious. When combined with the functionality added to the component in listing 25.13, the result is a set of buttons allowing the user to view products in a single category, as shown in figure 25.6.

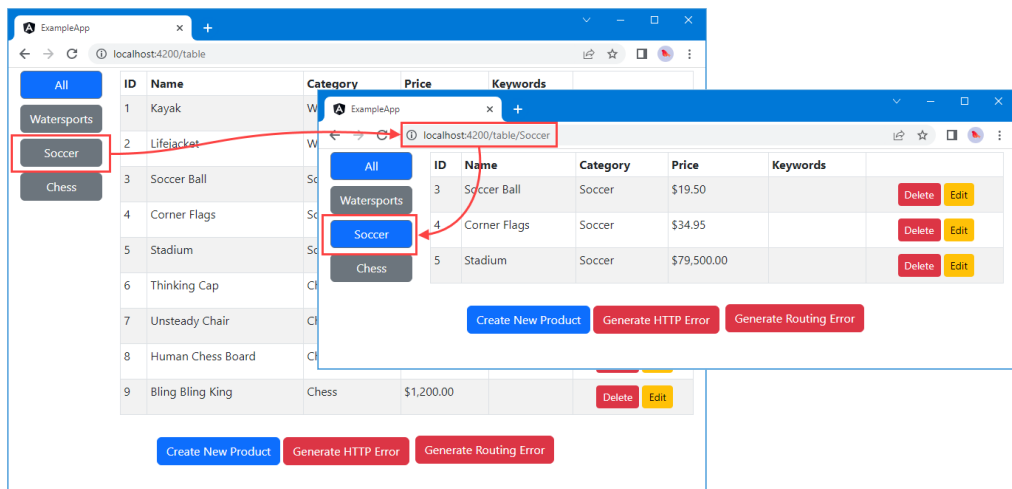


Figure 25.6. Filtering products

If you click the Soccer button, the application will navigate to the `/table/Soccer` URL, and the table will display only those products in the Soccer category. The Soccer button will also be highlighted since the `routerLinkActive` attribute means that Angular will add the button element to the Bootstrap `bg-primary` class.

25.3.3 Fixing the All button

The navigation buttons reveal a common problem, which is that the All button is always added to the active class, even when the user has filtered the table to show a specific category.

This happens because the `routerLinkActive` attribute performs partial matches on the active URL by default. In the case of the example, the `/` URL will always cause the All button to be activated because it is at the start of all URLs. This problem can be fixed by configuring the `routerLinkActive` directive, as shown in listing 25.15.

Listing 25.15. Configuring the directive in the table.component.html file in the src/app/core folder

```

...
<div class="d-grid gap-2">
  <button class="btn btn-secondary"
    routerLink="/table" routerLinkActive="bg-primary"
    [routerLinkActiveOptions]="{exact: true}">
    All
  </button>
  <button *ngFor="let category of Categories()"
    class="btn btn-secondary"
    [routerLink]="['/table', category]"
    routerLinkActive="bg-primary">
    {{category}}
  </button>
</div>
...

```

The configuration is performed using a binding on the `routerLinkActiveOptions` attribute, which accepts a literal object. The `exact` property is the only available configuration setting and is used to control matching the active route URL. Setting this property to `true` will add the element to the class specified by the `routerLinkActive` attribute only when there is an exact match with the active route's URL, which is changed to `/table`. With this change, the `All` button will be highlighted only when all of the products are shown, as illustrated by figure 25.7.

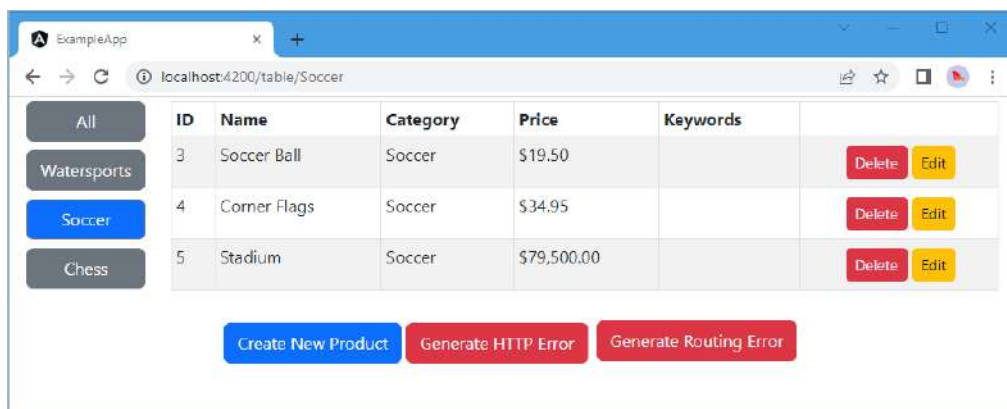


Figure 25.7. Fixing the All button problem

25.4 Creating child routes

Child routes allow components to respond to part of the URL by embedding `router-outlet` elements in their templates, creating more complex arrangements of content. I am going to use the simple components I created at the start of the chapter to demonstrate how child

routes work. These components will be displayed above the product table, and the component that is shown will be specified in the URLs shown in table 25.3.

Table 25.3. The URLs and the components they will select

URL	Component
/table/products	The ProductCountComponent will be displayed.
/table/categories	The CategoryCountComponent will be displayed.
/table	Neither component will be displayed.

Listing 25.16 shows the changes to the application's routing configuration to implement the routing strategy in the table.

Listing 25.16. Configuring routes in the app.routing.ts file in the src/app folder

```
import { Routes, RouterModule } from "@angular/router";
import { TableComponent } from "../core/table.component";
import { FormComponent } from "../core/form.component";
import { NotFoundComponent } from "../core/notFound.component";
import { ProductCountComponent } from "../core/productCount.component";
import { CategoryCountComponent } from "../core/categoryCount.component";

const routes: Routes = [
  { path: "form/:mode/:id", component: FormComponent },
  { path: "form/:mode", component: FormComponent },
  { path: "does", redirectTo: "/form/create", pathMatch: "prefix" },
  {
    path: "table",
    component: TableComponent,
    children: [
      { path: "products", component: ProductCountComponent },
      { path: "categories", component: CategoryCountComponent }
    ]
  },
  { path: "table/:category", component: TableComponent },
  { path: "table", component: TableComponent },
  { path: "", redirectTo: "/table", pathMatch: "full" },
  { path: "**", component: NotFoundComponent }
];

export const routing = RouterModule.forRoot(routes, {
  bindToComponentInputs: true
});
```

Child routes are defined using the `children` property, which is set to an array of routes defined in the same way as the top-level routes. When Angular uses the entire URL to match a route that has children, there will be a match only if the URL to which the browser navigates contains segments that match both the top-level segment and the segments specified by one of the child routes.

TIP Notice that I have added the new route before the one whose path is `table/:category`. Angular tries to match routes in the order in which they are defined. The `table/:category` path would match both the `/table/products` and `/table/categories` URLs and lead the table component to filter the products for nonexistent categories. By placing the more specific route first, the `/table/products` and `/table/categories` URLs will be matched before the `table/:category` path is considered.

25.4.1 Creating the child route outlet

The components selected by child routes are displayed in a `router-outlet` element defined in the template of the component selected by the parent route. In the case of the example, this means the child routes will target an element in the table component's template, as shown in listing 25.17, which also adds elements that will navigate to the new routes.

Listing 25.17. Adding an outlet in the `table.component.html` file in the `src/app/core` folder

```
<div class="container-fluid">
  <div class="row">
    <div class="col-auto">
      <div class="d-grid gap-2">
        <button class="btn btn-secondary"
          routerLink="/table" routerLinkActive="bg-primary"
          [routerLinkActiveOptions]={exact: true}">
          All
        </button>
        <button *ngFor="let category of Categories()"
          class="btn btn-secondary"
          [routerLink]="['/table', category]"
          routerLinkActive="bg-primary">
          {{category}}
        </button>
      </div>
    </div>
    <div class="col">
      <button class="btn btn-info mx-1"
        routerLink="/table/products">
        Count Products
      </button>
      <button class="btn btn-primary mx-1"
        routerLink="/table/categories">
        Count Categories
      </button>
      <button class="btn btn-secondary mx-1" routerLink="/table">
        Count Neither
      </button>
      <div class="my-2">
        <router-outlet></router-outlet>
      </div>
    </div>
  </div>
</div>
<table class="table table-sm table-bordered table-striped">
```

```

        <thead>
          <tr>
            <th>ID</th><th>Name</th><th>Category</th>
            <th>Price</th><th>Keywords</th><th></th>
          </tr>
        </thead>
        <tbody>
          <tr *ngFor="let item of Products()">
            <td>{{item.id}}</td>
            <td>{{item.name}}</td>
            <td>{{item.category}}</td>
            <td>{{item.price | currency:"USD" }}</td>
            <td>{{ item.keywords?.join(", ")}}</td>
            <td class="text-center">
              <button class="btn btn-danger btn-sm m-1"
                (click)="deleteProduct(item.id)">
                Delete
              </button>
              <button class="btn btn-warning btn-sm"
                [routerLink]="['/form', 'edit', item.id]">
                Edit
              </button>
            </td>
          </tr>
        </tbody>
      </table>
    </div>
  </div>
</div>

<div class="p-2 text-center">
  <button class="btn btn-primary mt-1" routerLink="/form/create">
    Create New Product
  </button>
  <button class="btn btn-danger mt-1 mx-1" (click)="deleteProduct(-1)">
    Generate HTTP Error
  </button>
  <button class="btn btn-danger m-1" routerLink="/does/not/exist">
    Generate Routing Error
  </button>
</div>

```

The button elements have `routerLink` attributes that specify the URLs listed in table 25.4, and there is also a `router-outlet` element, which will be used to display the selected component, as shown in figure 25.8, or no component if the browser navigates to the `/table` URL.

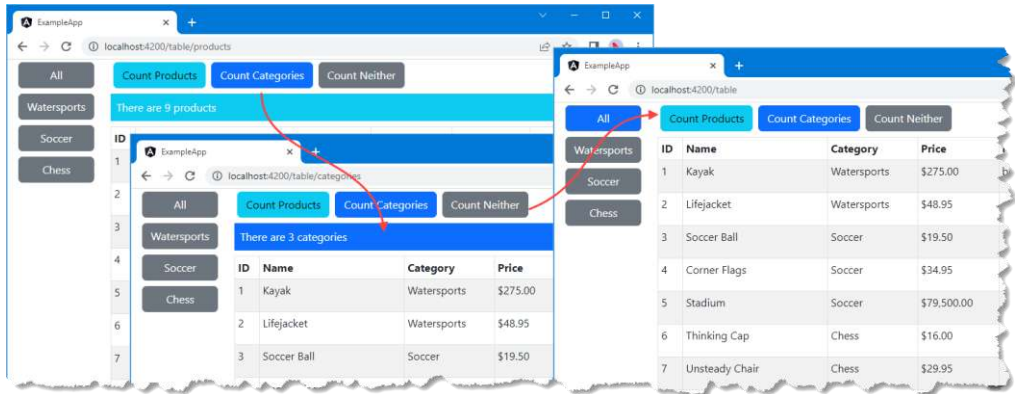


Figure 25.8. Using child routes

25.4.2 Accessing parameters from child routes

Child routes can use all the features available to the top-level routes, including defining route parameters and even having their own child routes. For this section, I am going to add support for the URLs described in table 25.4.

Table 25.4. The New URLs supported by the example application

Name	Description
/table/:category/products	This route will filter the contents of the table and select the ProductCountComponent.
/table/:category/categories	This route will filter the contents of the table and select the CategoryCountComponent.

Listing 25.18 defines the routes that support the URLs shown in the table.

Listing 25.18. Adding routes in the app.routing.ts file in the src/app folder

```
import { Routes, RouterModule } from "@angular/router";
import { TableComponent } from "../core/table.component";
import { FormComponent } from "../core/form.component";
import { NotFoundComponent } from "../core/notFound.component";
import { ProductCountComponent } from "../core/productCount.component";
import { CategoryCountComponent } from "../core/categoryCount.component";

const childRoutes: Routes = [
  { path: "products", component: ProductCountComponent },
  { path: "categories", component: CategoryCountComponent },
  { path: "", component: ProductCountComponent }
];

const routes: Routes = [
  { path: "form/:mode/:id", component: FormComponent },
```

```

    { path: "form/:mode", component: FormComponent },
    { path: "does", redirectTo: "/form/create", pathMatch: "prefix" },
    // {
    //   path: "table",
    //   component: TableComponent,
    //   children: [
    //     { path: "products", component: ProductCountComponent },
    //     { path: "categories", component: CategoryCountComponent }
    //   ]
    // },
    // { path: "table/:category", component: TableComponent },
    // { path: "table", component: TableComponent },
    { path: "table", component: TableComponent, children: childRoutes },
    { path: "table/:category", component: TableComponent,
      children: childRoutes },
    { path: "", redirectTo: "/table", pathMatch: "full" },
    { path: "**", component: NotFoundComponent }]

export const routing = RouterModule.forRoot(routes, {
  bindToComponentInputs: true
});

```

The type of the `children` property is a `Routes` object, which makes it easy to minimize duplication in the route configuration when you need to apply the same set of child routes in different parts of the URL schema. In the listing, I have defined the child routes in a `Routes` object called `childRoutes` and used it as the value for the `children` property in two different top-level routes.

To make it possible to target these new routes, listing 25.19 changes the buttons that appear above the table so they navigate relative to the current URL. I have removed the `Count Neither` button since the `ProductCountComponent` will be shown when the empty path child route matches the URL.

Listing 25.19. Using relative URLs in the `table.component.html` file in the `src/app/core` folder

```

...
<div class="col">

  <button class="btn btn-info mx-1" routerLink="products">
    Count Products
  </button>
  <button class="btn btn-primary mx-1" routerLink="categories">
    Count Categories
  </button>
  <!--<button class="btn btn-secondary mx-1" routerLink="/table">
    Count Neither
  </button> -->
  <div class="my-2">
    <router-outlet></router-outlet>
  </div>

  <table class="table table-sm table-bordered table-striped">
...

```


Additional work is required to reflect the change of child routes in components that rely on them. In the case of the routes added in listing 25.19, this means the `category` input property defined by the `ProductCountComponent` component is updated to reflect new active routes, but this change doesn't cause the computed signal to be recalculated because the input property isn't a signal itself. Listing 25.20 introduces another signal and implements the `ngOnChanges` method to ensure that the input property changes are correctly reflected in the value produced by the count computed signal.

Listing 25.20. Adding a signal in the `productCount.component.ts` file in the `src/app/core` folder

```
import { Component, Input, Signal, WritableSignal, computed, signal }
  from "@angular/core";
import { Model } from "../model/repository.model";

@Component({
  selector: "paProductCount",
  template: `<div class="bg-info text-white p-2">
    There are {{ count() }} products
  </div>`
})
export class ProductCountComponent {
  count: Signal<number>;
  categorySignal: WritableSignal<string|undefined> = signal(undefined);

  @Input()
  category?: string

  constructor(private model: Model) {
    this.count = computed(() => {
      return this.model.Products()
        .filter(p => this.categorySignal() == undefined
          || this.categorySignal() == p.category)
        .length;
    });
  }

  ngOnChanges() {
    this.categorySignal.set(this.category);
  }
}
```

To see the result, save the changes, click the Watersports button to filter the contents of the table, and then click the Count Products button, which selects the `ProductCountComponent`. This number of products reported by the component will correspond to the number of rows in the table, as shown in figure 25.9.

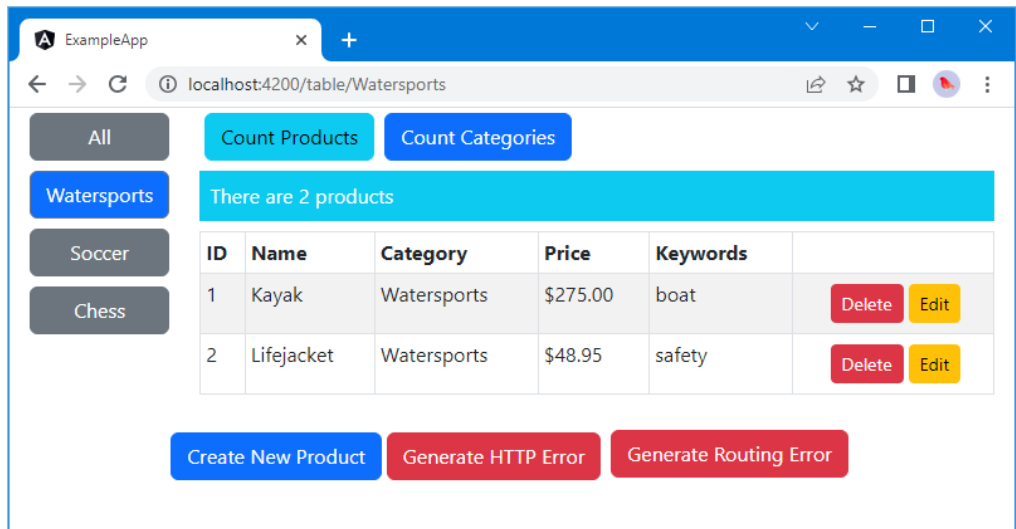


Figure 25.9. Ensuring child routes are reflected in component output

25.5 Summary

In this chapter, I continued to describe the features provided by the Angular URL routing system, going beyond the basic features described in the previous chapter. I explained how to create wildcard and redirection routes, how to create routes that navigate relative to the current URL, and how to create child routes to display nested components.

- Wildcards allow routes to match multiple URLs and can be used to create fallback routes.
- Redirection routes will trigger a redirection to a different URL path.
- Navigation within a component allows a component to adapt to URL changes by displaying different content.
- Elements can be styled to reflect route changes.
- Child routes allow a path structure to be defined once and applied repeatedly.

In the next chapter, I finish describing the URL routing system, focusing on the most advanced features.

26

Routing and navigation: part 3

This chapter covers

- Using resolvers and route guards to control route activation
- Displaying placeholder content to the user until data is loaded

In this chapter, I continue to describe the Angular URL routing system, focusing on the most advanced features. I explain how to control route activation, how to load feature modules dynamically, and how to use multiple outlet elements in a template. Table 26.1 summarizes the chapter.

Table 26.1. Chapter summary

Problem	Solution	Listing
Delaying navigation until a task is complete	Use a route resolver	1–6
Preventing route activation	Use an activation guard	7–13
Preventing the user from navigating away from the current content	Use a deactivation guard	14–17

26.1 Preparing the example project

For this chapter, I will continue using the `exampleApp` project that was created in chapter 20 and has been modified in each subsequent chapter. No changes are required for this chapter.

TIP You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/manningbooks/pro-angular-16>. See chapter 1 for how to get help if you have problems running the examples.

Open a new command prompt, navigate to the `exampleApp` folder, and run the following command to start the server that provides the RESTful web server:

```
npm run json
```

Open a separate command prompt, navigate to the `exampleApp` folder, and run the following command to start the Angular development tools:

```
ng serve
```

Open a new browser window and navigate to `http://localhost:4200` to see the content shown in figure 26.1.

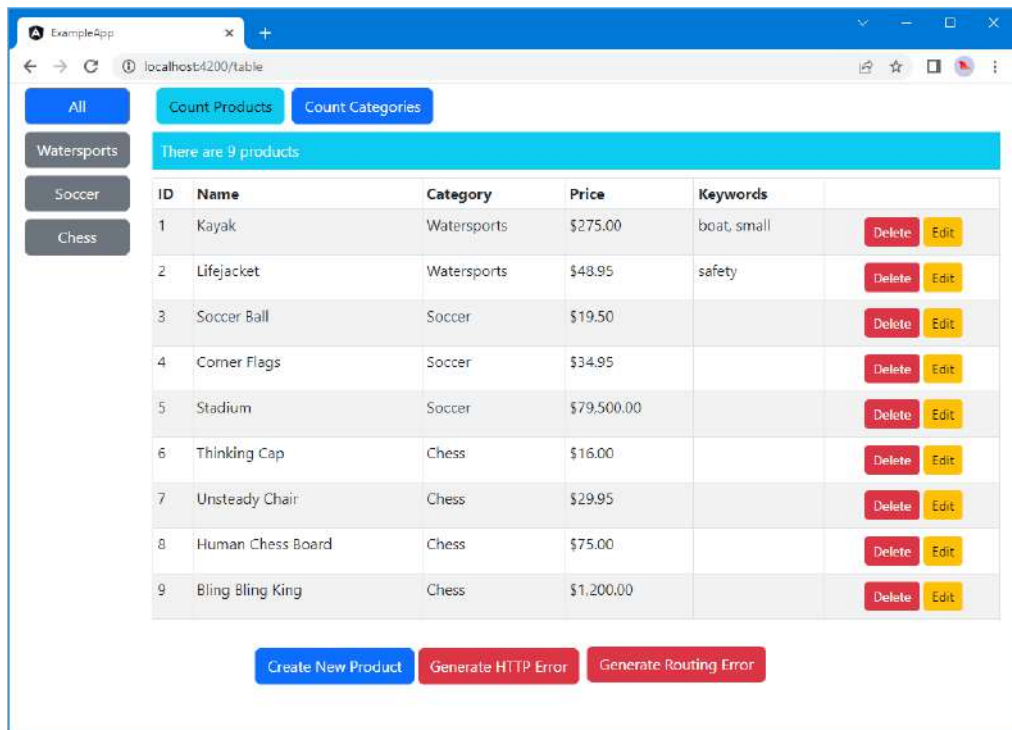


Figure 26.1. Running the example application

26.2 Guarding routes

At the moment, the user can navigate anywhere in the application at any time. This isn't always a good idea, either because some parts of the application may not always be ready or because some parts of the application are restricted until specific actions are performed. To control the use of navigation, Angular supports *guards*, which are specified as part of the route configuration using the properties defined by the `Routes` class, the most useful of which are described in table 26.2.

Table 26.2. Useful routes properties for guards

Name	Description
<code>resolve</code>	This property is used to specify guards that will delay route activation until some operation has been completed, such as loading data from a server.
<code>canActivate</code>	This property is used to specify the guards that will be used to determine whether a route can be activated.
<code>canActivateChild</code>	This property is used to specify the guards that will be used to determine whether a child route can be activated.
<code>canDeactivate</code>	This property is used to specify the guards that will be used to determine whether a route can be deactivated.

26.2.1 Delaying navigation with a resolver

A common reason for guarding routes is to ensure that the application has received the data that it requires before a route is activated. The example application loads data from the RESTful web service asynchronously, which means there can be a delay between the moment at which the browser is asked to send the HTTP request and the moment at which the response is received and the data is processed. You may not have noticed this delay as you followed the examples because the browser and the web service are running on the same machine. In a deployed application, there is a much greater prospect of there being a delay, caused by network congestion, a high server load, or a dozen other factors.

To simulate network congestion, listing 26.1 modifies the RESTful data source class to introduce a delay after the response is received from the web service.

Listing 26.1. Adding a delay in the `rest.datasource.ts` file in the `src/app/model` folder

```
import { Injectable, Signal } from "@angular/core";
import { HttpClient, HttpHeaders } from "@angular/common/http";
import { Product } from "../product.model";
import { Observable, catchError, delay } from "rxjs";

export const REST_URL = `http://${location.hostname}:3500/products`;

@Injectable()
export class RestDataSource {

  constructor(private http: HttpClient) { }

  getData(): Observable<Product[]> {
    return this.sendRequest<Product[]>("GET", REST_URL)
      .pipe(delay(5000));
  }

  saveProduct(product: Product): Observable<Product> {
    return this.sendRequest<Product>("POST", REST_URL, product);
  }
}
```

```

updateProduct(product: Product): Observable<Product> {
    return this.sendRequest<Product>("PUT",
        `${REST_URL}/${product.id}`, product);
}

deleteProduct(id: number): Observable<Product> {
    return this.sendRequest<Product>("DELETE", `${REST_URL}/${id}`);
}

private sendRequest<T>(verb: string, url: string, body?: Product)
    : Observable<T> {
    return this.http.request<T>(verb, url, {
        body: body,
        headers: new HttpHeaders({
            "Access-Key": "<secret>",
            "Application-Name": "exampleApp"
        })
    }).pipe(catchError((error: Response) => {
        throw `Network Error: ${error.statusText} (${error.status})`
    }));
}
}

```

The delay is added using the Reactive Extensions `delay` function and is applied to create a five-second delay, which is long enough to create a noticeable pause without being too painful to wait for every time the application is reloaded. To change the delay, increase or decrease the argument for the `delay` method, which is expressed in milliseconds.

The effect of the delay is that the user is presented with an incomplete and confusing layout while the application is waiting for the data to load, as shown in figure 26.2.

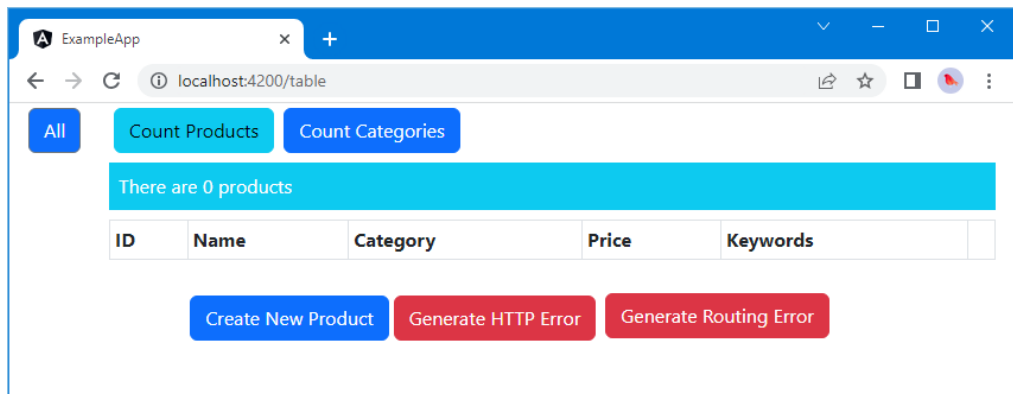


Figure 26.2. Waiting for data

CREATING A RESOLVER SERVICE

A *resolver* is used to ensure that a task is performed before a route can be activated. To create a resolver, I added a file called `model.resolver.ts` in the `src/app/model` folder and defined the class shown in listing 26.2.

Listing 26.2. The contents of the `model.resolver.ts` file in the `src/app/model` folder

```
import { Injectable, effect } from "@angular/core";
import { ActivatedRouteSnapshot, RouterStateSnapshot }
  from "@angular/router";
import { Model } from "../repository.model";
import { Product } from "../product.model";

@Injectable()
export class ModelResolver {
  private promise: Promise<Product[]>;

  constructor(private model: Model) {
    this.promise = new Promise((resolve) => {
      effect(() => {
        if (this.model.Products().length > 0) {
          resolve(this.model.Products());
        }
      })
    });
  }

  resolve(route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot) {
    return this.promise;
  }
}
```

Resolvers are functions that accept two arguments. The first argument is an `ActivatedRouteSnapshot` object, which describes the route that is being navigated to using the properties described in chapter 24. The second argument is a `RouterStateSnapshot` object, which describes the current route through a single property called `url`.

Resolves can be defined as standalone functions, but I prefer to define classes that have `resolve` methods so that I can use the class constructor to receive services via dependency injection, which is the approach I have taken in listing XXX-1. This resolver

The arguments provided to the `resolve` method/function can be used to adapt the resolver to the navigation that is about to be performed, although neither is required by the resolver in the listing, which uses the same behavior regardless of the routes that are being navigated to and from.

The `resolve` method can return three different types of result, as described in table 26.3.

Table 26.3. The result types allowed by the `resolve` method

Result Type	Description
<code>Observable<any></code>	The browser will activate the new route when the <code>Observer</code> emits an event.

<code>Promise<any></code>	The browser will activate the new route when the <code>Promise</code> resolves.
Any other result	The browser will activate the new route as soon as the method produces a result.

The `Observable` and `Promise` results are useful when dealing with asynchronous operations, such as requesting data using an HTTP request. Angular waits until the asynchronous operation is complete before activating the new route. Any other result is interpreted as the result of a synchronous operation, and Angular will activate the new route immediately.

The resolver in listing 26.2 uses its constructor to receive a `Model` object via dependency injection, which it uses to create a `Promise<Product[]>` that is resolved when data is received into the model. Keeping track of the data is done with a signal effect, which is triggered every time the model's `Products` signal is modified.

NOTE The functions used to create signals, including `effect`, can only be used where Angular will inject services. It is for this reason that I have to create the effect signal in the resolver's constructor and not the `resolve` method.

When the `resolve` method is called, it returns the promise, which doesn't resolve until data has been received. Angular will delay activating the new route until the promise is resolved.

Angular will call the guard's `resolve` method every time that the application tries to navigate to a route to which the resolver has been applied and so it is important to ensure that the promise or observable used by the guard won't block subsequent navigation.

Notice that I don't care about the data produced by the repository through the signal. All that matters from the perspective of the guard is that the promise is resolved once data has been received.

REGISTERING THE RESOLVER SERVICE

The next step is to register the resolver as a service in its feature module, as shown in listing 26.3.

Listing 26.3. Registering the resolver in the `model.module.ts` file in the `src/app/model` folder

```
import { NgModule } from "@angular/core";
import { StaticDataSource } from "../static.datasource";
import { Model } from "../repository.model";
import { HttpClientModule } from "@angular/common/http";
import { RestDataSource } from "../rest.datasource";
import { ModelResolver } from "../model.resolver";

@NgModule({
  imports: [HttpClientModule],
  providers: [Model, RestDataSource, ModelResolver]
})
export class ModelModule { }
```


APPLYING THE RESOLVER

The resolver is applied to routes using the `resolve` property, as shown in listing 26.4.

Listing 26.4. Applying a resolver in the `app.routing.ts` file in the `src/app` folder

```
import { Routes, RouterModule, mapToResolve } from "@angular/router";
import { TableComponent } from "../core/table.component";
import { FormComponent } from "../core/form.component";
import { NotFoundComponent } from "../core/notFound.component";
import { ProductCountComponent } from "../core/productCount.component";
import { CategoryCountComponent } from "../core/categoryCount.component";
import { ModelResolver } from "../model/model.resolver";

const childRoutes: Routes = [
  {
    path: "",
    children: [
      { path: "products", component: ProductCountComponent },
      {
        path: "categories",
        component: CategoryCountComponent,
        children: [
          { path: "", component: ProductCountComponent }
        ],
        resolve: { model: mapToResolve(ModelResolver) }
      }
    ]
  }
];

const routes: Routes = [
  { path: "form/:mode/:id", component: FormComponent },
  { path: "form/:mode", component: FormComponent },
  { path: "does", redirectTo: "/form/create", pathMatch: "prefix" },
  { path: "table", component: TableComponent, children: childRoutes },
  { path: "table/:category", component: TableComponent,
    children: childRoutes },
  { path: "", redirectTo: "/table", pathMatch: "full" },
  { path: "**", component: NotFoundComponent }
];

export const routing = RouterModule.forRoot(routes, {
  bindToComponentInputs: true
});
```

The `resolve` property accepts a map object whose property values are the resolver functions that will be applied to the route (the property names have no meaning and can be chosen freely). When using a class resolver, the `mapToResolve` function is used to adapt the class `resolve` method to be a function.

I want to apply the resolver to all the views that display the product table, so to avoid duplication, I created a route with the `resolve` property and used it as the parent for the existing child routes.

The effect is that the user will see no content until the data has been received from the web service and processed by the application, as shown in figure 26.3.

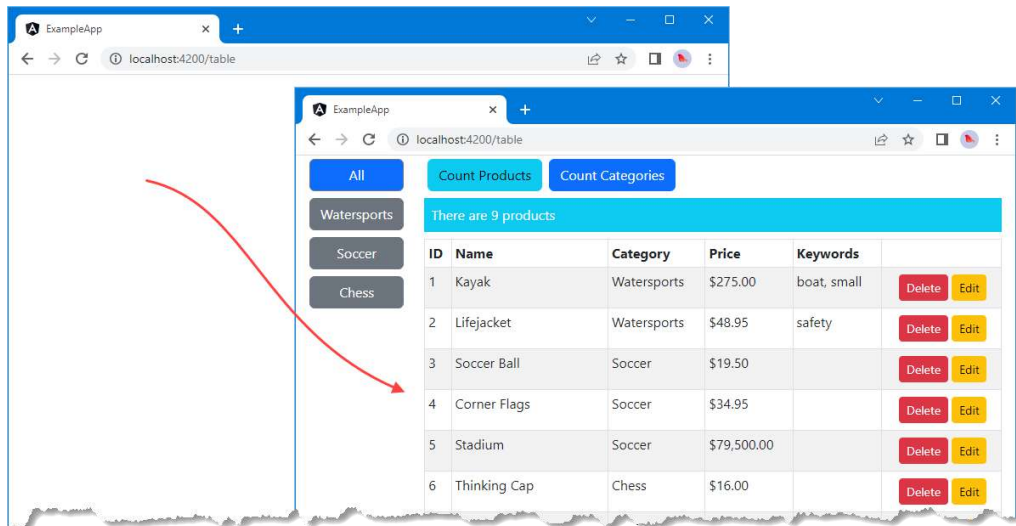


Figure 26.3. Using a route guard

DISPLAYING PLACEHOLDER CONTENT

Angular uses the resolver before activating any of the routes to which it has been applied, which prevents the user from seeing the product table until the model has been populated with the data from the web service. Sadly, that just means the user sees an empty window while the browser is waiting for the server to respond. To address this, listing 26.5 enhances the resolver to use the message service to tell the user what is happening when the data is being loaded.

Listing 26.5. Displaying a message in the `model.resolver.ts` file in the `src/app/model` folder

```
import { Injectable, effect } from "@angular/core";
import { ActivatedRouteSnapshot, RouterStateSnapshot }
  from "@angular/router";
import { Model } from "../repository.model";
import { Product } from "../product.model";
import { MessageService } from "../messages/message.service";
import { Message } from "../messages/message.model";

@Injectable()
export class ModelResolver {
  private promise: Promise<Product[]>;

  constructor(private model: Model, private messages: MessageService) {
    this.promise = new Promise((resolve) => {
      effect(() => {
        if (this.model.Products().length > 0) {
          resolve(this.model.Products());
        }
      });
    });
  }
}
```

```

    })
  });
}

resolve(route: ActivatedRouteSnapshot,
state: RouterStateSnapshot) {
  this.messages.reportMessage(new Message("Loading data..."));
  return this.promise;
}
}

```

The guard uses the message service to give the user an indication that something is happening and relies on the way that the service removes messages when navigation events are received. The result is that the user sees a loading message until data is received, as shown in figure 26.4.

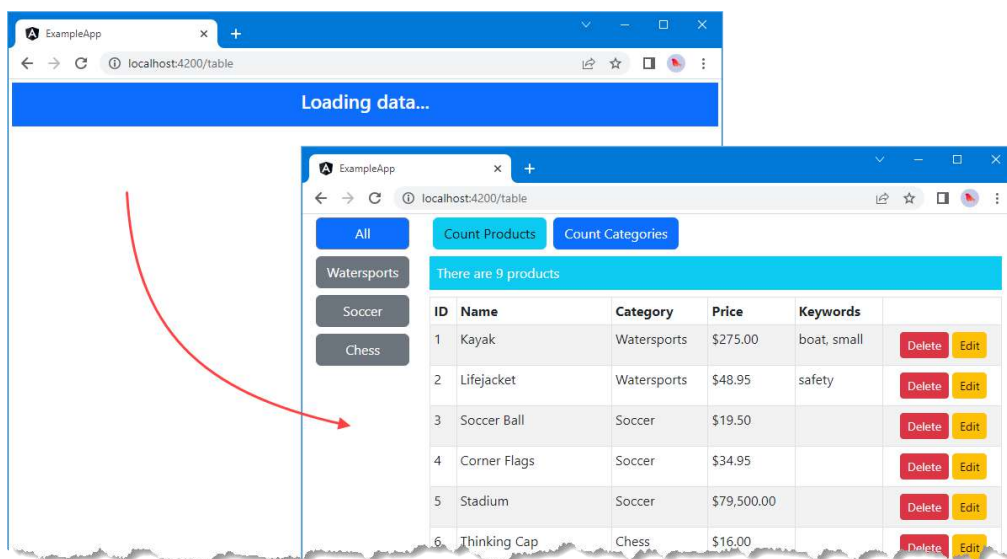


Figure 26.4. Displaying a loading message

USING A RESOLVER TO PREVENT URL ENTRY PROBLEMS

A resolver can be applied more broadly so that it protects multiple routes, which extends the loading message when the user navigates directly to a URL for a specific product, as shown in listing 26.6.

Listing 26.6. Applying the resolver to other routes in the app.routing.ts file in the src/app folder

```

import { Routes, RouterModule, mapToResolve } from "@angular/router";
import { TableComponent } from "../core/table.component";
import { FormComponent } from "../core/form.component";
import { NotFoundComponent } from "../core/notFound.component";

```

```

import { ProductCountComponent } from "../core/productCount.component";
import { CategoryCountComponent } from "../core/categoryCount.component";
import { ModelResolver } from "../model/model.resolver";

const childRoutes: Routes = [
  {
    path: "",
    children: [{ path: "products", component: ProductCountComponent },
               { path: "categories",
                 component: CategoryCountComponent },
               { path: "", component: ProductCountComponent }],
    resolve: { model: mapToResolve(ModelResolver) }
  }
];

const routes: Routes = [
  {
    path: "form/:mode/:id", component: FormComponent,
    resolve: { model: mapToResolve(ModelResolver) }
  },
  {
    path: "form/:mode", component: FormComponent,
    resolve: { model: mapToResolve(ModelResolver) }
  },
  { path: "does", redirectTo: "/form/create", pathMatch: "prefix" },
  { path: "table", component: TableComponent, children: childRoutes },
  { path: "table/:category", component: TableComponent,
    children: childRoutes },
  { path: "", redirectTo: "/table", pathMatch: "full" },
  { path: "**", component: NotFoundComponent }]

export const routing = RouterModule.forRoot(routes, {
  bindToComponentInputs: true
});

```

Applying the `ModelResolver` class to the routes that target `FormComponent` ensures that the user is shown the placeholder message while the data is loaded, as shown in figure 26.5. This deals with the direct navigation problem so that users can request a URL such as `http://localhost:4200/form/edit/2` and see product data without having to first request the application's main URL.

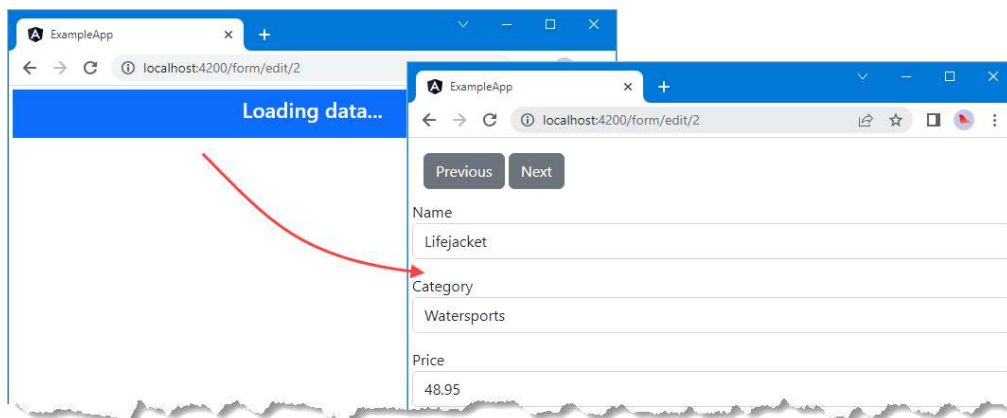


Figure 26.5. Expanding the use of a resolver

26.2.2 Preventing navigation with guards

Resolvers are used to delay navigation while the application performs some prerequisite work, such as loading data. The other guards that Angular provides are used to control whether navigation can occur at all, which can be useful when you want to alert the user to prevent potentially unwanted operations (such as abandoning data edits) or limit access to parts of the application unless the application is in a specific state, such as when a user has been authenticated.

Many uses for route guards introduce an additional interaction with the user, either to gain explicit approval to perform an operation or to obtain additional data, such as authentication credentials. For this chapter, I am going to handle this kind of interaction by extending the message service so that messages can require user input. In listing 26.7, I have added an optional `responses` constructor argument/property to the `Message` model class, which will allow messages to contain prompts to the user and callbacks that will be invoked when they are selected. The `responses` property is an array of TypeScript tuples, where the first value is the name of the response, which will be presented to the user, and the second value is the callback function, which will be passed the name as its argument.

Listing 26.7. Adding responses in the `message.model.ts` file in the `src/app/messages` folder

```
export class Message {
  constructor(public text: string,
    public error: boolean = false,
    public responses?: [string, (r: string) => void][]) { }
}
```

The only other change required to implement this feature is to present the response options to the user. Listing 26.8 adds `button` elements below the message text for each response. Clicking the buttons will invoke the callback function.

Listing 26.8. Presenting Responses in the `message.component.html` File in the `src/app/core` Folder

```
<div *ngIf="lastMessage()"
  class="bg-primary text-white p-2 text-center"
  [class.bg-danger]="lastMessage()?.error">
  <h4>{{lastMessage()?.text}}</h4>
</div>
<div class="text-center my-2">
  <button *ngFor="let resp of lastMessage()?.responses; let i = index"
    (click)="resp[1](resp[0])"
    class="btn btn-primary m-2" [class.btn-secondary]="i > 0">
    {{resp[0]}}
  </button>
</div>
```

PREVENTING ROUTE ACTIVATION

Guards can be used to prevent a route from being activated, helping to protect the application from entering an unwanted state or warning the user about the impact of an operation. To demonstrate, I am going to guard the `/form/create` URL to prevent the user from starting the process of creating a new product unless the user agrees to the application's terms and conditions.

Guards for route activation are functions that receive the same `ActivatedRouteSnapshot` and `RouterStateSnapshot` arguments as resolvers. Resolvers can also be defined as classes that define a `canActivate` method can be implemented to return three different result types, as described in table 26.4.

Table 26.4. The result types allowed by the `canActivate` method

Result Type	Description
<code>boolean</code>	This type of result is useful when performing synchronous checks to see whether the route can be activated. A <code>true</code> result will activate the route, and a result of <code>false</code> will not, effectively ignoring the navigation request.
<code>Observable<boolean></code>	This type of result is useful when performing asynchronous checks to see whether the route can be activated. Angular will wait until the <code>Observable</code> emits a value, which will be used to determine whether the route is activated. When using this kind of result, it is important to terminate the <code>Observable</code> by calling the <code>complete</code> method; otherwise, Angular will just keep waiting.
<code>Promise<boolean></code>	This type of result is useful when performing asynchronous checks to see whether the route can be activated. Angular will wait until the

<p>Promise is resolved and activate the route if it yields <code>true</code>. If the Promise yields <code>false</code>, then the route will not be activated, effectively ignoring the navigation request.</p>
--

To get started, I added a file called `terms.guard.ts` to the `src/app` folder and defined the class shown in listing 26.9.

NOTE Input properties cannot be used to receive routing parameters in route guards. Route data can only be accessed using the `ActivatedRouteSnapshot` object.

Listing 26.9. The contents of the `terms.guard.ts` file in the `src/app` folder

```
import { Injectable } from "@angular/core";
import {
  ActivatedRouteSnapshot, RouterStateSnapshot, Router
} from "@angular/router";
import { MessageService } from "../messages/message.service";
import { Message } from "../messages/message.model";

@Injectable()
export class TermsGuard {

  constructor(private messages: MessageService,
              private router: Router) { }

  canActivate(route: ActivatedRouteSnapshot,
              state: RouterStateSnapshot): Promise<boolean> | boolean {

    if (route.params["mode"] == "create") {

      return new Promise<boolean>(resolve => {
        let responses: [string, () => void][]
          = [
            ["Yes", () => resolve(true)],
            ["No", () => resolve(false)]
          ];
        this.messages.reportMessage(
          new Message("Do you accept the terms & conditions?",
                      false, responses));
      });
    } else {
      return true;
    }
  }
}
```

The `canActivate` method can return two different types of results. The first type is a `boolean`, which allows the guard to respond immediately for routes that it doesn't need to protect, which in this case is any that lacks a parameter called `mode` whose value is `create`. If the URL matched by the route doesn't contain this parameter, the `canActivate` method returns `true`, which tells Angular to activate the route. This is important because the `edit` and `create` features both rely on the same routes, and the guard should not interfere with edit operations.

The other type of result is a `Promise<boolean>`, which I have used instead of `Observable<true>` for variety. The `Promise` uses the modifications to the message service

to solicit a response from the user, confirming they accept the (unspecified) terms and conditions. There are two possible responses from the user. If the user clicks the Yes button, then the `Promise` will resolve and yield `true`, which tells Angular to activate the route, displaying the form that is used to create a new product. The `Promise` will resolve and yield `false` if the user clicks the No button, which tells Angular to ignore the navigation request.

Listing 26.10 registers the `TermsGuard` as a service so that it can be used in the application's routing configuration.

Listing 26.10. Registering the guard as a service 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 { ModelModule } from "../model/model.module";
import { CoreModule } from "../core/core.module";
import { TableComponent } from "../core/table.component";
import { FormComponent } from "../core/form.component";
import { MessageModule } from "../messages/message.module";
import { MessageComponent } from "../messages/message.component";
import { AppComponent } from './app.component';
import { routing } from './app.routing';
import { TermsGuard } from "../terms.guard"

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

Listing 26.11 applies the guard to the routing configuration. Activation guards are applied to a route using the `canActivate` property, which is assigned an array of guard services. The `canActivate` method of all the guards must return `true` (or return an `Observable` or `Promise` that eventually yields `true`) before Angular will activate the route.

Listing 26.11. Applying the guard to a route in the `app.routing.ts` file in the `src/app` folder

```
import { Routes, RouterModule, mapToResolve, mapToCanActivate }
  from "@angular/router";
import { TableComponent } from "../core/table.component";
import { FormComponent } from "../core/form.component";
import { NotFoundComponent } from "../core/notFound.component";
import { ProductCountComponent } from "../core/productCount.component";
import { CategoryCountComponent } from "../core/categoryCount.component";
import { ModelResolver } from "../model/model.resolver";
import { TermsGuard } from "../terms.guard";

const childRoutes: Routes = [
  { path: "",
```



```

        children: [{ path: "products", component: ProductCountComponent },
                    { path: "categories",
                      component: CategoryCountComponent },
                    { path: "", component: ProductCountComponent }],
        resolve: { model: mapToResolve(ModelResolver) }
    }
];

const routes: Routes = [
    {
        path: "form/:mode/:id", component: FormComponent,
        resolve: { model: mapToResolve(ModelResolver) }
    },
    {
        path: "form/:mode", component: FormComponent,
        resolve: { model: ModelResolver },
        canActivate: mapToCanActivate([TermsGuard])
    },
    { path: "does", redirectTo: "/form/create", pathMatch: "prefix" },
    { path: "table", component: TableComponent, children: childRoutes },
    { path: "table/:category", component: TableComponent,
      children: childRoutes },
    { path: "", redirectTo: "/table", pathMatch: "full" },
    { path: "**", component: NotFoundComponent }]

export const routing = RouterModule.forRoot(routes, {
    bindToComponentInputs: true
});

```

The guard class is applied using the `mapToCanActivate` function, which accepts an array of classes that define the `canActivate` method.

The effect of creating and applying the activation guard is that the user is prompted when clicking the Create New Product button, as shown in figure 26.6. If they respond by clicking the Yes button, then the navigation request will be completed, and Angular will activate the route that selects the form component, which will allow a new product to be created. If the user clicks the No button, then the navigation request will be canceled. In both cases, the routing system emits an event that is received by the component that displays the messages to the user, which clears its display and ensures that the user doesn't see stale messages.

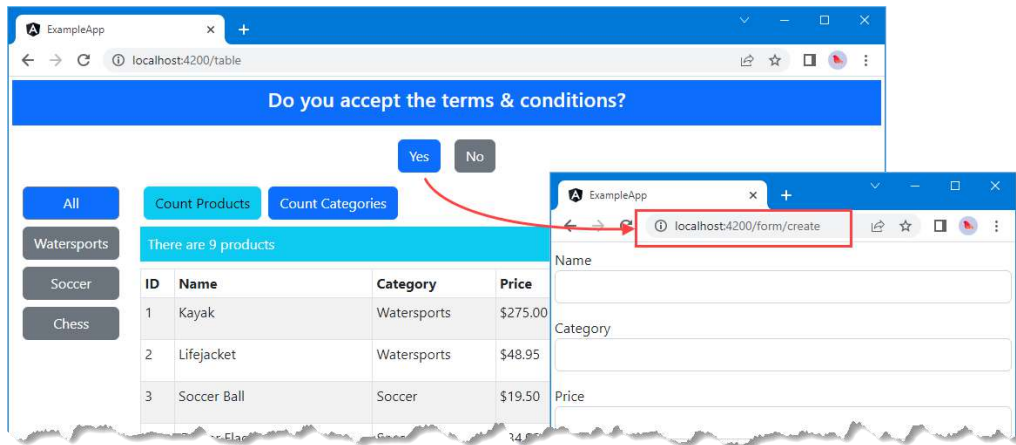


Figure 26.6. Guarding route activation

CONSOLIDATING CHILD ROUTE GUARDS

If you have a set of child routes, you can guard against their activation using a child route guard, which is a function that receives the same `ActivatedRouteSnapshot` and `RouterStateSnapshot` objects as the other guards and can return the set of result types described in table 26.4. Child route guards can also be classes that define a `canActivateChild` method, which simplifies the use of dependency injection and is the approach that I prefer.

The guard is applied to the parent route in the application's configuration, and the `canActivateChild` method is called whenever any of the child routes are about to be activated.

This guard in this example is more readily dealt with by changing the configuration before implementing the `canActivateChild` method, as shown in listing 26.12.

Listing 26.12. Guarding child routes in the `app.routing.ts` file in the `src/app` folder

```
import { Routes, RouterModule, mapToResolve, mapToCanActivate,
  mapToCanActivateChild } from "@angular/router";
import { TableComponent } from "../core/table.component";
import { FormComponent } from "../core/form.component";
import { NotFoundComponent } from "../core/notFound.component";
import { ProductCountComponent } from "../core/productCount.component";
import { CategoryCountComponent } from "../core/categoryCount.component";
import { ModelResolver } from "../model/model.resolver";
import { TermsGuard } from "../terms.guard";

const childRoutes: Routes = [
  {
    path: "",
    canActivateChild: mapToCanActivateChild([TermsGuard]),
    children: [{ path: "products", component: ProductCountComponent },
      { path: "categories",
        component: CategoryCountComponent },
    ],
  },
];
```

```

        { path: "", component: ProductCountComponent }],
        resolve: { model: mapToResolve(ModelResolver) }
    }
];

const routes: Routes = [
    {
        path: "form/:mode/:id", component: FormComponent,
        resolve: { model: mapToResolve(ModelResolver) }
    },
    {
        path: "form/:mode", component: FormComponent,
        resolve: { model: ModelResolver },
        canActivate: mapToCanActivate([TermsGuard])
    },
    { path: "does", redirectTo: "/form/create", pathMatch: "prefix" },
    { path: "table", component: TableComponent, children: childRoutes },
    { path: "table/:category", component: TableComponent,
        children: childRoutes },
    { path: "", redirectTo: "/table", pathMatch: "full" },
    { path: "**", component: NotFoundComponent }]

export const routing = RouterModule.forRoot(routes, {
    bindToComponentInputs: true
});

```

Classes that define the `canActivateChild` method are applied as guards using the `mapToCanActivateChild` function, which generates a result that can be assigned to the route's `canActivateChild` property.

This guard's `canActivateChild` method will be called before Angular activates any of the route's children. Listing 26.13 adds the `canActivateChild` method to the guard class from the previous section.

Listing 26.13. Implementing child route guards in the `terms.guard.ts` file in the `src/app` folder

```

import { Injectable } from "@angular/core";
import {
    ActivatedRouteSnapshot, RouterStateSnapshot, Router
} from "@angular/router";
import { MessageService } from "../messages/message.service";
import { Message } from "../messages/message.model";

@Injectable()
export class TermsGuard {

    constructor(private messages: MessageService,
                private router: Router) { }

    canActivate(route: ActivatedRouteSnapshot,
                state: RouterStateSnapshot): Promise<boolean> | boolean {

        if (route.params["mode"] == "create") {

            return new Promise<boolean>(resolve => {
                let responses: [string, () => void][]

```

```

        = [["Yes", () => resolve(true)],
          ["No", () => resolve(false)]];
        this.messages.reportMessage(
          new Message("Do you accept the terms & conditions?",
            false, responses));
      });
    } else {
      return true;
    }
  }
}

canActivateChild(route: ActivatedRouteSnapshot,
  state: RouterStateSnapshot): Promise<boolean> | boolean {

  if (route.url.length > 0
    && route.url[route.url.length - 1].path == "categories") {

    return new Promise<boolean>((resolve, reject) => {
      let responses: [string, (arg: string) => void][] = [
        ["Yes", () => resolve(true)],
        ["No ", () => resolve(false)]
      ];

      this.messages.reportMessage(
        new Message("Do you want to see the categories?",
          false, responses));
    });
  } else {
    return true;
  }
}
}

```

The guard only protects the `categories` child route and will return `true` immediately for any other route. The guard prompts the user using the message service but does something different if the user clicks the No button. In addition to rejecting the active route, the guard navigates to a different URL using the `Router` service, which is received as a constructor argument. This is a common pattern for authentication when the user is redirected to a component that will solicit security credentials if a restricted operation is attempted. The example is simpler in this case, and the guard navigates to a sibling route that shows a different component.

To see the effect of the guard, click the Count Categories button, as shown in figure 26.7. Responding to the prompt by clicking the Yes button will show the `CategoryCountComponent`, which displays the number of categories in the table. Clicking No will reject the active route and navigate to a route that displays the `ProductCountComponent` instead.

NOTE Guards are applied only when the active route changes. So, for example, if you click the Count Categories button when the `/table` URL is active, then you will see the prompt, and clicking Yes will change the active route. But nothing will happen if you click the Count

Categories button again because Angular doesn't trigger a route change when the target route and the active route are the same.

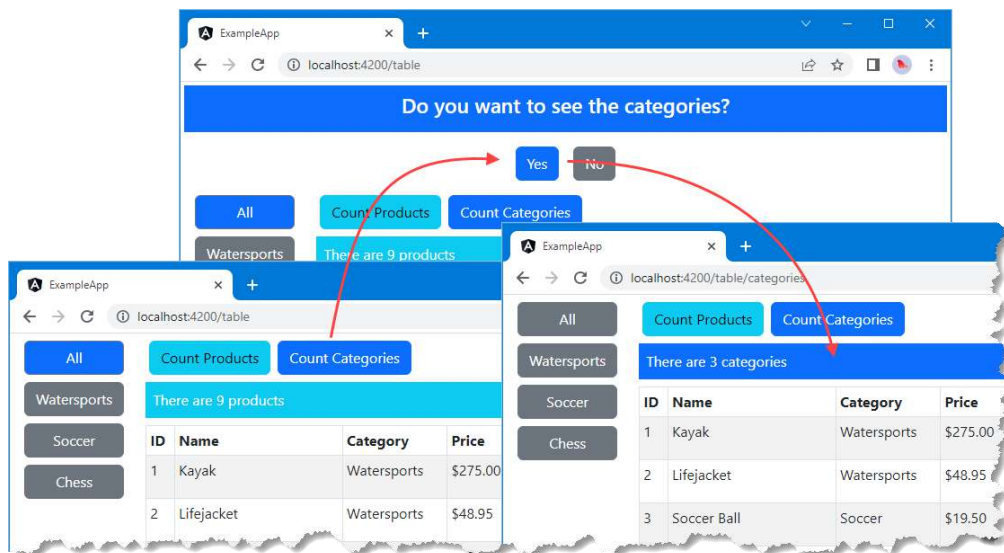


Figure 26.7. Guarding child routes

PREVENTING ROUTE DEACTIVATION

When you start working with routes, you will tend to focus on the way that routes are activated to respond to navigation and present new content to the user. But equally important is route *deactivation*, which occurs when the application navigates away from a route.

The most common use for deactivation guards is to prevent the user from navigating when there are unsaved edits to data. In this section, I will create a guard that warns the user when they are about to abandon unsaved changes when editing a product. In preparation, a change is required in the template so that the Cancel button doesn't invoke the form's reset event handler, as shown in listing 26.14.

Listing 26.14. Disabling form reset in the form.component.html file in the src/app/core folder

```
...
</div>

<form [formGroup]="productForm" #form="ngForm" (ngSubmit)="submitForm()">
  <div class="form-group">
    <label>Name</label>
    <input class="form-control" formControlName="name" />
    <ul class="text-danger list-unstyled mt-1">
      <li *validationErrors="productForm; control:'name'; let err">
        {{ err }}
      </li>
    </ul>
  </div>
</form>
```

```

    </li>
  </ul>

```

```

...

```

To create the guard, I added a file called `unsaved.guard.ts` in the `src/app/core` folder and defined the class shown in listing 26.15.

Listing 26.15. The contents of the `unsaved.guard.ts` file in the `src/app/core` folder

```

import { Injectable } from "@angular/core";
import { ActivatedRouteSnapshot, RouterStateSnapshot, Router }
  from "@angular/router";
import { MessageService } from "../messages/message.service";
import { Message } from "../messages/message.model";
import { FormComponent } from "../form.component";

@Injectable()
export class UnsavedGuard {

  constructor(private messages: MessageService,
              private router: Router) { }

  canActivate(component: FormComponent, route: ActivatedRouteSnapshot,
              state: RouterStateSnapshot): Promise<boolean> | boolean {
    if (component.editing && component.productForm.dirty) {
      return new Promise(resolve => {
        let responses: [string, (r: string) => void][] = [
          ["Yes", () => resolve(true)],
          ["No", () => {
            this.router.navigateByUrl(this.router.url);
            resolve(false);
          }]
        ];
      });
      this.messages.reportMessage(
        new Message("Discard Changes?", true, responses));
    }
    return true;
  }
}

```

Deactivation guards are functions that receive three arguments: the component that is about to be deactivated and the `ActivatedRouteSnapshot` and `RouterStateSnapshot` objects. This guard checks to see whether there are unsaved edits in the component and prompts the user if there are. This guard uses a `Promise<true>` to tell Angular whether it should activate the route based on the response selected by the user. Deactivation guards can also be classes that define a `canDeactivate` method that accepts the same three arguments, which is the approach I have taken in listing 26.15.

The next step is to register the guard as a service in the module that contains it, as shown in listing 26.16.

Listing 26.16. Registering the guard in the `core.module.ts` file in the `src/app/core` folder

```

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

```

```

import { FormsModule, NG_VALIDATORS, ReactiveFormsModule }
  from "@angular/forms";
import { ModelModule } from "../model/model.module";
import { TableComponent } from "../table.component";
import { FormComponent } from "../form.component";
//import { SharedState } from "../sharedState.service";
import { ValidationHelper } from "../validationHelper.pipe";
import { ValidationErrorsDirective } from "../validationErrors.directive";
import { HiLowValidatorDirective } from "../validation/hilow";
import { RouterModule } from "@angular/router";
import { ProductCountComponent } from "../productCount.component";
import { CategoryCountComponent } from "../categoryCount.component";
import { NotFoundComponent } from "../notFound.component";
import { UnsavedGuard } from "../unsaved.guard";

@NgModule({
  imports: [BrowserModule, FormsModule, ModelModule,
    ReactiveFormsModule, RouterModule],
  declarations: [TableComponent, FormComponent, ValidationHelper,
    ValidationErrorsDirective, HiLowValidatorDirective,
    ProductCountComponent, CategoryCountComponent,
    NotFoundComponent],
  exports: [ModelModule, TableComponent, FormComponent],
  providers: [UnsavedGuard]
})
export class CoreModule { }

```

Finally, listing 26.17 applies the guard to the application's routing configuration. Deactivation guards are applied to routes using the `canDeactivate` property and class-based guards are applied using the

Listing 26.17. Applying the guard in the `app.routing.ts` file in the `src/app` folder

```

import { Routes, RouterModule, mapToResolve, mapToCanActivate,
  mapToCanActivateChild, mapToCanDeactivate } from "@angular/router";
import { TableComponent } from "../core/table.component";
import { FormComponent } from "../core/form.component";
import { NotFoundComponent } from "../core/notFound.component";
import { ProductCountComponent } from "../core/productCount.component";
import { CategoryCountComponent } from "../core/categoryCount.component";
import { ModelResolver } from "../model/model.resolver";
import { TermsGuard } from "../terms.guard";
import { UnsavedGuard } from "../core/unsaved.guard";

const childRoutes: Routes = [
  {
    path: "",
    canActivateChild: mapToCanActivateChild([TermsGuard]),
    children: [{ path: "products", component: ProductCountComponent },
      { path: "categories",
        component: CategoryCountComponent },
      { path: "", component: ProductCountComponent }],
    resolve: { model: mapToResolve(ModelResolver) }
  }
];

const routes: Routes = [
  {

```

```

    path: "form/:mode/:id", component: FormComponent,
    resolve: { model: mapToResolve(ModelResolver) },
    canActivate: mapToCanDeactivate([UnsavedGuard])
  },
  {
    path: "form/:mode", component: FormComponent,
    resolve: { model: ModelResolver },
    canActivate: mapToCanActivate([TermsGuard])
  },
  { path: "does", redirectTo: "/form/create", pathMatch: "prefix" },
  { path: "table", component: TableComponent, children: childRoutes },
  { path: "table/:category", component: TableComponent,
    children: childRoutes },
  { path: "", redirectTo: "/table", pathMatch: "full" },
  { path: "**", component: NotFoundComponent }]

export const routing = RouterModule.forRoot(routes, {
  bindToComponentInputs: true
});

```

To see the effect of the guard, click one of the Edit buttons in the table; edit the data in one of the text fields; then click the Cancel, Next, or Previous button. The guard will prompt you before allowing Angular to activate the route you selected, as shown in figure 26.8.

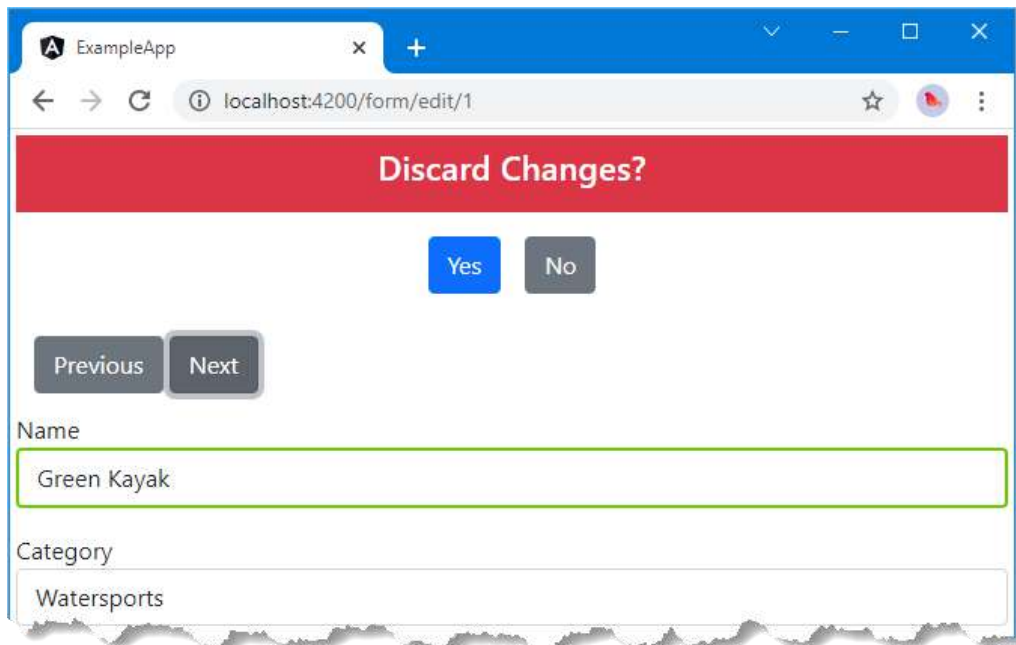


Figure 26.8. Guarding route deactivation

26.3 Summary

In this chapter, I finished describing the Angular URL routing features and explaining how to guard routes to control when a route is activated, how to load modules only when they are needed, and how to use multiple outlet elements to display components to the user.

- Route resolvers delay activation of a route until a condition is satisfied, such as loading data.
- Placeholder content can be displayed to the user until a resolver's condition is satisfied.
- Route activation and deactivation can be controlled with route guards.

In the next chapter, I will show you how to optimize application delivery and startup.

27

Optimizing application delivery

This chapter covers

- Reducing the size of the application using dynamically loaded modules
- Using server-side rendering to create an HTML representation of the application
- Using rehydration to transition from server-rendered HTML to browser-generated content
- Prerendering the application to create a static HTML representation of the application

Complex Angular applications can require large JavaScript files, which can take a long time to download over a slow network connection. In this chapter, I describe the features Angular provides for optimizing application delivery to minimize the amount of time before the user can interact with the application.

Table 27.1. Putting optimized application delivery in context

Question	Answer
What is it?	Optimization reduces the amount of time that the user sees an empty browser window while the application loads.
Why is it useful?	Slow startup frustrates users and undermines SEO efforts.
How is it used?	There are several related features. Dynamically loaded modules exclude code from the initial download and only load it when it is needed. Server-side rendering executes the application at the server so the user sees an HTML representation of the application while the browser loads the JavaScript files. Prerendering creates a static HTML version of the application.

Are there any pitfalls or limitations?	These features must be applied carefully. Not all application features are suitable for dynamic modules and not all Angular features are supported by server-side rendering and prerendering.
Are there any alternatives?	There are third-party packages that provide similar features but they are not as well integrated as the Angular features.

Table 27.2 summarizes the chapter.

Table 27.2. Chapter summary

Problem	Solution	Listing
Reduce the size of the initial JavaScript download	Create modules that are loaded dynamically	1-10
Minimize the amount of time the user sees an empty browser window	Use server-side rendering to execute the application on the server and generate HTML documents the browser can display	11-24
Create an HTML representation without requiring server resources	Prerender the application	25-28

27.1 Preparing the example project

For this chapter, I will continue using the `exampleApp` project that was created in chapter 20 and has been modified in each subsequent chapter. To prepare for this chapter, listing 27.1 disables the delay applied to loading data from the RESTful web service.

Listing 27.1. Disabling the delay in the `rest.datasource.ts` file in the `src/app/model` folder

```
import { Injectable, Signal } from "@angular/core";
import { HttpClient, HttpHeaders } from "@angular/common/http";
import { Product } from "../product.model";
import { Observable, catchError, delay } from "rxjs";

export const REST_URL = `http://${location.hostname}:3500/products`;

@Injectable()
export class RestDataSource {

  constructor(private http: HttpClient) { }

  getData(): Observable<Product[]> {
    return this.sendRequest<Product[]>("GET", REST_URL);
    // .pipe(delay(5000));
  }

  // ...other methods omitted for brevity...
}
```

TIP You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/manningbooks/pro-angular-16>. See chapter 1 for how to get help if you have problems running the examples.

Open a new command prompt, navigate to the `exampleApp` folder, and run the following command to start the server that provides the RESTful web server:

```
npm run json
```

Open a separate command prompt, navigate to the `exampleApp` folder, and run the following command to start the Angular development tools:

```
ng serve
```

Open a new browser window and navigate to `http://localhost:4200` to see the content shown in figure 27.1.

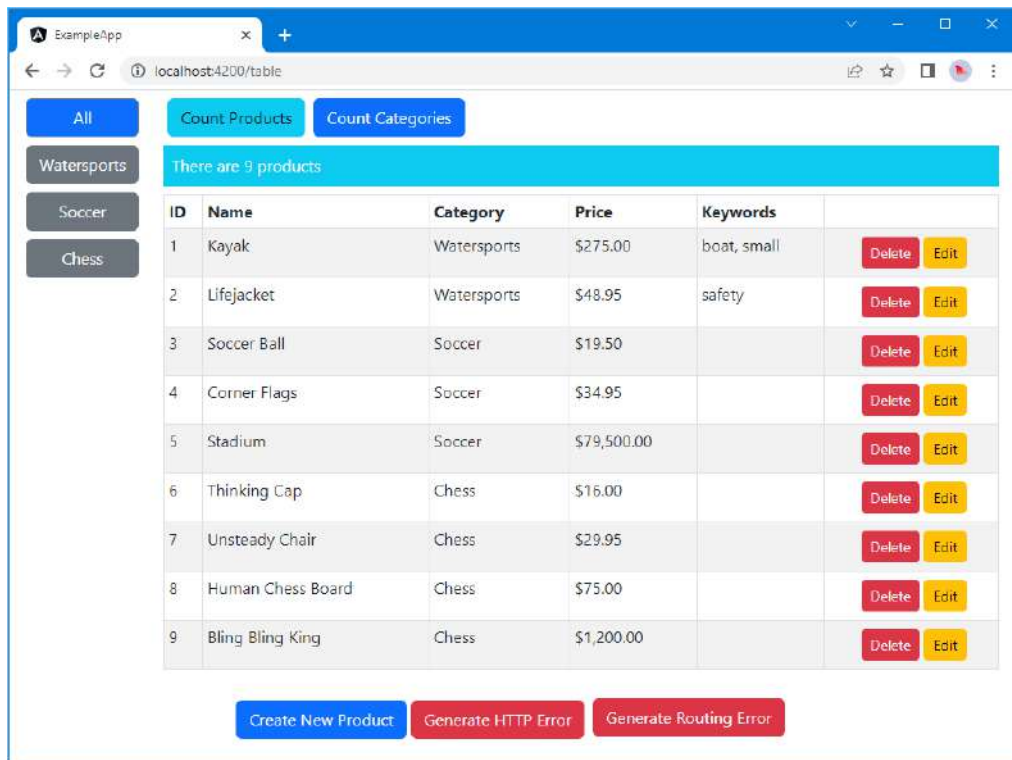


Figure 27.1. Running the example application

27.2 Understanding the delivery problem

It can take time for a browser to load and execute the JavaScript files required for an application, even when non-essential modules are loaded dynamically. Not all clients can rely

on the fast networks that are common in development environments, and clients with slow networks will end up looking at an empty browser window until the application loads.

Most modern browsers include a simulator for bandwidth throttling so you can get a sense of how long it will take a client to load an application over a slow connection. For Chrome, this feature is on the Network tab of the F12 developer tools, and I selected the Fast 3G present, as shown in figure 27.2.

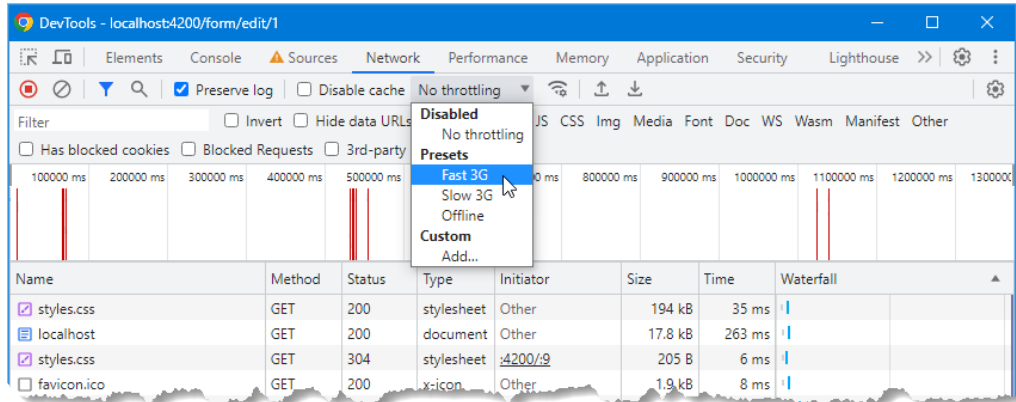


Figure 27.2. Applying rate limiting to test rehydration

Hold down the browser's reload button with the F12 developer tools window open to select the "Empty cache and hard reload" option, as shown in figure 27.3, which will ensure the browser loads all of the JavaScript files from the server, even if they were previously cached.

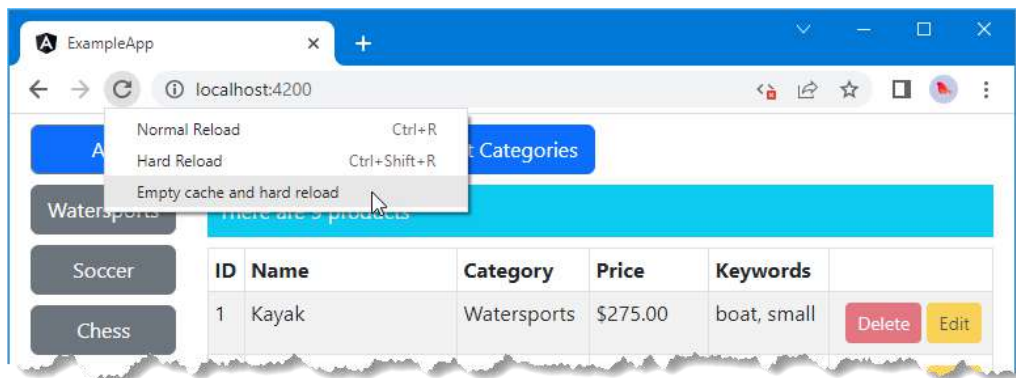


Figure 27.3. Reloading the application

The Network tab in the F12 tools window will show you the progress of the file downloads and report the total time when the process is complete. On my system, the process takes 21 seconds, during which time the user will see a completely blank screen.

This time is just a baseline for comparison. During development, Angular doesn't optimize the size of the JavaScript files and they are much larger than the files in a production build.

Once you have established the baseline, you can disable the bandwidth throttling, which isn't used again until later in the chapter.

27.3 Loading feature modules dynamically

Angular supports loading feature modules only when they are required, known as *dynamic loading* or *lazy loading*. This can be useful for functionality that is unlikely to be required by all users and which would otherwise slow down the loading process. In the sections that follow, I create a simple feature module and demonstrate how to configure the application so that Angular will load the module only when the application navigates to a specific URL.

Loading modules dynamically is a trade-off. The application will be smaller and faster to download for most users, improving their overall experience. But users who require the dynamically loaded features will have to wait while Angular gets the module and its dependencies. The effect can be jarring because the user has no idea that some features have been loaded and others have not. When you create dynamically loaded modules, you are balancing improving the experience for some users against making it worse for others. Consider how your users fall into these groups and take care not to degrade the experience of your most valuable and important customers.

27.3.1 Creating a simple feature module

Dynamically loaded modules must contain only functionality that not all users require. I can't use the existing modules because they provide the core functionality for the application, which means that I need a new module for this part of the chapter. I started by creating a folder called `ondemand` in the `src/app` folder. To give the new module a component, I added a file called `ondemand.component.ts` in the `example/app/ondemand` folder and added the code shown in listing 27.2.

CAUTION It is important not to create dependencies between other parts of the application and the classes in the dynamically loaded module so that the JavaScript module loader doesn't try to load the module before it is required.

Listing 27.2. The contents of the `ondemand.component.ts` file in the `src/app/ondemand` folder

```
import { Component } from "@angular/core";

@Component({
  selector: "ondemand",
  templateUrl: "ondemand.component.html"
})
export class OndemandComponent { }
```

To provide the component with a template, I added a file called `ondemand.component.html` and added the markup shown in listing 27.3.

Listing 27.3. The `ondemand.component.html` file in the `src/app/ondemand` folder

```
<div class="bg-primary text-white p-2">
  This is the ondemand component
</div>
<button class="btn btn-primary m-2" routerLink="/">Back</button>
```

The template contains a message that will make it obvious when the component is selected and that contains a `button` element that will navigate back to the application's root URL when clicked.

To define the module, I added a file called `ondemand.module.ts` and added the code shown in listing 27.4.

Listing 27.4. The contents of the `ondemand.module.ts` file in the `src/app/ondemand` folder

```
import { NgModule } from "@angular/core";
import { CommonModule } from "@angular/common";
import { OndemandComponent } from "../ondemand.component";

@NgModule({
  imports: [CommonModule],
  declarations: [OndemandComponent],
  exports: [OndemandComponent]
})
export class OndemandModule { }
```

The module imports the `CommonModule` functionality, which is used instead of the browser-specific `BrowserModule` to access the built-in directives in feature modules that are loaded on demand.

27.3.2 Loading the module dynamically

There are two steps to set up dynamically loading a module. The first is to set up a routing configuration inside the feature module to provide the rules that will allow Angular to select a component when the module is loaded. Listing 27.5 adds a single route to the feature module.

Listing 27.5. Defining routes in the `ondemand.module.ts` file in the `src/app/ondemand` folder

```
import { NgModule } from "@angular/core";
import { CommonModule } from "@angular/common";
import { OndemandComponent } from "../ondemand.component";
import { RouterModule } from "@angular/router";

let routing = RouterModule.forChild([
  { path: "", component: OndemandComponent }
]);

@NgModule({
  imports: [CommonModule, routing],
```

```

    declarations: [OndemandComponent],
    exports: [OndemandComponent]
  })
  export class OndemandModule { }

```

Routes in dynamically loaded modules are defined using the same properties as in the main part of the application and can use all the same features, including child components, guards, and redirections. The route defined in the listing matches the empty path and selects the `OndemandComponent` for display.

One important difference is the method used to generate the module that contains the routing information, as follows:

```

...
let routing = RouterModule.forChild([
  { path: "", component: OndemandComponent }
]);
...

```

When I created the application-wide routing configuration, I used the `RouterModule.forRoot` method. This is the method that is used to set up the routes in the root module of the application. When creating dynamically loaded modules, the `RouterModule.forChild` method must be used; this method creates a routing configuration that is merged into the overall routing system when the module is loaded.

CREATING A ROUTE TO DYNAMICALLY LOAD A MODULE

The second step to set up a dynamically loaded module is to create a route in the main part of the application that provides Angular with the module's location, as shown in listing 27.6.

Listing 27.6. Creating an on-demand route in the `app.routing.ts` file in the `src/app` folder

```

import { Routes, RouterModule, mapToResolve, mapToCanActivate,
  mapToCanActivateChild, mapToCanDeactivate } from "@angular/router";
import { TableComponent } from "../core/table.component";
import { FormComponent } from "../core/form.component";
import { NotFoundComponent } from "../core/notFound.component";
import { ProductCountComponent } from "../core/productCount.component";
import { CategoryCountComponent } from "../core/categoryCount.component";
import { ModelResolver } from "../model/model.resolver";
import { TermsGuard } from "../terms.guard";
import { UnsavedGuard } from "../core/unsaved.guard";

const childRoutes: Routes = [
  {
    path: "",
    canActivateChild: mapToCanActivateChild([TermsGuard]),
    children: [{ path: "products", component: ProductCountComponent },
      { path: "categories",
        component: CategoryCountComponent,
        { path: "", component: ProductCountComponent } }],
    resolve: { model: mapToResolve(ModelResolver) }
  }
];

const routes: Routes = [
  {
    path: "ondemand",
    loadChildren: () => import("../ondemand/ondemand.module")
  }
];

```



```

        .then(m => m.OndemandModule)
    },
    {
      path: "form/:mode/:id", component: FormComponent,
      resolve: { model: mapToResolve(ModelResolver) },
      canDeactivate: mapToCanDeactivate([UnsavedGuard])
    },
    {
      path: "form/:mode", component: FormComponent,
      resolve: { model: ModelResolver },
      canActivate: mapToCanActivate([TermsGuard])
    },
    { path: "does", redirectTo: "/form/create", pathMatch: "prefix" },
    { path: "table", component: TableComponent, children: childRoutes },
    { path: "table/:category", component: TableComponent,
      children: childRoutes },
    { path: "", redirectTo: "/table", pathMatch: "full" },
    { path: "**", component: NotFoundComponent } ]

export const routing = RouterModule.forRoot(routes, {
  bindToComponentInputs: true
});

```

The `loadChildren` property is used to provide Angular with details of how the module should be loaded. The property is assigned a function that invokes `import`, passing in the path to the module. The result is a `Promise` whose `then` method is used to select the module after it has been imported. The function in the listing tells Angular to load the `OndemandModule` class from the `ondemand/ondemand.module` file.

USING A DYNAMICALLY LOADED MODULE

All that remains is to add support for navigating to the URL that will activate the route for the on-demand module, as shown in listing 27.7, which adds a button to the template for the table component.

Listing 27.7. Adding navigation in the `table.component.html` file in the `src/app/core` folder

```

...
<div class="p-2 text-center">
  <button class="btn btn-primary mt-1" routerLink="/form/create">
    Create New Product
  </button>
  <button class="btn btn-danger mt-1 mx-1" (click)="deleteProduct(-1)">
    Generate HTTP Error
  </button>
  <button class="btn btn-danger m-1" routerLink="/does/not/exist">
    Generate Routing Error
  </button>
  <button class="btn btn-danger" routerLink="/ondemand">
    Load Module
  </button>
</div>
...

```

No special measures are required to target a route that loads a module, and the Load Module button in the listing uses the standard `routerLink` attribute to navigate to the URL specified by the route added in listing 27.6.

Click the Load Module button, and you will see an HTTP request in the browser's F12 developer tools window for the new module. When the button is clicked, Angular uses the routing configuration to load the module, inspect its routing configuration, and select the component that will be displayed to the user, as shown in figure 27.4. The HTTP request is made the first time the module is required, and subsequent navigation does not require additional requests.

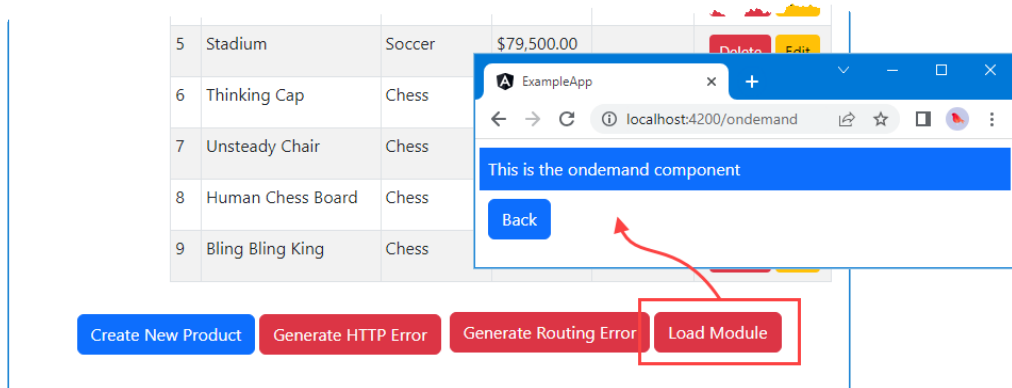


Figure 27.4. Loading a module dynamically

27.3.3 Guarding dynamic modules

You can guard against dynamically loading modules to ensure that they are loaded only when the application is in a specific state or when the user has explicitly agreed to wait while Angular does the loading (this latter option is typically used only for administration functions, where the user can be expected to have some understanding of how the application is structured).

The guard for the module must be defined in the main part of the application, so I added a file called `load.guard.ts` in the `src/app` folder and defined the class shown in listing 27.8.

Listing 27.8. The contents of the `load.guard.ts` file in the `src/app` folder

```
import { Injectable } from "@angular/core";
import { ActivatedRouteSnapshot, Router, RouterStateSnapshot }
  from "@angular/router";
import { MessageService } from "../messages/message.service";
import { Message } from "../messages/message.model";

@Injectable()
export class LoadGuard {
  private loaded: boolean = false;
```

```

    constructor(private messages: MessageService,
                 private router: Router) { }

    canActivate(route: ActivatedRouteSnapshot,
                state: RouterStateSnapshot): Promise<boolean> | boolean {

        return this.loaded || new Promise<boolean>((resolve, reject) => {
            let responses: [string, (r: string) => void] [] = [
                ["Yes", () => {
                    this.loaded = true;
                    resolve(true);
                }],
                ["No", () => {
                    this.router.navigateByUrl(this.router.url);
                    resolve(false);
                }
            ];

            this.messages.reportMessage(
                new Message("Do you want to load the module?",
                           false, responses));
        });
    }
}

```

Dynamic modules are guarded in the same way as for regular routes and listing 27.8 defines a class with a `canLoad` method. The guard is required only when the URL that loads the module is first activated, so it defines a `loaded` property that is set to `true` when the module has been loaded so that subsequent requests are immediately approved. Otherwise, this guard follows the same pattern as earlier examples and returns a `Promise` that will be resolved when the user clicks one of the buttons displayed by the message service. Listing 27.9 registers the guard as a service in the root module.

Listing 27.9. Registering the guard as a service 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 { ModelModule } from './model/model.module';
import { CoreModule } from './core/core.module';
import { TableComponent } from './core/table.component';
import { FormComponent } from './core/form.component';
import { MessageModule } from './messages/message.module';
import { MessageComponent } from './messages/message.component';
import { AppComponent } from './app.component';
import { routing } from './app.routing';
import { TermsGuard } from './terms.guard'
import { LoadGuard } from './load.guard';

@NgModule({
    declarations: [AppComponent],

```

```

    imports: [BrowserModule, ModelModule, CoreModule, MessageModule,
              routing],
    providers: [TermsGuard, LoadGuard],
    bootstrap: [AppComponent]
  })
  export class AppModule { }

```

APPLYING A DYNAMIC LOADING GUARD

Listing 27.10 applies the `LoadGuard` class to the route that dynamically loads the module.

Listing 27.10. Guarding the route in the `app.routing.ts` file in the `src/app` folder

```

import { Routes, RouterModule, mapToResolve, mapToCanActivate,
        mapToCanActivateChild, mapToCanDeactivate } from "@angular/router";
import { TableComponent } from "../core/table.component";
import { FormComponent } from "../core/form.component";
import { NotFoundComponent } from "../core/notFound.component";
import { ProductCountComponent } from "../core/productCount.component";
import { CategoryCountComponent } from "../core/categoryCount.component";
import { ModelResolver } from "../model/model.resolver";
import { TermsGuard } from "../terms.guard";
import { UnsavedGuard } from "../core/unsaved.guard";
import { LoadGuard } from "../load.guard";

const childRoutes: Routes = [
  {
    path: "",
    canActivateChild: mapToCanActivateChild([TermsGuard]),
    children: [{ path: "products", component: ProductCountComponent },
              { path: "categories",
                component: CategoryCountComponent,
                { path: "", component: ProductCountComponent } }],
    resolve: { model: mapToResolve(ModelResolver) }
  }
];

const routes: Routes = [
  {
    path: "ondemand",
    canActivate: mapToCanActivate([LoadGuard]),
    loadChildren: () => import("../ondemand/ondemand.module")
      .then(m => m.OndemandModule)
  },
  {
    path: "form/:mode/:id", component: FormComponent,
    resolve: { model: mapToResolve(ModelResolver) },
    canDeactivate: mapToCanDeactivate([UnsavedGuard])
  },
  {
    path: "form/:mode", component: FormComponent,
    resolve: { model: ModelResolver },
    canActivate: mapToCanActivate([TermsGuard])
  },
  { path: "does", redirectTo: "/form/create", pathMatch: "prefix" },
  { path: "table", component: TableComponent, children: childRoutes },
  { path: "table/:category", component: TableComponent,
    children: childRoutes },
  { path: "", redirectTo: "/table", pathMatch: "full" },

```

```
{ path: "**", component: NotFoundComponent }]
```

```
export const routing = RouterModule.forRoot(routes, {
  bindToComponentInputs: true
});
```

The result is that the user is prompted to determine whether they want to load the module the first time that Angular tries to activate the route, as shown in figure 27.5.

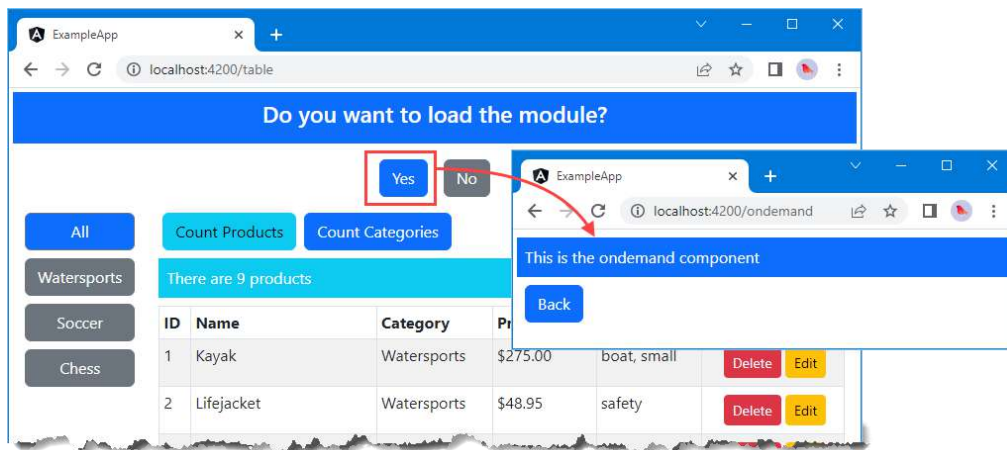


Figure 27.5. Guarding dynamic loading

27.4 Using server-side rendering

Server-side rendering (SSR) uses a JavaScript runtime at the server to execute the application and sends the browser static HTML, which ensures that the application is displayed in the browser. The HTML document includes `script` elements that tell the browser to download and execute the JavaScript files, and a feature called *hydration* is used to seamlessly transition from the HTML content generated by the JavaScript running on the server to content generated by the JavaScript running in the browser.

The main limitation of SSR is that it can require considerable server-side resources to execute the application on behalf of the clients. It also requires a JavaScript runtime at the server to execute the application, which may not be available on all production platforms.

SSR doesn't support all Angular application features and requires some preparation, as the following sections demonstrate. But, even so, SSR can be a powerful tool and can ensure that clients have something useful to work with as soon as possible.

27.4.1 Installing the server-side rendering packages

Stop the `ng serve` command and run the command shown in listing 27.11 in the `exampleApp` folder to update to the latest version of Angular 16, which will avoid conflicts in package dependencies that would otherwise arise.

Listing 27.11. Updating the Angular packages

```
ng update @angular/cli@^16 @angular/core@^16
```

Next, run the command shown in listing 27.12 in the `exampleApp` package to install the Angular Universal package, which contains the server-side rendering features.

Listing 27.12. Installing the package

```
ng add @nguniversal/express-engine@^16
```

The SSR package used in listing 27.12 uses the popular JavaScript Express HTTP server to execute the Angular application. As part of the package configuration process, additional files are added to the project to support SSR, as noted in the console output:

```
...
CREATE src/main.server.ts (60 bytes)
CREATE src/app/app.server.module.ts (318 bytes)
CREATE tsconfig.server.json (272 bytes)
CREATE server.ts (2034 bytes)
UPDATE package.json (1572 bytes)
UPDATE angular.json (5227 bytes)
UPDATE src/app/app.routing.ts (2095 bytes)
...
```

The new and altered files are described in table 27.3.

NOTE You may find that your code editor starts highlighting errors in project files once the packages have been installed. These can be ignored and will usually go away if you restart your editor.

Table 27.3. The files added or modified for SSR

Name	Description
src/main.server.ts	This is the entry point for the application when it is executed on the server. It exports the JavaScript module defined in the <code>app.server.module.ts</code> file.
src/app/app.server.module.ts	This file defines the top-level Angular module for the server version of the application. The module combines the contents of the browser top-level module with the module that supports server execution.
tsconfig.server.json	This file configures the TypeScript compiler to build the server version of the application, which incorporates the <code>main.server.ts</code> and <code>server.ts</code> files in the output.
server.ts	This file contains code written for the Express API for handling HTTP requests and is the bridge between the HTTP server and the Angular application.

<code>package.json</code>	The <code>scripts</code> section of this file is modified to add new commands for building and running SSR features.
<code>angular.json</code>	This file is modified to add a separate set of configuration settings for the SSR version of the application.
<code>src/app/app.routing.ts</code>	The routing configuration is changed to set the <code>initialNavigation</code> configuration property, which blocks the initial navigation performed by the routing system until the root component has been created. The routes defined by the application are not altered.

27.4.2 Preparing the application

As already noted, SSR doesn't support all of the features that a normal client-side Angular application can use. In the sections that follow, I prepare the example application so that a subset of features will be available to the user via SSR, while ensuring that the full set of features is available once the JavaScript files have been downloaded and executed by the browser.

UNDOING THE ROUTING CONFIGURATION CHANGE

As noted in table 27.3, the Angular Universal package installation changes the routing configuration to set the `initialNavigation` configuration property, like this:

```
...
export const routing = RouterModule.forRoot(routes, {
  bindToComponentInputs: true,
  initialNavigation: 'enabledBlocking'
});
...
```

This setting changes the way the application starts so that the bootstrap process blocks until the initial navigation performed by the routing system is complete. This setting reportedly helps avoid content refreshes as the application migrates from SSR to browser-executed JavaScript, but it causes a problem with the example application.

The issue is that the resolver that ensures data is loaded before content is presented to the user relies on a signal effect and the process of triggering effects when underlying signals are modified is currently part of the content rendering system, whose initialization is blocked by the configuration change. The result is a deadlock: the initial navigation process is waiting for a resolver that depends on a signal effect that isn't executed until after the initial navigation process has completed.

This is a problem I hope will be resolved as signals are promoted from a developer preview and are fully integrated into the Angular platform, but for the present, the simplest solution is to reverse the configuration change, as shown in listing 27.13.

Listing 27.13. Disabling the configuration change in the `app.routing.ts` file in the `src/app` folder

```
...
```

```
export const routing = RouterModule.forRoot(routes, {
  bindToComponentInputs: true,
  //initialNavigation: 'enabledBlocking'
});
...
```

ENABLING THE HYDRATION FEATURE

The configuration change in listing 27.13 doesn't present a problem because this chapter also uses the hydration feature, which eases the migration from SSR to browser-executed JavaScript and which avoids the content rendering issue the `enabledBlocking` setting is intended to address.

Hydration is the process of reusing the HTML elements rendered by SSR when the browser starts executing the application's JavaScript code. Without hydration, the transition from SSR to browser-executed JavaScript requires replacing the HTML elements, which can be jarring to the user. Listing 27.14 enables the hydration feature. (Note that this configuration change is applied in the `app.module.ts` file and not the `app.server.module.ts` file that was created when the Angular Universal package was installed).

CAUTION Hydration is a developer preview feature in Angular 16 and is subject to change.

Listing 27.14 Enabling hydration in the `app.module.ts` file in the `src/app` folder

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

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

import { ModelModule } from "../model/model.module";
import { CoreModule } from "../core/core.module";
import { TableComponent } from "../core/table.component";
import { FormComponent } from "../core/form.component";
import { MessageModule } from "../messages/message.module";
import { MessageComponent } from "../messages/message.component";
import { AppComponent } from './app.component';
import { routing } from './app.routing';
import { TermsGuard } from './terms.guard';
import { LoadGuard } from './load.guard';

@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule, ModelModule, CoreModule, MessageModule,
    routing],
  providers: [TermsGuard, LoadGuard, provideClientHydration()],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

The `provideClientHydration` function is defined in the `@angular/platform-browser` package and is invoked so that its results are added to the module's `providers` array.

ADDING A PLATFORM DETECTION SERVICE

The application presented to the user via SSR typically has a subset of the full set of features and usually requires adaptations to ensure that code executes as expected in both the server and the browser. The key to this process is being able to easily identify the current environment. The Angular API allows the current platform to be identified, but I prefer to wrap this functionality in a service that is more easily used. Add a file named `platform.service.ts` in the `src/app` folder with the content shown in listing 27.15.

Listing 27.15. The contents of the `platform.service.ts` file in the `src/app` folder

```
import { isPlatformServer } from "@angular/common";
import { Inject, Injectable, PLATFORM_ID } from "@angular/core";

@Injectable()
export class PlatformService {

    constructor(@Inject(PLATFORM_ID) private platformId: Object) {}

    get isServer() { return isPlatformServer(this.platformId); }

}
```

The constructor uses the `@Inject` decorator to receive a service that has been defined using a value, rather than a class. The value in this case is `PLATFORM_ID` and the constructor receives an object that is passed to the `isPlatformServer` function, which Angular provides to identify when an application is running on the server. The service defines an `isServer` getter, which allows the platform to be identified without using the `PLATFORM_ID` service and `isPlatformServer` function throughout the application. Listing 27.16 registers the service.

Listing 27.16. Registering a service in the `app.module.ts` file in the `src/app` folder

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

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

import { ModelModule } from "../model/model.module";
import { CoreModule } from "../core/core.module";
import { TableComponent } from "../core/table.component";
import { FormComponent } from "../core/form.component";
import { MessageModule } from "../messages/message.module";
import { MessageComponent } from "../messages/message.component";
import { AppComponent } from './app.component';
import { routing } from './app.routing';
import { TermsGuard } from './terms.guard';
import { LoadGuard } from './load.guard';
import { PlatformService } from './platform.service';

@NgModule({
    declarations: [AppComponent],
    imports: [BrowserModule, ModelModule, CoreModule, MessageModule,
```

```

        routing],
    providers: [TermsGuard, LoadGuard, provideClientHydration(),
               PlatformService],
    bootstrap: [AppComponent]
  })
  export class AppModule { }

```

AVOIDING BROWSER APIS

SSR applications cannot rely on the APIs that browsers provide, because the code is executed at the server. All of the standard APIs provided by Angular will work in both the server and browser, but any direct reliance on the browser API must be removed or used only when the application is being executed by a browser.

There is only one place in the example application where I use the browser API, and that is when I compose the URL that will be used to request data from the RESTful web service. I didn't want to hard-code the hostname into the URL and so I used the global `location` object that browsers provide. In listing 27.17, I have altered the `RestDataSource` class so that HTTP requests are sent to `localhost` when the application is running on the server.

Listing 27.17. Changing the URL in the `rest.datasource.ts` file in the `src/app/model` folder

```

import { Injectable } from "@angular/core";
import { HttpClient, HttpHeaders } from "@angular/common/http";
import { Product } from "../product.model";
import { Observable, catchError } from "rxjs";
import { PlatformService } from "../platform.service";

//export const REST_URL = `http://${location.hostname}:3500/products`;

@Injectable()
export class RestDataSource {
  private REST_URL: string;

  constructor(private http: HttpClient, ps: PlatformService) {
    this.REST_URL = ps.isServer ?
      "http://localhost:3500/products"
      : `http://${location.hostname}:3500/products`;
  }

  getData(): Observable<Product[]> {
    return this.sendRequest<Product[]>("GET", this.REST_URL);
  }

  saveProduct(product: Product): Observable<Product> {
    return this.sendRequest<Product>("POST", this.REST_URL, product);
  }

  updateProduct(product: Product): Observable<Product> {
    return this.sendRequest<Product>("PUT",
      `${this.REST_URL}/${product.id}`, product);
  }

  deleteProduct(id: number): Observable<Product> {
    return this.sendRequest<Product>("DELETE",

```

```
        `${this.REST_URL}/${id}`);
    }

    private sendRequest<T>(verb: string, url: string, body?: Product)
      : Observable<T> {
      return this.http.request<T>(verb, url, {
        body: body,
        headers: new HttpHeaders({
          "Access-Key": "<secret>",
          "Application-Name": "exampleApp"
        })
      }).pipe(catchError((error: Response) => {
        throw `Network Error: ${error.statusText} (${error.status})`
      })));
    }
  }
}
```

The constructor uses the platform service to choose the URL to which requests are sent, using localhost when the application is running on the server and relying on the `location` object to determine the hostname when running in a browser. Both URLs assume that the HTTP server that delivers the application to the client is running alongside the RESTful web service.

27.4.3 Building and running the SSR application

The process of supporting SSR isn't complete, but the application is now ready for a quick test to make sure the basics are in place. One of the changes made to the `package.json` file during the package set up was the addition of commands for building and running the application with SSR support, as described in table 27.4.

Table 27.4. The SSR commands

Command	Description
<code>dev:ssr</code>	This command builds and runs the application in development mode, with a watcher and automatic rebuilds and reloads.
<code>build:ssr</code>	This command builds the application for deployment.
<code>serve:ssr</code>	This command executes the production SSR server.

To start the Angular development server with SSR support, run the command shown in listing 27.18 in the `exampleApp` folder.

Listing 27.18. Building and running the application

```
npm run dev:ssr
```

The best way to test SSR is to disable JavaScript in the browser, otherwise the SSR version of the application will be replaced with the browser-based version as soon as the browser loads the JavaScript files. For Chrome on Windows, open the menu by clicking on the three vertical dots and select Run Command. Type `java` into the prompt and select Disable JavaScript from the list of options, as shown in figure 27.6.

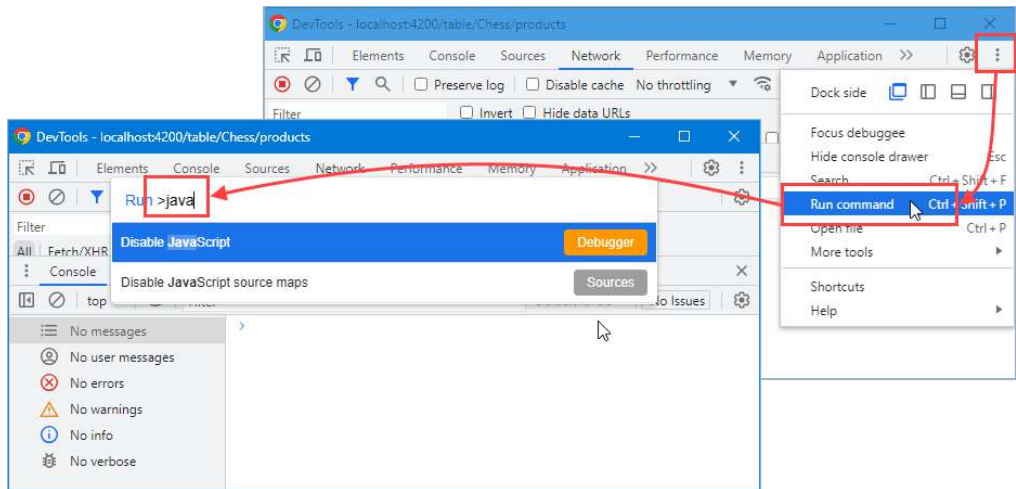


Figure 27.6. Running a command in Chrome

Once JavaScript has been disabled and the Angular build process has completed, use the browser to request <http://localhost:4200>, which will load the HTML-only version of the application, as shown in figure 27.7.

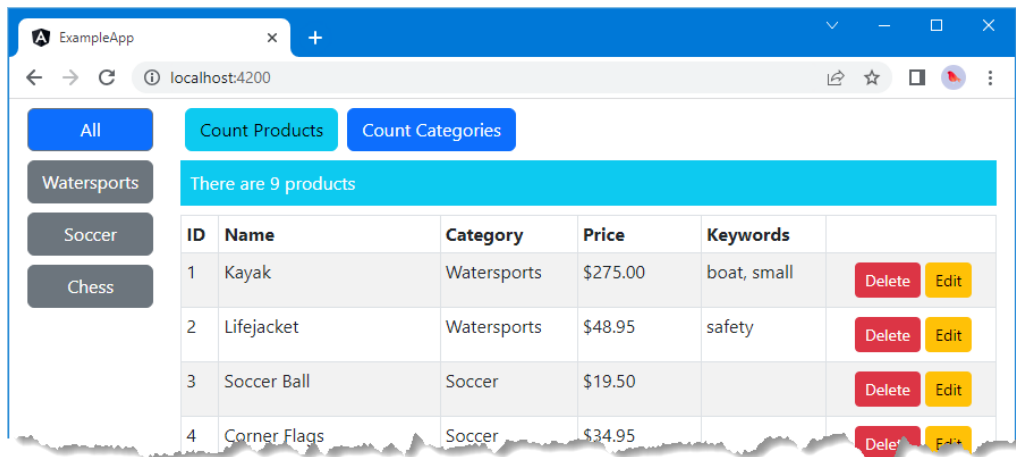


Figure 27.7. Loading the HTML version of the application

There is no difference in the appearance of the application at this point, but you will find that none of the buttons work and that the content presented to the user is essentially static. But

importantly, the Angular application has been executed by the server to produce the HTML that was sent to the browser, confirming that the basic SSR features are working.

With the Fast 3G bandwidth throttling enabled, a hard reload with an empty cache takes 2 seconds on my development machine, which means the user is left looking at an empty browser window for just a small fraction of the time when compared to the start of the chapter.

27.4.4 Enabling navigation

The SSR version of the application renders fast but isn't especially useful yet. None of the buttons presented by the application work, because they rely on event handlers, applied to `button` elements. Navigation in an HTML-only application requires using anchor elements (with the `a` tag), which means revising the templates to change element tags. The Angular navigation directives and the Bootstrap CSS styles work equally well on button and anchor elements, which simplifies the process.

Not all application features have to be included in the SSR version of the application, especially if the expectation is to give the user something basic to work with until the JavaScript files are loaded, and the browser application is started.

This means that I can switch to anchor elements for the navigation features I want to support and just disable the button elements for the features that will be available once the browser JavaScript is ready. To prepare, listing 27.19 adds a property to the `TableComponent` class that provides its template with details of the current platform.

Listing 27.19. Adding a property in the `table.component.ts` file in the `src/app/core` folder

```
import { Component, Input, Signal, computed } from "@angular/core";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";
import { PlatformService } from "../platform.service";

@Component({
  selector: "paTable",
  templateUrl: "table.component.html"
})
export class TableComponent {

  constructor(private model: Model, private ps: PlatformService) { }

  @Input()
  category?: string

  getProduct(key: number): Product | undefined {
    return this.model.getProduct(key);
  }

  get Products(): Signal<Product[]> {
    return computed(() => {
      return this.model.Products().filter(p =>
        this.category == null || p.category == this.category);
    });
  }
}
```

```

get Categories(): Signal<string[]> {
  return computed(() => {
    return this.model.Products()
      .map(p => p.category)
      .filter((c, index, arr) => c != undefined
        && arr.indexOf(c) == index) as string[];
  })
}

deleteProduct(key?: number) {
  if (key != undefined) {
    this.model.deleteProduct(key);
  }
}

get isServer() { return this.ps.isServer }
}

```

Listing 27.20 updates the component's template to allow limited navigation while disabling other elements.

Listing 27.20. Managing navigation in the table.component.html file in the src/app/core folder

```

<div class="container-fluid">
  <div class="row">
    <div class="col-auto">
      <div class="d-grid gap-2">
        <a class="btn btn-secondary"
          routerLink="/table" routerLinkActive="bg-primary"
          [routerLinkActiveOptions]="{exact: true}">
          All
        </a>
        <a *ngFor="let category of Categories()"
          class="btn btn-secondary"
          [routerLink]="['/table', category]"
          routerLinkActive="bg-primary">
          {{category}}
        </a>
      </div>
    </div>
  </div>
  <div class="col">
    <a class="btn btn-info mx-1" routerLink="products">
      Count Products
    </a>
    <a class="btn btn-primary mx-1" routerLink="categories">
      Count Categories
    </a>
    <div class="my-2">
      <router-outlet></router-outlet>
    </div>
    <table class="table table-sm table-bordered table-striped">
      <thead>
        <tr>
          <th>ID</th><th>Name</th><th>Category</th>

```

```

        <th>Price</th><th>Keywords</th><th></th>
    </tr>
</thead>
<tbody>
    <tr *ngFor="let item of Products()">
        <td>{{item.id}}</td>
        <td>{{item.name}}</td>
        <td>{{item.category}}</td>
        <td>{{item.price | currency:"USD" }}</td>
        <td>{{ item.keywords?.join(", ")}}</td>
        <td class="text-center">
            <button class="btn btn-danger btn-sm m-1"
                (click)="deleteProduct(item.id)"
                [disabled]="isServer">
                Delete
            </button>
            <button class="btn btn-warning btn-sm"
                [routerLink]="['/form', 'edit', item.id]"
                [disabled]="isServer">
                Edit
            </button>
        </td>
    </tr>
</tbody>
</table>
</div>
</div>
</div>

<div class="p-2 text-center">
    <button class="btn btn-primary mt-1" routerLink="/form/create"
        [disabled]="isServer">
        Create New Product
    </button>
    <ng-container *ngIf="!isServer">
        <button class="btn btn-danger mt-1 mx-1"
            (click)="deleteProduct(-1)">
            Generate HTTP Error
        </button>
        <button class="btn btn-danger m-1" routerLink="/does/not/exist">
            Generate Routing Error
        </button>
        <button class="btn btn-danger" routerLink="/ondemand">
            Load Module
        </button>
    </ng-container>
</div>

```

I have changed the `button` elements that select categories to anchor elements, which means they will work in the SSR version of the application. For other button elements, I have used data bindings to set the `disabled` attribute, which means that the buttons will do nothing initially, but will become active once the browser starts executing the application. I have also introduced an `ng-container` element to prevent some elements from being displayed by the SSR version of the application, just for variety.

27.4.5 Modifying the terms guard

One of the features enabled in the SSR version of the application is the ability to display the number of categories, which is a navigation change protected by a guard that prompts the user and waits for confirmation. This type of guard doesn't work with SSR because the user never gets to see the prompt and doesn't have the chance to respond. Listing 27.21 modifies the guard to always allow navigation when the application is being executed on the server.

Listing 27.21. Allowing navigation in the terms.guard.ts file in the src/app folder

```
import { Injectable } from "@angular/core";
import {
  ActivatedRouteSnapshot, RouterStateSnapshot, Router
} from "@angular/router";
import { MessageService } from "../messages/message.service";
import { Message } from "../messages/message.model";
import { PlatformService } from "../platform.service";

@Injectable()
export class TermsGuard {

  constructor(private messages: MessageService,
              private router: Router,
              private ps: PlatformService) { }

  canActivate(route: ActivatedRouteSnapshot,
              state: RouterStateSnapshot): Promise<boolean> | boolean {

    if (route.params["mode"] == "create") {

      return new Promise<boolean>(resolve => {
        let responses: [string, () => void][]
          = [
            ["Yes", () => resolve(true)],
            ["No", () => resolve(false)]
          ];
        this.messages.reportMessage(
          new Message("Do you accept the terms & conditions?",
            false, responses));
      });
    } else {
      return true;
    }
  }

  canActivateChild(route: ActivatedRouteSnapshot,
                  state: RouterStateSnapshot): Promise<boolean> | boolean {

    if ((!this.ps.isServer) && route.url.length > 0
        && route.url[route.url.length - 1].path == "categories") {

      return new Promise<boolean>((resolve, reject) => {
        let responses: [string, (arg: string) => void][] = [
          ["Yes", () => resolve(true)],
          ["No ", () => resolve(false)]
        ];
      });
    }
  }
}
```



```

        this.messages.reportMessage(
            new Message("Do you want to see the categories?",
                false, responses));
    });
    } else {
        return true;
    }
}
}

```

Save the changes and reload the browser to see the changes (the browser won't reload automatically because JavaScript is disabled). Navigating between product categories or displaying the number of products/categories now works, as shown in figure 27.8.

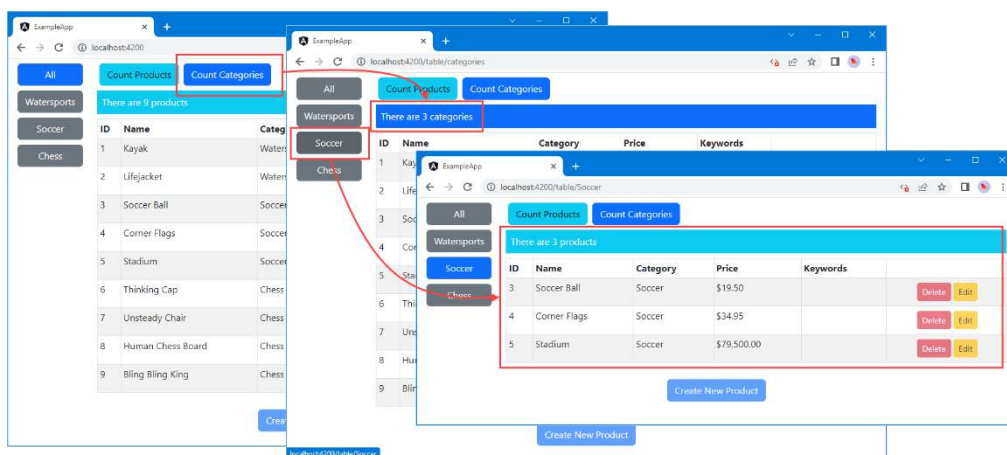
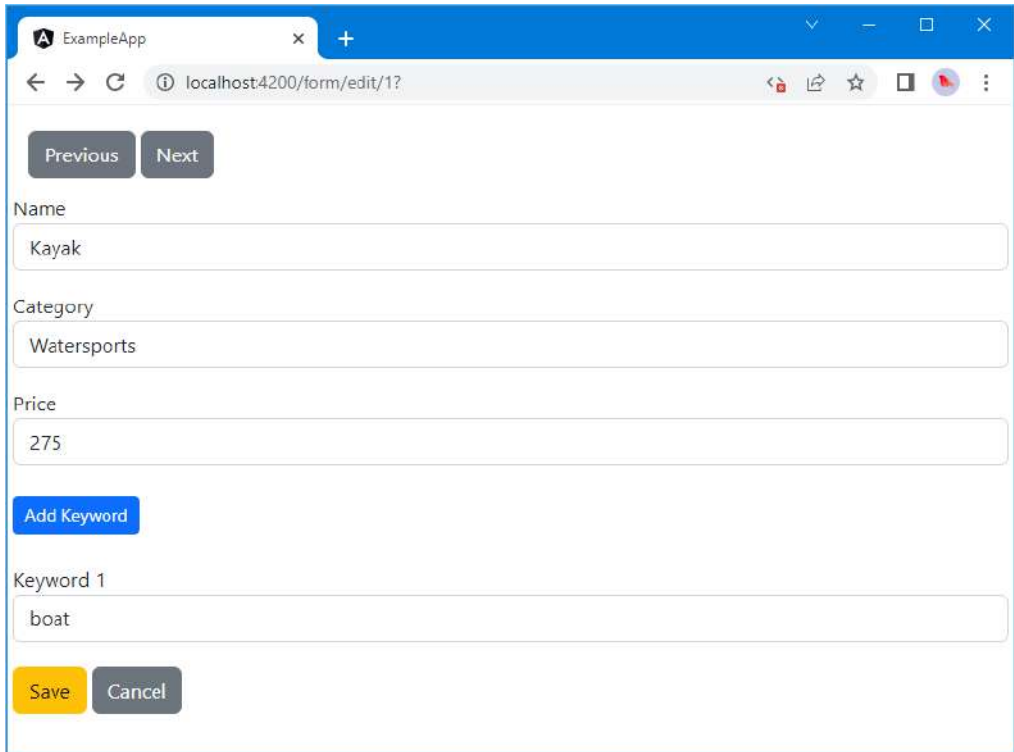


Figure 27.8. Enabling navigation for the SSR application

27.4.6 Dealing with direct navigation

One problem with the SSR process is that it makes all of the application routes available for direct navigation, which means that users can navigate to parts of the application that are not suited to SSR. You can see this by requesting <http://localhost:4200/form/edit/1>, which produces the result shown in figure 27.9.



The screenshot shows a web browser window with the title 'ExampleApp'. The address bar displays 'localhost:4200/form/edit/1?'. The page content includes a form with the following elements:

- Two buttons at the top: 'Previous' and 'Next'.
- A text input field labeled 'Name' containing the text 'Kayak'.
- A text input field labeled 'Category' containing the text 'Watersports'.
- A text input field labeled 'Price' containing the text '275'.
- A blue button labeled 'Add Keyword'.
- A text input field labeled 'Keyword 1' containing the text 'boat'.
- Two buttons at the bottom: a yellow 'Save' button and a grey 'Cancel' button.

Figure 27.9. Direct navigation to a URL

I disabled the Edit buttons because editing product details relies on features that are not well-suited to SSR, such as data validation. The user can change the values in the form fields but clicking on the Save button doesn't send the data to the server. This is because the `form` element presented to the user is configured to send HTTP GET requests by default but changing the form element wouldn't help because the SSR process doesn't create handlers for HTTP POST requests.

Making the edit features work goes beyond the point where an Angular SSR is useful and using a dedicated framework for a separate HTML application is preferable. You can write request handlers that will be executed by the Express server that is generating the SSR content, but that is no longer Angular code, even if it is still JavaScript/TypeScript. (Express is an excellent server and I think highly of it, but there are easier ways to write HTML-only applications if that's what you require).

To preserve my use of Angular SSR as a way to improve the delivery of a browser-based application, I want to restrict SSR to specific routes and ignore other requests. One way to do this is to use the Express server to filter out specific URLs, but my preference is to use the

Angular routing system directly using a route guard. Add a file named `browser.guard.ts` to the `src/app` folder with the content shown in listing 27.22.

Listing 27.22. The content of the `browser.guard.ts` file in the `src/app` folder

```
import { Injectable } from "@angular/core";
import { PlatformService } from "../platform.service";
import { ActivatedRouteSnapshot, Router, RouterStateSnapshot }
  from "@angular/router";

@Injectable()
export class BrowserGuard {

  constructor(private router: Router, private ps: PlatformService) {}

  canActivate(route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot) {

    if (this.ps.isServer) {
      this.router.navigateByUrl("/");
      return false;
    }
    return true;
  }
}
```

The class implements the `canActivate` method, which means it can be used as a route activation guard. The platform service is used to determine whether the application is running on the server and performs a route redirection if it is.

The effect is that requests for direct URLs to which the guard is applied will return content generated for the default route. Listing 27.23 registers the guard class as a service.

Listing 27.23. Registering the guard in the `app.module.ts` file in the `src/app` folder

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

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

import { ModelModule } from "../model/model.module";
import { CoreModule } from "../core/core.module";
import { TableComponent } from "../core/table.component";
import { FormComponent } from "../core/form.component";
import { MessageModule } from "../messages/message.module";
import { MessageComponent } from "../messages/message.component";
import { AppComponent } from './app.component';
import { routing } from './app.routing';
import { TermsGuard } from './terms.guard';
import { LoadGuard } from './load.guard';
import { PlatformService } from '../platform.service';
import { BrowserGuard } from './browser.guard';

@NgModule({
  declarations: [AppComponent],
  imports: [BrowserModule, ModelModule, CoreModule, MessageModule,
```

```

        routing],
    providers: [TermsGuard, LoadGuard, provideClientHydration(),
        PlatformService, BrowserGuard],
    bootstrap: [AppComponent]
  })
  export class AppModule { }

```

The final step is to apply the guard to the routes that I want to disable for SSR, as shown in listing 27.24.

Listing 27.24. Guarding routes in the app.routing.ts file in the src/app folder

```

import { Routes, RouterModule, mapToResolve, mapToCanActivate,
    mapToCanActivateChild, mapToCanDeactivate } from "@angular/router";
import { TableComponent } from "../core/table.component";
import { FormComponent } from "../core/form.component";
import { NotFoundComponent } from "../core/notFound.component";
import { ProductCountComponent } from "../core/productCount.component";
import { CategoryCountComponent } from "../core/categoryCount.component";
import { ModelResolver } from "../model/model.resolver";
import { TermsGuard } from "../terms.guard";
import { UnsavedGuard } from "../core/unsaved.guard";
import { LoadGuard } from "../load.guard";
import { BrowserGuard } from "../browser.guard";

const childRoutes: Routes = [
  {
    path: "",
    canActivateChild: mapToCanActivateChild([TermsGuard]),
    children: [{ path: "products", component: ProductCountComponent },
      { path: "categories",
        component: CategoryCountComponent },
      { path: "", component: ProductCountComponent }],
    resolve: { model: mapToResolve(ModelResolver) }
  }
];

const routes: Routes = [
  {
    path: "ondemand",
    canActivate: mapToCanActivate([BrowserGuard, LoadGuard]),
    loadChildren: () => import("../ondemand/ondemand.module")
      .then(m => m.OndemandModule)
  },
  {
    path: "form/:mode/:id", component: FormComponent,
    canActivate: mapToCanActivate([BrowserGuard]),
    resolve: { model: mapToResolve(ModelResolver) },
    canDeactivate: mapToCanDeactivate([UnsavedGuard])
  },
  {
    path: "form/:mode", component: FormComponent,
    resolve: { model: ModelResolver },
    canActivate: mapToCanActivate([BrowserGuard, TermsGuard])
  },
  { path: "does", redirectTo: "/form/create", pathMatch: "prefix" },
  { path: "table", component: TableComponent, children: childRoutes },
  { path: "table/:category", component: TableComponent,

```

```

    children: childRoutes },
    { path: "", redirectTo: "/table", pathMatch: "full" },
    { path: "**", component: NotFoundComponent }]

export const routing = RouterModule.forRoot(routes, {
  bindToComponentInputs: true,
  //initialNavigation: 'enabledBlocking'
});

```

Save the changes and use the browser to request <http://localhost:4200/form/edit/1>, which will produce the result shown in figure 27.10. This isn't a perfect approach because the redirection performed by the guard alters the content generated by the application without modifying the URL displayed by the browser, which may confuse the user, but it does solve the problem without requiring non-Angular code to be written.

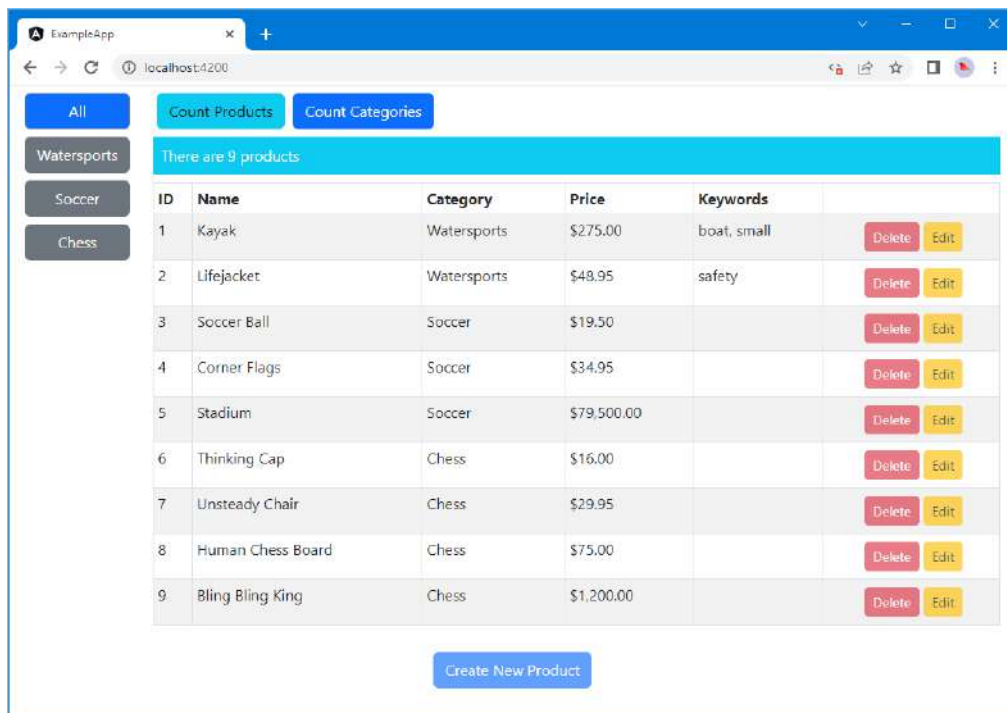


Figure 27.10. Disabling routes for SSR

27.4.7 Testing rehydration

The final step is to check that the JavaScript code executed by the browser reuses the HTML elements created by SSR, which is handled by the rehydration process. This process can be hard to spot when the browser is running on the same machine as the Angular server, because the JavaScript files are loaded so quickly. To slow down the process, enable the Fast 3G

bandwidth throttle used earlier in the chapter, use the browser’s Run Command option to enable JavaScript (start typing `Java` and the `Enable JavaScript` command will be displayed in the list of options) and then hold down the browser’s reload button with the F12 developer tools window open to select the “Empty cache and hard reload” option, which will ensure the browser loads all of the JavaScript files from the server, even if they were previously cached.

It will take a moment for the browser to load the JavaScript files. Observe the browser window and you will see new HTML elements displayed and existing elements enabled when the JavaScript is executed and the navigation that was previously disabled becomes available, as shown in figure 27.11.

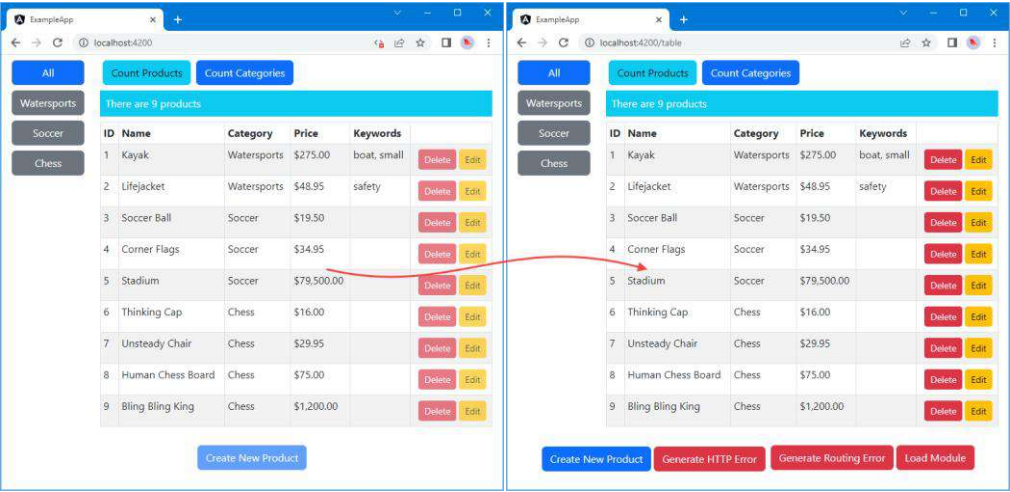


Figure 27.11. Rehydrating an SSR application

27.5 Using prerendering

An alternative to SSR is *prerendering*, which creates static HTML documents for the routes defined by an application once, instead of creating them dynamically for each request. Prerendering doesn’t require the server-side resources that SSR can demand but the HTML documents sent to the client are static snapshots that won’t reflect changes in shared state, such as data in a database.

27.5.1 Preparing the application

One consequence of generating all the content in advance is that routes that accept runtime parameters and child routes won’t be included in the HTML output automatically. To address this shortcoming, create a file named `routes.txt` in the `exampleApp` folder, with the content shown in listing 27.25.

Listing 27.25. The contents of the `routes.txt` file in the `exampleApp` folder

```

/table/Watersports
/table/Soccer
/table/Chess
/table/products
/table/categories

```

The prerender process will detect some routes automatically and the file should only be used to add those that are missing, which can be a process of trial and error for each application.

Listing 27.26 adds an argument to the command that will prerender the application to include the route file.

Listing 27.26. Adding an argument in the package.json file in the exampleApp folder

```

...
"scripts": {
  "ng": "ng",
  "start": "ng serve",
  "build": "ng build",
  "watch": "ng build --watch --configuration development",
  "test": "ng test",
  "json": "json-server --p 3500 restData.js",
  "dev:ssr": "ng run exampleApp:serve-ssr",
  "serve:ssr": "node dist/exampleApp/server/main.js",
  "build:ssr": "ng build && ng run exampleApp:server",
  "prerender": "ng run exampleApp:prerender --routes-file routes.txt"
},
...

```

27.5.2 Prerendering the application

To prerender the application, stop the `npm run dev:ssr` command used to run the SSR application and run the command shown in listing 27.27 in the `exampleApp` folder.

Listing 27.27. Prerendering the example application

```
npm run prerender
```

This command creates a static HTML representation of the application in the `dist/exampleApp/browser` folder. There is no option to dynamically build and serve a prerendered application, which means that a separate HTTP server is required. Run the command shown in listing 27.28 in the `exampleApp` folder to download and execute the excellent JavaScript `http-server` package.

Listing 27.28. Running an HTTP server to deliver the prerendered application

```
npx http-server@14.1.1 dist/exampleApp/browser --port 4500
```

Once the server is running, open a new browser window, disable JavaScript, and request `http://localhost:4500`. The browser will receive an HTML document and clicking on the category buttons will load other prerendered HTML documents, as shown in figure 27.12.

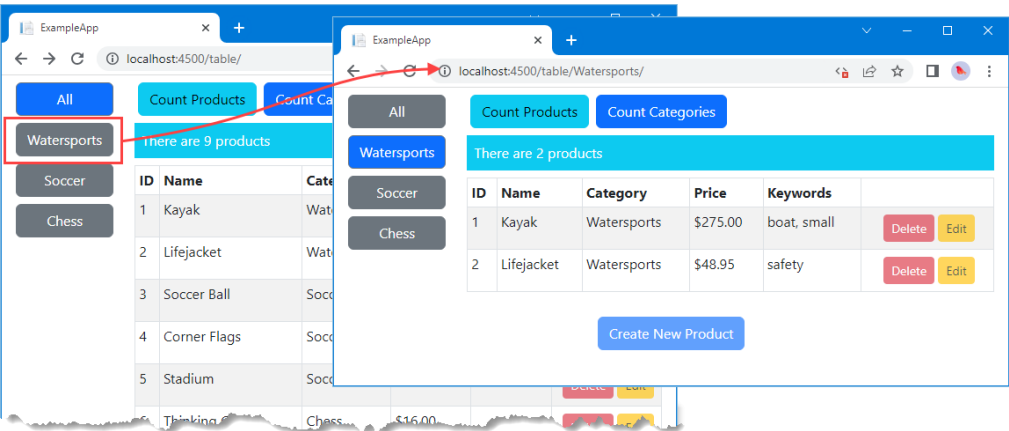


Figure 27.12. Prerendering the application

Enable JavaScript and reload the browser and you will see the rehydration process use the prerendered HTML as the basis for dynamically generated content, as shown in figure 27.13.

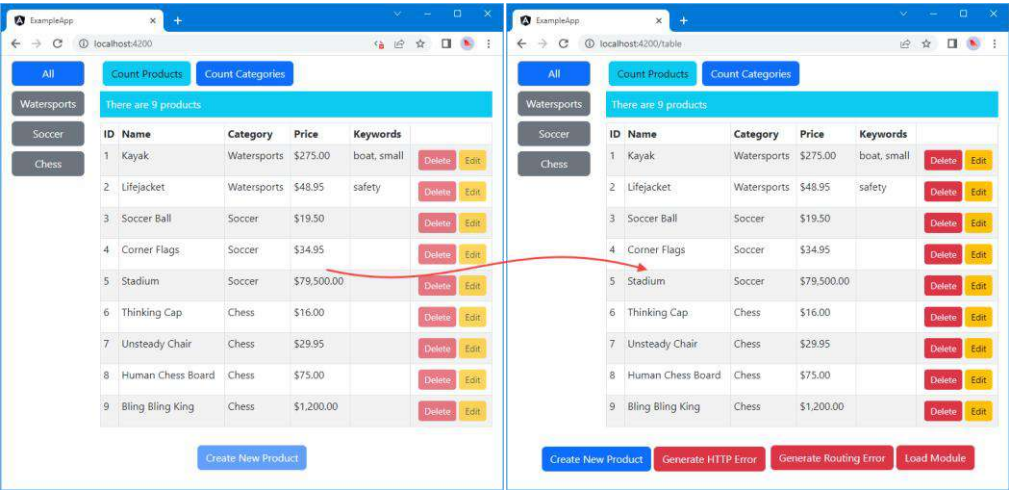


Figure 27.13. Rehydrating a prerendered application

27.6 Summary

In this chapter, I described the Angular features for optimizing the delivery of the application.

- Modules can be loaded dynamically, so they can be excluded from the large initial download required to run the application.

- Dynamically loaded modules can be guarded to prevent accidental loading.
- Dynamic modules must be kept separate from the main application to avoid including them in the initial download.
- Server-side rendering executes the application on the server and returns an HTML document that the browser can display while waiting for the JavaScript files to download.
- The rehydration feature uses the elements in the HTML document when the JavaScript code executes in the browser.
- Not all Angular features work with server-side rendering and applications must be carefully prepared.
- Prerendering uses server-side rendering to create static HTML documents that don't require the resources of a live application. Rehydration still works, but the HTML content sent to the browser will not reflect subsequent changes in state or data.

In the next chapter, I describe the use of Angular component libraries.

28

Working with component libraries

This chapter covers

- Installing and using the Angular Material component library
- Selecting component library features to use
- Integrating component library features with existing functionality

Component libraries are packages that contain Angular components and directives, such as buttons, tables, and layouts. Throughout this book, I have been creating custom components and directives to demonstrate Angular features, but component libraries use these same features to provide building blocks that you can use to simplify the development process.

One of the recurring themes in this book is that nothing in Angular is magic, and this extends to component libraries, which are written using the same features that you used in earlier chapters. Component libraries are useful because they mean you don't have to write code and templates for basic tasks, such as creating a button, for example, and can focus on dealing with what happens when the user clicks the button.

In this chapter, I use the Angular Material component library to add components to the project and explain how to use CSS to give a custom component an appearance that is consistent with the library components. Table 28.1 puts the use of component libraries in context.

NOTE This chapter is not a detailed description of Angular Material or any other component library. There are several good component libraries available for Angular and each has its own set of features and API.

Table 28.1. Putting component libraries in context

Question	Answer
----------	--------

What are they?	Component libraries are packages containing commonly required user interface features for Angular applications.
Why are they useful?	Component libraries can speed up project development and ensure a consistent appearance in the finished application.
How are they used?	Features are presented as Angular components or directives, which are applied in the same way as custom components and directives.
Are there any pitfalls or limitations?	Component libraries can require data to be presented in a specific way or for the application to be structured using a specific pattern. These restrictions may not suit all projects.
Are there any alternatives?	Component libraries are entirely optional and are not required for Angular development.

Table 28.2 summarizes the chapter.

Table 28.2. Chapter summary

Problem	Solution	Listing
Applying the features provided by a component library	Use the components or directives provided contained in the library package	4–11
Using the advanced features provided by a component library	Adopt the structure or API that the component library provides for integration	12–14
Styling custom components to match the theme used by the component library	Use the CSS styles provided by the component library, which are typically provided for use with Sass	15–23

28.1 Preparing for this chapter

In this chapter, I continue using the `exampleApp` project that was first created in chapter 20 and has been the focus of every chapter since.

TIP You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/manningbooks/pro-angular-16>. See chapter 1 for how to get help if you have problems running the examples.

To prepare for this chapter, open a new command prompt, navigate to the `exampleApp` folder, and run the command shown in listing 28.1 in the `exampleApp` folder to download and install the Angular Material package:

Listing 28.1. Installing the package

```
ng add @angular/material@^16
```

The Angular Material package uses the schematics API to configure the project. Press Y to confirm the installation. Select the default option for the questions asked by the installer, which will complete the installation and list the files that have been updated:

```
...
UPDATE package.json (1664 bytes)
UPDATE angular.json (5361 bytes)
UPDATE src/index.html (585 bytes)
UPDATE src/styles.css (181 bytes)
...
```

Choosing a component library

I have used Angular Material because it is the most popular Angular component library. There are several other packages available. Teradata Covalent (<https://teradata.github.io/covalent>) is an open-source library that follows the same Material Design standard as Angular Material, but with the addition of good charting components. Some packages present the features of the Bootstrap CSS package using Angular features, such as ng-bootstrap (<https://ng-bootstrap.github.io>) and ngx-bootstrap (<https://valor-software.com/ngx-bootstrap>), and each provides a different approach to developing components. There are also commercial packages, such as Kendo UI (<https://www.telerik.com/kendo-angular-ui>), which can be useful for development teams that require support.

If you don't know where to start, then try Angular Material. The documentation (<https://material.angular.io>) is good, and the package contains the components required by most projects.

28.1.1 Removing buttons

Listing 28.2 replaces the contents of the template for the table component, removing features that are not required for this chapter.

Listing 28.2. The contents of the table.component.html file in the src/app/core folder

```
<table class="table table-sm table-bordered table-striped">
  <thead>
    <tr>
      <th>ID</th><th>Name</th><th>Category</th>
      <th>Price</th><th>Keywords</th><th></th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let item of Products()">
      <td>{{item.id}}</td>
      <td>{{item.name}}</td>
      <td>{{item.category}}</td>
      <td>{{item.price | currency:"USD" }}</td>
      <td>{{ item.keywords?.join(", ")}}</td>
```

```

        <td class="text-center">
            <button class="btn btn-danger btn-sm m-1"
                (click)="deleteProduct(item.id)"
                [disabled]="isServer">
                Delete
            </button>
            <button class="btn btn-warning btn-sm"
                [routerLink]="['/form', 'edit', item.id]"
                [disabled]="isServer">
                Edit
            </button>
        </td>
    </tr>
</tbody>
</table>

<div class="p-2 text-center">
    <button class="btn btn-primary mt-1" routerLink="/form/create"
        [disabled]="isServer">
        Create New Product
    </button>
</div>

```

28.1.2 Adjusting the HTML file

Installing the Angular Material package requires a change to the `index.html` file to resolve a conflict with the Bootstrap CSS styles that causes a scrollbar to be displayed even when the content fits within the browser window, caused by styles added to the `styles.css` file. Listing 28.3 changes the class to which the `body` element is assigned to resolve the issue.

Listing 28.3. Changing an element class in the `index.html` file in the `src` folder

```

<!doctype html>
<html lang="en">
<head>
    <meta charset="utf-8">
    <title>ExampleApp</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 class="p-1">
    <app-root></app-root>
</body>
</html>

```

28.1.3 Running the project

Open a new command prompt, navigate to the `exampleApp` folder, and run the following command to start the server that provides the RESTful web server:

```
npm run json
```

Open a separate command prompt, navigate to the `exampleApp` folder, and run the following command to start the Angular development tools:

```
ng serve
```

Open a new browser window and navigate to `http://localhost:4200` to see the content shown in figure 28.1.

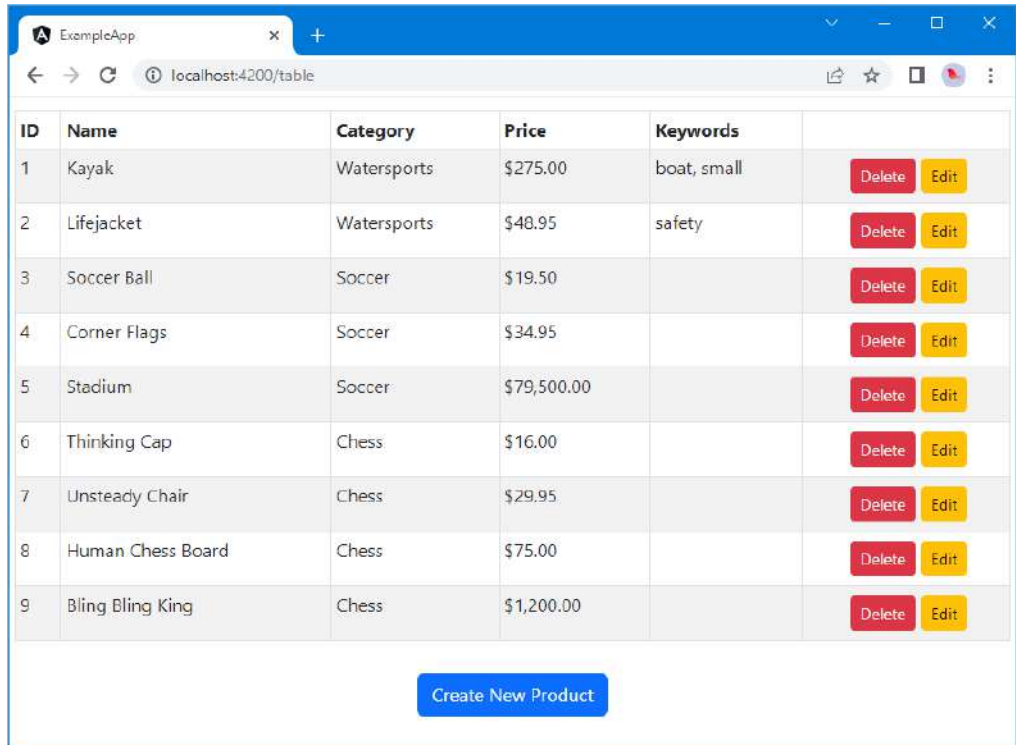


Figure 28.1. Running the example application

28.2 Using the library components

The simplest approach to using a component library is, as you might expect, to use the components it provides. In this section, I demonstrate how to integrate two features from the Angular Material library into the example project.

28.2.1 Using the Angular Material button directive

The Angular Material support for buttons is provided as a directive applied to button or anchor elements, as shown in listing 28.4.

Listing 28.4. Using the Angular Material button in the table.component.html File in the src/app/core folder

```

<table class="table table-sm table-bordered table-striped">
  <thead>
    <tr>
      <th>ID</th><th>Name</th><th>Category</th>
      <th>Price</th><th>Keywords</th><th></th>
    </tr>
  </thead>
  <tbody>
    <tr *ngFor="let item of Products()">
      <td>{{item.id}}</td>
      <td>{{item.name}}</td>
      <td>{{item.category}}</td>
      <td>{{item.price | currency:"USD" }}</td>
      <td>{{ item.keywords?.join(", ")}}</td>
      <td class="text-center">
        <button mat-flat-button color="accent"
          (click)="deleteProduct(item.id)"
          [disabled]="isServer">
          Delete
        </button>
        <button mat-flat-button color="warn"
          [routerLink]="['/form', 'edit', item.id]"
          [disabled]="isServer">
          Edit
        </button>
      </td>
    </tr>
  </tbody>
</table>

<div class="p-2 text-center">
  <button mat-flat-button color="primary" routerLink="/form/create"
    [disabled]="isServer">
    Create New Product
  </button>
</div>

```

Angular Material provides several different styles of button, which are applied using the attributes described in table 28.3.

Table 28.3. The Angular Material button attributes

Name	Description
mat-button	This attribute creates a simple borderless button, whose text is styled using an Angular Material theme color.
mat-stroked-button	This attribute adds a rectangular border to the mat-button style.
mat-raised-button	This attribute creates a button that appears to be raised from the page, displayed with a small amount of shadow. The button background is styled using an Angular Material theme color.

<code>mat-flat-button</code>	This attribute creates a button without the raised shadow and whose background is styled using an Angular Material theme color.
<code>mat-icon-button</code>	This attribute creates a button with a transparent background, intended to display an icon, which is styled using an Angular Material theme color.
<code>mat-fab</code>	This attribute creates a circular button with a shadow, whose background is styled using an Angular Material theme color.
<code>mat-mini-fab</code>	This button creates a small circular button with a shadow and a background styled using an Angular Material theme color.

Angular Material uses a color theme that is selected when the package is installed and which defines three color names, as described in table 28.4.

Table 28.4. The Angular Material color names

Name	Description
<code>primary</code>	This name refers to the color used most often throughout the application.
<code>accent</code>	This name refers to the color used to highlight key parts of the user interface.
<code>warn</code>	This name refers to the color used for warnings and errors or to denote operations that require caution.

In listing 28.4, I applied the `mat-flat-button` attribute, which will create a button whose appearance most closely matches the buttons I created using the Bootstrap styles. The theme color is specified using the `color` attribute, like this:

```
...
<button mat-flat-button color="accent" (click)="deleteProduct(item.id)"
  [disabled]="isServer">
...

```

The Angular Material button is applied to a regular HTML `button` element, which means that the `click` event is used to respond to user interaction.

ADDING THE MARGIN STYLE

Angular Material doesn't include utility styles for adding margins or padding to elements. Listing 28.5 defines a new global style that adds space around flat buttons.

CAUTION You may be tempted to mix and match styles from different packages, such as applying the Bootstrap `m-1` style to button elements to which the `mat-flat-button` attribute has been added. Care must be taken because package styles are rarely written with this kind of combination in mind and there can be odd interactions.

Listing 28.5. Adding Styles in the `styles.css` File in the `src` Folder

```
html, body { height: 100%; }
body { margin: 0; font-family: Roboto, "Helvetica Neue", sans-serif; }
```



```
button[mat-flat-button] { margin: 2px; }
```

The Angular Material attributes added to elements are preserved in the content presented to the user, which means that I can use a CSS selector to locate button elements and apply a margin.

IMPORTING THE COMPONENT MODULE

Angular Material uses separate modules for each feature, which means that an application includes only the features it requires and doesn't add unused code to the download required by the client.

In a complex project, there can be a large number of dependencies on a component library, and they can be spread throughout the project's modules. To make it easier to manage the dependencies, it is a good idea to use a separate module. Add a file named `material.module.ts` in the `src/app` folder with the content shown in listing 28.6.

Listing 28.6. The contents of the `material.module.ts` file in the `src/app` folder

```
import { NgModule } from "@angular/core";
import { MatButtonModule } from "@angular/material/button";

const features = [MatButtonModule];

@NgModule({
  imports: [features],
  exports: [features]
})
export class MaterialFeatures {}
```

The `MaterialFeatures` module imports and exports the `MatButtonModule` module from the Angular Material package. Listing 28.7 adds a dependency on the new module, which will be the only change to the core module to enable Angular Material features.

Listing 28.7. Importing a module in the `core.module.ts` file in the `src/app/core` folder

```
import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule, NG_VALIDATORS, ReactiveFormsModule }
  from "@angular/forms";
import { ModelModule } from "../model/model.module";
import { TableComponent } from "../table.component";
import { FormComponent } from "../form.component";
//import { SharedState } from "../sharedState.service";
import { ValidationHelper } from "../validationHelper.pipe";
import { ValidationErrorsDirective } from "../validationErrors.directive";
import { HiLowValidatorDirective } from "../validation/hilow";
import { RouterModule } from "@angular/router";
import { ProductCountComponent } from "../productCount.component";
import { CategoryCountComponent } from "../categoryCount.component";
import { NotFoundComponent } from "../notFound.component";
import { UnsavedGuard } from "../unsaved.guard";
import { MaterialFeatures } from "../material.module"

@NgModule({
  imports: [BrowserModule, FormsModule, ModelModule,
    ReactiveFormsModule, RouterModule, MaterialFeatures],
```

```

    declarations: [TableComponent, FormComponent, ValidationHelper,
        ValidationErrorsDirective, HiLowValidatorDirective,
        ProductCountComponent, CategoryCountComponent,
        NotFoundComponent],
    exports: [ModelModule, TableComponent, FormComponent],
    providers: [UnsavedGuard]
  })
  export class CoreModule { }

```

Save the changes, and the Angular Material button will be displayed when the application is reloaded, as shown in figure 28.2.

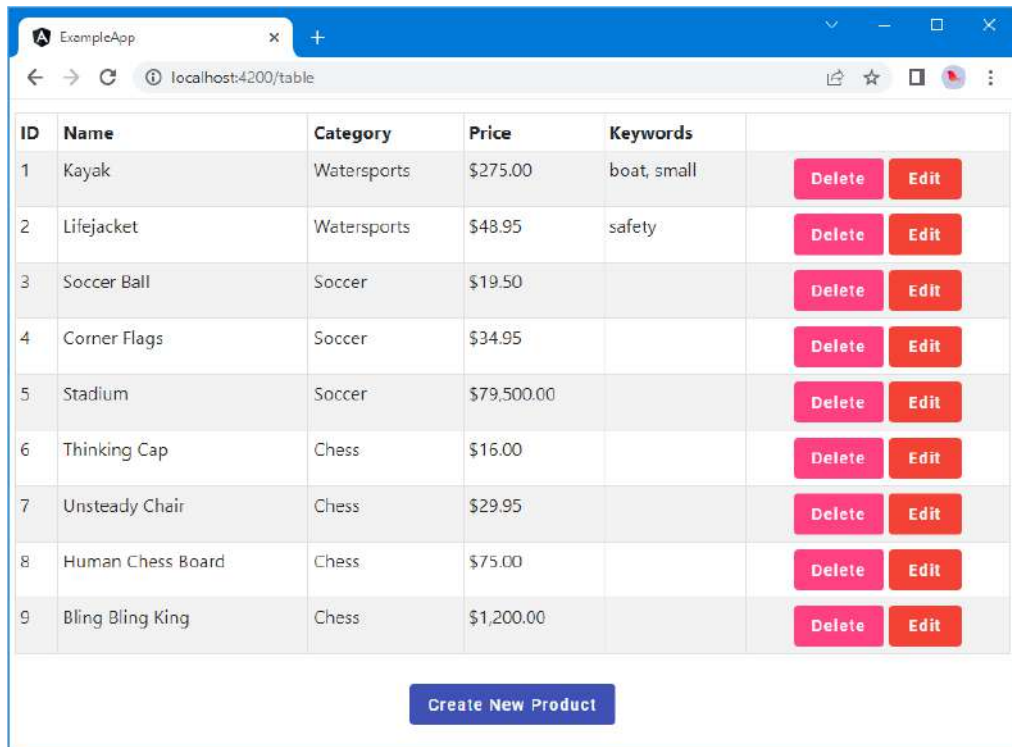


Figure 28.2. Using a component library

The Angular Material button feature is simple, but it shows the basic pattern to follow when using a component from a package: apply the component, add some tuning CSS styles, and import the feature module.

NOTE Styles provided by the Bootstrap CSS package are still being used in the example, with classes such as `p-2` and `text-center`, which are used to center content and add

padding. Most projects will use a single package, but Bootstrap and Angular Material will coexist.

28.2.2 Using the Angular Material table

Buttons are relatively simple, and the main benefit of using the Angular Material button is consistency. Other components are more complex and provide more features, such as tables. Listing 28.8 removes the Bootstrap CSS styles from the table that displays product details and introduces the Angular Material table feature.

Listing 28.8. Changing the table in the table.component.html file in the src/app/core folder

```
<table mat-table [dataSource]="Products()">

  <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 item">
      <button mat-flat-button color="accent"
        (click)="deleteProduct(item.id)">
        Delete
      </button>
      <button mat-flat-button color="warn"
        [routerLink]="['/form', 'edit', item.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="p-2 text-center">
  <button mat-flat-button color="primary" routerLink="/form/create"
    [disabled]="isServer">
    Create New Product
  </button>
</div>
```

Angular Material tables are created by applying the `mat-table` attribute to a table element and creating a `dataSource` data binding that selects an array of values to display.

Angular Material focuses on defining columns to describe the contents of a table. A `mat-text-column` element is used for simple columns, where the column header is the name of the data property and the value is displayed without modification, like this:

```
...
<mat-text-column name="id"></mat-text-column>
...
```

The `name` attribute selects the property to be displayed and sets the name by which the column is identified. For more complex columns, the `matColumnDef` attribute is applied to an `ng-container` element that contains `th` and `td` elements that are included in the table head and body, respectively:

```
...
<ng-container matColumnDef="price">
  <th mat-header-cell *matHeaderCellDef>Price</th>
  <td mat-cell *matCellDef="let item"> {{item.price | currency:"USD"}}
</td>
</ng-container>
...
```

The `th` element is given the `mat-header-cell` attribute, and the concise syntax is used to apply the `matHeaderCellDef` directive. The `td` element is given the `mat-cell` attribute, and the `matCellDef` directive is used to create an expression that selects the data used to create the contents of a table cell. There is an implicit value that provides the current data value, and, for the `price` column, this is formatted as a currency value using a pipe. This approach allows data values to be formatted or composed from multiple data source properties.

If you jump directly to using a component library without taking the time to understand how Angular works, the steps required to set up complex features can be impenetrable. But the knowledge you gained in earlier chapters helps reveal how the Angular Material table works, using features such as the concise directive syntax and implicit values to map the data in the data source to the content in the column descriptions.

The next step is to define the templates for the header and body rows, like this:

```
...
<tr mat-header-row *matHeaderRowDef="colsAndRows"></tr>
<tr mat-row *matRowDef="let row; columns: colsAndRows"></tr>
...
```

Columns are not shown unless they are configured with a row template, which is an array containing the names assigned to the columns. Listing 28.9 adds a property to the component class to select all of the columns defined in listing 28.8.

Listing 28.9. Selecting columns in the `table.component.ts` file in the `src/app/core` folder

```
import { Component, Input, Signal, computed } from "@angular/core";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";
import { PlatformService } from "../platform.service";

@Component({
  selector: "paTable",
  templateUrl: "table.component.html"
})
export class TableComponent {
```

```

    constructor(private model: Model, private ps: PlatformService) { }

    @Input()
    category?: string

    getProduct(key: number): Product | undefined {
        return this.model.getProduct(key);
    }

    get Products(): Signal<Product[]> {
        return computed(() => {
            return this.model.Products().filter(p =>
                this.category == null || p.category == this.category);
        });
    }

    get Categories(): Signal<string[]> {
        return computed(() => {
            return this.model.Products()
                .map(p => p.category)
                .filter((c, index, arr) => c != undefined
                    && arr.indexOf(c) == index) as string[];
        })
    }

    deleteProduct(key?: number) {
        if (key != undefined) {
            this.model.deleteProduct(key);
        }
    }

    get isServer() { return this.ps.isServer }

    colsAndRows: string[] = ['id', 'name', 'category', 'price', 'buttons'];
}

```

The next step is to define CSS styles that will supplement those used by Angular Material and style the table, as shown in listing 28.10.

Listing 28.10. 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; }

button[mat-flat-button] { margin: 2px; }

table[mat-table] { width: 100%; }
th[mat-header-cell] { font-size: large; font-weight: bold;}
td.mat-column-price { font-style: italic;}

```

The table element is decorated with the `mat-table` attribute, which allows me to set the width of the table. Similarly, the table header cells are decorated with the `mat-header-cell` attribute, which I use to change the font.

As the Angular Material generates the content for the table body, elements are added to classes that indicate which column a cell belongs to so that cells for the `price` column, for

example, are added to the `mat-column-price` class. I use this class to italicize the values in the `price` column.

To finish up applying the Angular Material table, listing 28.11 imports the module that contains the table features.

Listing 28.11. Importing the component module in the `material.module.ts` file in the `src/app` folder

```
import { NgModule } from "@angular/core";
import { MatButtonModule } from "@angular/material/button";
import { MatTableModule } from "@angular/material/table";

const features = [MatButtonModule, MatTableModule];

@NgModule({
  imports: [features],
  exports: [features]
})
export class MaterialFeatures {}
```

When the application reloads, the Angular Material features are used to generate the table content, as shown in figure 28.3.

Id	Name	Category	Price	
1	Kayak	Watersports	<i>\$275.00</i>	Delete Edit
2	Lifejacket	Watersports	<i>\$48.95</i>	Delete Edit
3	Soccer Ball	Soccer	<i>\$19.50</i>	Delete Edit
4	Corner Flags	Soccer	<i>\$34.95</i>	Delete Edit
5	Stadium	Soccer	<i>\$79,500.00</i>	Delete Edit
6	Thinking Cap	Chess	<i>\$16.00</i>	Delete Edit
7	Unsteady Chair	Chess	<i>\$29.95</i>	Delete Edit
8	Human Chess Board	Chess	<i>\$75.00</i>	Delete Edit
9	Bling Bling King	Chess	<i>\$1,200.00</i>	Delete Edit

Create New Product

Figure 28.3. Using the Angular Material table

USING THE BUILT-IN TABLE FEATURES

The basic table features don't offer much beyond the code with which I started the chapter. But one of the reasons for using a component library is to take advantage of features that are provided by the library authors, which you would otherwise have to write yourself.

The Angular Material table has some useful capabilities, including integrated support for paginating and sorting data. In listing 28.12, I have declared dependencies on the Angular Material modules that provide these features, along with the module that contains animation support, which the table features use.

Listing 28.12. Adding dependencies in the material.module.ts file in the src/app folder

```
import { NgModule } from "@angular/core";
import { MatButtonModule } from "@angular/material/button";
import { MatTableModule } from "@angular/material/table";
import { MatPaginatorModule } from "@angular/material/paginator";
import { MatSortModule } from "@angular/material/sort";
import { BrowserAnimationsModule }
  from "@angular/platform-browser/animations";

const features = [MatButtonModule, MatTableModule,
  MatPaginatorModule, MatSortModule];

@NgModule({
  imports: [features, BrowserAnimationsModule],
  exports: [features]
})
export class MaterialFeatures {}
```

Listing 28.13 changes the template that displays the table to add support for sorting data by the price column and for paginating data.

Listing 28.13. Enhancing the table in the table.component.html file in the src/app/core folder

```
<table mat-table [dataSource]="DataSource" matSort>

  <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 mat-sort-header>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 item">
      <button mat-flat-button color="accent">
```

```

                (click)="deleteProduct(item.id)">
                Delete
            </button>
            <button mat-flat-button color="warn"
                [routerLink]="['/form', 'edit', item.id]">
                Edit
            </button>
        </ng-container>

        <tr mat-header-row *matHeaderRowDef="colsAndRows"></tr>
        <tr mat-row *matRowDef="let row; columns: colsAndRows"></tr>
    </table>

    <mat-paginator [pageSize]="5" [pageSizeOptions]="[3, 5, 10]">
</mat-paginator>

    <div class="p-2 text-center">
        <button mat-flat-button color="primary" routerLink="/form/create"
            [disabled]="isServer">
            Create New Product
        </button>
    </div>

```

The `matSort` attribute is applied to the table element, and the `mat-sort-header` attribute is added to headers that will allow the user to sort data. The `mat-paginator` component displays pagination controls for the table data.

The final step is to create a data source that supports sorting and pagination and that is populated with data through the observable exposed by the repository, as shown in listing 28.14.

Listing 28.14. Creating a data source in the `table.component.ts` file in the `src/app/core` folder

```

import { Component, Input, Signal, ViewChild, computed, effect }
  from "@angular/core";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";
import { PlatformService } from "../platform.service";
import { MatTableDataSource } from "@angular/material/table";
import { MatPaginator } from "@angular/material/paginator";
import { MatSort } from "@angular/material/sort";

@Component({
  selector: "paTable",
  templateUrl: "table.component.html"
})
export class TableComponent {

  constructor(private model: Model, private ps: PlatformService) {
    this.DataSource = new MatTableDataSource<Product>();
    effect(() => {
      this.DataSource.data = this.model.Products()
    })
  }

  DataSource: MatTableDataSource<Product>;

```



```

@Input()
category?: string

// getProduct(key: number): Product | undefined {
//   return this.model.getProduct(key);
// }

// get Products(): Signal<Product[]> {
//   return computed(() => {
//     return this.model.Products().filter(p =>
//       this.category == null || p.category == this.category);
//   });
// }

// get Categories(): Signal<string[]> {
//   return computed(() => {
//     return this.model.Products()
//       .map(p => p.category)
//       .filter((c, index, arr) => c != undefined
//         && arr.indexOf(c) == index) as string[];
//   })
// }

deleteProduct(key?: number) {
  if (key != undefined) {
    this.model.deleteProduct(key);
  }
}

get isServer() { return this.ps.isServer }

colsAndRows: string[] = ['id', 'name', 'category', 'price', 'buttons'];

@ViewChild(MatPaginator) paginator!: MatPaginator;
@ViewChild(MatSort) sort!: MatSort;

ngAfterViewInit() {
  this.DataSource.paginator = this.paginator;
  this.DataSource.sort = this.sort;
}
}

```

The `MatTableDataSource<Product>` object represents a data source for `Product` objects, and its `data` property is used to update the data the table displays. The `paginator` and `sort` properties are used to associate the `MatPaginator` component and `MatSort` directive with the data source, which I do in the `ngAfterViewInit` method, to ensure that the child content is queried and assigned to the `ViewChild` properties. The result is that the data in the table is paginated and can be sorted by clicking the header for the Price column, as shown in figure 28.4.

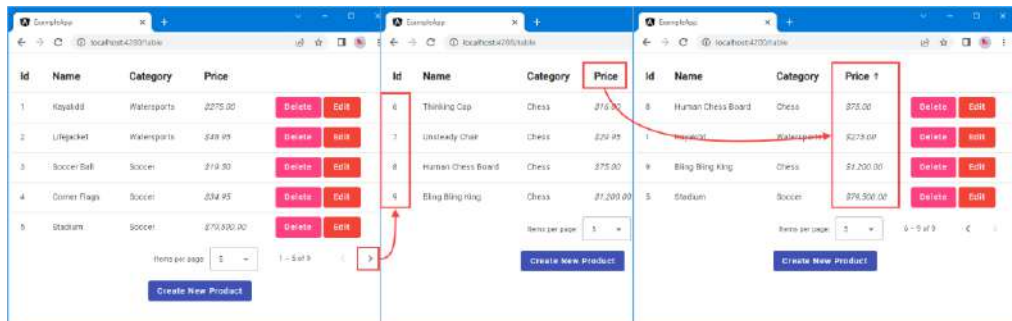


Figure 28.4. Using built-in table features

Writing pagination and sorting code isn't difficult—I demonstrated a simple paginator in the SportsStore application, for example—but using a component library means that you can rely on out-of-the-box features that are already tested. The trade-off is that you generally have to fit into a predefined model of how data will be expressed to get the most benefit, such as using the `MatTableDataSource<T>` class with Angular Material.

28.3 Matching the component library theme

You can get a long way using just the features provided by a good component library, but some projects require more specialized features, which leads to custom Angular directives or components.

Most component libraries provide access to the underlying themes they use to style content, often expressed using *Sass*, which is a superset of CSS that makes it easier to create complex sets of styles without endless duplication of CSS properties. Sass files have the `.scss` extension and are compiled when the project is built to generate standard CSS stylesheets that the browser can understand. (Confusingly, Sass also supports a closely related syntax in files with the `.sass` file extension. The history of CSS and attempts to improve it are long and complex and can be ignored.)

I am not going to describe Sass in detail in this book—see <https://sass-lang.com> for full details—but I will explain the features that are required to style custom components with the Angular Material theme.

28.3.1 Creating the custom component

I am going to create a custom button component, which will let me show the use of themes without getting bogged down in the component itself. Add a file named `customButton.component.ts` to the `src/app/core` folder, with the content shown in listing 28.15.

Listing 28.15. The `custombutton.component.ts` file in the `src/app/core` folder

```
import { Component, ElementRef, Input, ViewChild } from "@angular/core";
```

```

@Component({
  selector: "customButton",
  templateUrl: "customButton.component.html"
})
export class CustomButton {

  @Input("themeColor")
  themeColor: string = "primary"

  @ViewChild("buttonTarget")
  button?: ElementRef

  ngAfterViewInit() {
    this.button?.nativeElement.classList
      .add(`custom-button-${this.themeColor}`);
  }
}

```

The component queries its template to locate a `button` element and assigns it to a class based on the value received through an input property.

To define the template for the component, add a file named `customButton.component.html` to the `src/app/core` folder with the content shown in listing 28.16.

Listing 28.16. The `custombutton.component.html` file in the `src/app/core` folder

```

<button #buttonTarget>
  <ng-content></ng-content>
</button>

```

Listing 28.17 adds the custom button component to the `core` module.

Listing 28.17. Adding the Component to the Module in the `core.module.ts` File in the `src/app/core` Folder

```

import { NgModule } from "@angular/core";
import { BrowserModule } from "@angular/platform-browser";
import { FormsModule, NG_VALIDATORS, ReactiveFormsModule }
  from "@angular/forms";
import { ModelModule } from "../model/model.module";
import { TableComponent } from "../table.component";
import { FormComponent } from "../form.component";
//import { SharedState } from "../sharedState.service";
import { ValidationHelper } from "../validationHelper.pipe";
import { ValidationErrorsDirective } from "../validationErrors.directive";
import { HiLowValidatorDirective } from "../validation/hiLow";
import { RouterModule } from "@angular/router";
import { ProductCountComponent } from "../productCount.component";
import { CategoryCountComponent } from "../categoryCount.component";
import { NotFoundComponent } from "../notFound.component";
import { UnsavedGuard } from "../unsaved.guard";
import { MaterialFeatures } from "../material.module";
import { CustomButton } from "../customButton.component";

@NgModule({
  imports: [BrowserModule, FormsModule, ModelModule,
    ReactiveFormsModule, RouterModule, MaterialFeatures],

```

```

    declarations: [TableComponent, FormComponent, ValidationHelper,
                  ValidationErrorsDirective, HiLowValidatorDirective,
                  ProductCountComponent, CategoryCountComponent,
                  NotFoundComponent, CustomButton],
    exports: [ModelModule, TableComponent, FormComponent],
    providers: [UnsavedGuard]
  })
  export class CoreModule { }

```

The final preparatory step is to use the new component, as shown in listing 28.18.

Listing 28.18. Applying the custom component in the `table.component.html` file in the `src/app/core` folder

```

<table mat-table [dataSource]="DataSource" matSort>

  <!-- ...elements omitted for brevity... -->

</table>

<mat-paginator [pageSize]="5" [pageSizeOptions]="[3, 5, 10]">
</mat-paginator>

<div class="p-2 text-center">
  <button mat-flat-button color="primary" routerLink="/form/create"
    [disabled]="isServer">
    Create New Product
  </button>
  <customButton themeColor="primary" routerLink="/form/create">
    Create New Product
  </customButton>
</div>

```

Save the changes, and you will see the new (unstyled) button shown alongside the standard Angular Material button, as shown in figure 28.5.

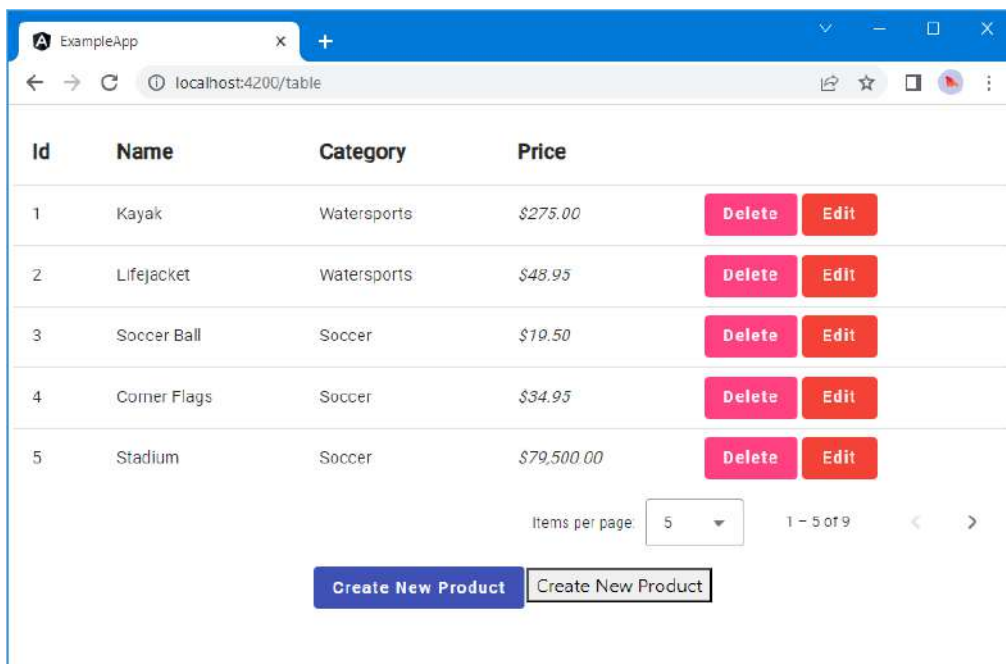


Figure 28.5. Applying a custom component

28.3.2 Using the Angular Material theme

The Angular build tools have integrated support for working with SCSS files, which is the file format used by SASS. Add a file named `customButton.component.scss` to the `src/app/core` folder with the content shown in listing 28.19.

Figuring out theme details

An investment of time is required to figure out how to apply the Angular Material themes to custom components, so do not rush into this process expecting it to be quick and easy.

To figure out how to create the styles I needed for the button component, I relied on the Angular Material theme documentation (<https://material.angular.io/guide/theming-your-components>) and the Material Design theme description (<https://material.io/design/material-theming/overview.html>), both of which contain useful guidance. But I spent most of the time reading through the SCSS files in the Angular Material package (<https://github.com/angular/components>) to figure out the purpose of different functions and to understand how the styles for the built-in components are generated. It was also helpful to use the browser F12 developer tools to see how HTML elements are styled.

But don't be put off. Once you have worked your way through the process for one component, you will have learned enough to make subsequent components much simpler.

Listing 28.19. The customButton.component.scss file in the src/app/core folder

```
@use "@angular/material" as material;

$primary: material.define-palette(material.$indigo-palette);
$accent: material.define-palette(material.$pink-palette, A200, A100, A400);
$warn: material.define-palette(material.$red-palette);

$typography: material.define-typography-config();

button[class*="custom-button-"] {
  padding: 7px 12px;
  border: none;
  border-radius: 4px;
  margin: 2px;
}

button.custom-button {
  @each $name, $palette in (primary: $primary,
    accent: $accent, warn: $warn) {
    &-#{ $name } {
      background-color:
        material.get-color-from-palette($palette, default);
      color: material.get-color-from-palette($palette,
        default-contrast);
      font: {
        family: material.font-family($typography, button);
        size: material.font-size($typography, button);
        weight: material.font-weight($typography, button);
      }
    }
  }
}

$bg: material.$light-theme-background-palette;
$fg: material.$light-theme-foreground-palette;

:host[disabled] button[class*="custom-button-"],
  button[class*="custom-button-"]:disabled {
  background-color:
    material.get-color-from-palette($bg, disabled-button);
  color: material.get-color-from-palette($fg, disabled-button);
}
```

Sass has a concise syntax, which can make it difficult to understand what is happening in the listing until you have at least a little experience. The first statement is an `@use` expression:

```
...
@use "@angular/material" as material;
...
```

Sass support function and variables, which can be used to generate CSS styles, and the `@use` expression provides access to the Sass features that Angular Material provides. The next group

of statements create the primary, accent, and warn palettes from the Angular Material theme:

```
...
$primary: material.define-palette(material.$indigo-palette);
$accent: material.define-palette(material.$pink-palette, A200, A100, A400);
$warn: material.define-palette(material.$red-palette);
...
```

Angular Material defines a set of base palettes, which contain a range of hues for a single color. The term `material.$indigo-palette`, for example, refers to the set of indigo hues. (The `material` prefix was specified in the `@use` expression and allows me to access Angular Material Sass features, and the `$` sign indicates a variable so that `material.$indigo-palette` refers to a variable named `indigo-palette` defined by the Angular Material package.) The `define-palette` function is used to select specific hues and give them convenient names, such as `default` and `text`, which help ensure consistency when applying styles. The indigo and pink palettes correspond to the default theme, which was chosen when the Angular Material package was installed. If you select a different theme for a project, then you will need to use the palettes that correspond to the colors of that theme.

The next step is to get the font configuration that Angular Material applies to its components:

```
...
$typography: material.define-typography-config();
...
```

The `define-typography-config` function returns a map where the keys are the names of styles that can be applied to text. A complete list of these styles can be found at <https://material.angular.io/guide/typography>, but the style name I want for this example is `button`, which provides the font settings for buttons.

Not all of the styles applied to buttons are specific to a palette, and I have used a selector that will match all of the palette-specific classes to apply these styles:

```
...
button[class*="custom-button-"] {
  padding: 7px 12px;
  border: none;
  border-radius: 4px;
  margin: 2px;
}
...
```

The next expression is the most complex and is responsible for generating the styles that are specific to a palette:

```
...
button.custom-button {
  @each $name, $palette in (primary: $primary,
    accent: $accent, warn: $warn) {
    &-#{$name} {
      background-color:
        material.get-color-from-palette($palette, default);
      color: material.get-color-from-palette($palette,
        default-contrast);
      font: {
        family: material.font-family($typography, button);
      }
    }
  }
}
```

```

        size: material.font-size($typography, button);
        weight: material.font-weight($typography, button);
    }
}
}
...

```

The effect of this expression is to generate a style for each of the primary, accent, and warn palettes, which contains background-color, color, and font properties that are specific to each palette. The `get-color-from-palette` function is used to get a color from a palette, either by hue or by using one of the names created by the `define-palette` function. The name `default` refers to the default color, and the `default-contrast` name refers to a color that can be used for text:

```

...
background-color: material.get-color-from-palette($palette, default);
color: material.get-color-from-palette($palette, default-contrast);
...

```

The values for the font properties are obtained using the `font-family`, `font-size`, and `font-weight` functions, which read values from the typography configuration settings.

Two more palettes are required to deal with disabled buttons:

```

...
$bg: material.$light-theme-background-palette;
$fg: material.$light-theme-foreground-palette;
...

```

The themes that Angular Material provides are categorized as either *light* or *dark*, and there are additional palettes of foreground and background colors that are shared by these light and dark themes, such as the colors for disabled buttons. The default indigo/pink theme is light, so I have assigned the light theme palettes to variables named `fg` and `bg`. These palettes are used to create a style that is applied to disabled buttons:

```

...
:host[disabled] button[class*="custom-button-"],
  button[class*="custom-button-"]:disabled {
    background-color: material.get-color-from-palette($bg, disabled-
button);
    color: material.get-color-from-palette($fg, disabled-button);
  }
...

```

The foreground and background palettes contain colors named `disabled-button`, which are used to set the `background-color` and `color` properties when a button is disabled. The selector matches button elements that are disabled or whose host element is disabled. The `:host` selector is required by the Angular view encapsulation feature and allows the component to be disabled by applying the `disabled` attribute to the `customButton` element.

SCSS files are applied to the component in just the same way as regular CSS files, as shown in listing 28.20.

Listing 28.20. Applying styles in the `customButton.component.ts` file in the `src/app/core` folder

```

import { Component, ElementRef, Input, ViewChild } from "@angular/core";

```



```

@Component({
  selector: "customButton",
  templateUrl: "customButton.component.html",
  styleUrls: ["customButton.component.scss"]
})
export class CustomButton {

  @Input("themeColor")
  themeColor: string = "primary"

  @ViewChild("buttonTarget")
  button?: ElementRef

  ngAfterViewInit() {
    this.button?.nativeElement.classList
      .add(`custom-button-${this.themeColor}`);
  }
}

```

During the build process, the SCSS files are processed to generate CSS files that can be sent to the browser. Figure 28.6 shows the built-in Angular Material button alongside the styles custom component.

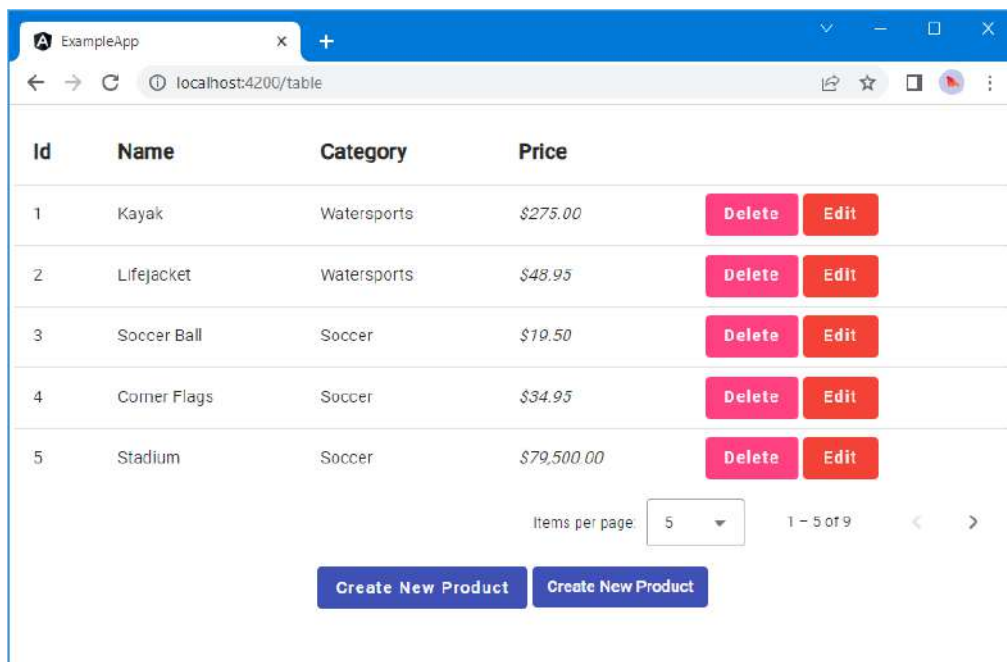


Figure 28.6. Applying a theme to a custom component

28.3.3 Applying the ripple effect

To finish off this chapter, I am going to add an animation effect to my custom button. Angular Material includes a ripple effect that is used to highlight user interaction, such as when a button is clicked. It is difficult to show this on a printed page, but figure 28.7 gives an idea of how the color of a built-in Angular Material button is progressively changed when it is clicked.



Figure 28.7. The Angular Material button ripple effect

When the user clicks a button, a circle of lighter color spreads out from the pointer. The best way to see this effect is to hold the mouse button down because the animation will be terminated when the mouse is released.

Angular Material makes the ripple feature available as a directive that can be applied to any component. Listing 28.21 imports the ripple module from the Angular Material package.

Listing 28.21. Adding a dependency in the material.module.ts file in the src/app folder

```
import { NgModule } from "@angular/core";
import { MatButtonModule } from "@angular/material/button";
import { MatTableModule } from "@angular/material/table";
import { MatPaginatorModule } from "@angular/material/paginator";
import { MatSortModule } from "@angular/material/sort";
import { BrowserAnimationsModule }
  from "@angular/platform-browser/animations";
import { MatRippleModule } from "@angular/material/core";

const features = [MatButtonModule, MatTableModule,
  MatPaginatorModule, MatSortModule, MatRippleModule];

@NgModule({
  imports: [features, BrowserAnimationsModule],
  exports: [features]
})
export class MaterialFeatures {}
```

Listing 28.22 applies the ripple directive to the button element in the custom component's template.

Listing 28.22. Applying a ripple in the custombutton.component.html file in the src/core/app folder

```
<button #buttonTarget matRipple>
  <ng-content></ng-content>
</button>
```

Ripples work by adding a `div` element inside the button, to which the animation is applied. The color used for the ripple is derived from the palette used for the button, as shown in listing 28.23.

Listing 28.23. Defining a style in the customButton.component.scss file in the src/app/core folder

```
@use "@angular/material" as material;

$primary: material.define-palette(material.$indigo-palette);
$accent: material.define-palette(material.$pink-palette, A200, A100, A400);
$warn: material.define-palette(material.$red-palette);

$typography: material.define-typography-config();

button[class*="custom-button-"] {
  padding: 7px 12px;
  border: none;
  border-radius: 4px;
  margin: 2px;
}

button.custom-button {
  @each $name, $palette in (primary: $primary,
    accent: $accent, warn: $warn) {
    &-#{$name} {
      background-color:
        material.get-color-from-palette($palette, default);
      color: material.get-color-from-palette($palette,
        default-contrast);
      font: {
        family: material.font-family($typography, button);
        size: material.font-size($typography, button);
        weight: material.font-weight($typography, button);
      }
    }

    &-#{$name} ::ng-deep .mat-ripple-element {
      background-color: material.get-color-from-palette($palette,
        default-contrast, 0.1);
    }
  }
}

$bg: material.$light-theme-background-palette;
$fg: material.$light-theme-foreground-palette;

:host[disabled] button[class*="custom-button-"],
  button[class*="custom-button-"]:disabled {
  background-color:
    material.get-color-from-palette($bg, disabled-button);
  color: material.get-color-from-palette($fg, disabled-button);
}
```

The `::ng-deep` pseudoclass is used to prevent Angular from modifying the name of the `mat-ripple-element` class selector for view encapsulation. (The `/deep/` and `>>>` selectors are not supported by Sass.) The `get-color-from-palette` function is used to get a color from the chosen palette with an opacity value, which was chosen to match the one used by the built-in Angular Material button feature. The result is that the custom button displays a ripple when clicked, as shown in figure 28.8.



Figure 28.8. Applying a ripple effect to a custom component

28.4 Summary

In this chapter, I demonstrated how a component library such as Angular Material can be introduced into a project to supplement or replace custom components. I also explained how the theme provided by Angular Material can be applied to custom components to ensure consistency across the application.

- Component libraries provide Angular functionality in a format that is easy to install and use.
- Angular Material is the most popular component library, but there are others available, including alternative UI frameworks.
- Most component libraries can be extended or blended with other functionality, to adapt to the needs of each project, although this can involve digging around in the source code and exploring the HTML generated by the library.

In the next chapter, I explain how to perform unit testing in an Angular project.

29

Angular unit testing

This chapter covers

- Creating and performing unit tests in Angular projects
- Isolating components and directives for unit testing
- Testing Angular features, such as input and output properties
- Testing the HTML content generated by components

In this chapter, I describe the tools that Angular provides for unit testing components and directives. Some Angular building blocks, such as pipes and services, can be readily tested in isolation using the basic testing tools that I set up at the start of the chapter. Components (and, to a lesser extent, directives) have complex interactions with their host elements and with their template content and require special features. Table 29.1 puts Angular unit testing in context.

DECIDING WHETHER TO UNIT TEST

Unit testing is a contentious topic. This chapter assumes you do want to do unit testing and shows you how to set up the tools and apply them to Angular components and directives. It isn't an introduction to unit testing, and I make no effort to persuade skeptical readers that unit testing is worthwhile. If you would like an introduction to unit testing, then there is a good article here: https://en.wikipedia.org/wiki/Unit_testing.

I like unit testing, and I use it in my projects—but not all of them and not as consistently as you might expect. I tend to focus on writing unit tests for features and functions that I know will be hard to write and likely to be the source of bugs in deployment. In these situations, unit testing helps structure my thoughts about how to best implement what I need. I find that just thinking about what I need to test helps produce ideas about potential problems, and that's before I start dealing with actual bugs and defects.

That said, unit testing is a tool and not a religion, and only you know how much testing you require. If you don't find unit testing useful or if you have a different methodology that suits you better, then don't feel you need to unit test just because it is fashionable. (However, if you don't have a better methodology and you are not testing at all, then you are probably letting users find your bugs, which is rarely ideal.)

Table 29.1. Putting Angular unit testing in context

Question	Answer
What is it?	Angular components and directives require special support for testing so that their interactions with other parts of the application infrastructure can be isolated and inspected.
Why is it useful?	Isolated unit tests can assess the basic logic provided by the class that implements a component or directive but do not capture the interactions with host elements, services, templates, and other important Angular features.
How is it used?	Angular provides a test bed that allows a realistic application environment to be created and then used to perform unit tests.
Are there any pitfalls or limitations?	Like much of Angular, the unit testing tools are complex. It can take some time and effort to get to the point where unit tests are easily written and run and you are sure that you have isolated the correct part of the application for testing.
Are there any alternatives?	As noted, you don't have to unit test your projects. But if you do want to unit testing, then you will need to use the Angular features described in this chapter.

Table 29.2 summarizes the chapter.

Table 29.2. Chapter summary

Problem	Solution	Listing
Performing a basic test on a component	Initialize a test module and create an instance of the component. If the component has an external template, an additional compilation step must be performed.	1–10, 12–14
Testing a component's data bindings	Use the <code>DebugElement</code> class to query the component's template.	11

Testing a component's response to events	Trigger the events using the debug element.	15–17
Testing a component's output properties	Subscribe to the <code>EventEmitter</code> created by the component.	18, 19
Testing a component's input properties	Create a test component whose template applies the component under test.	20, 22
Testing a directive	Create a test component whose template applies the directive under test.	22, 23

29.1 Preparing the example project

I continue to use the `exampleApp` project from earlier chapters, but I only need a simple target to focus on for unit testing. Add a file named `simple.component.ts` to the `src/app` folder, with the content shown in listing 29.1.

TIP You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/manningbooks/pro-angular-16>. See chapter 1 for how to get help if you have problems running the examples.

Listing 29.1. The contents of the `simple.component.ts` file in the `src/app` folder

```
import { Component } from "@angular/core";

@Component({
  selector: "simple",
  template: `<div class="bg-primary text-white p-2">Unit testing</div>`
})
export class SimpleComponent { }
```

Listing 29.2 adds the new component to the application module.

Listing 29.2. Registering a component in the `app.module.ts` file in the `src/app` folder

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

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

import { ModelModule } from "../model/model.module";
import { CoreModule } from "../core/core.module";
import { TableComponent } from "../core/table.component";
import { FormComponent } from "../core/form.component";
import { MessageModule } from "../messages/message.module";
import { MessageComponent } from "../messages/message.component";
import { AppComponent } from './app.component';
import { routing } from './app.routing';
import { TermsGuard } from './terms.guard';
import { LoadGuard } from './load.guard';
import { PlatformService } from './platform.service';
```

```
import { BrowserGuard } from './browser.guard';
import { SimpleComponent } from './simple.component';

@NgModule({
  declarations: [AppComponent, SimpleComponent],
  imports: [BrowserModule, ModelModule, CoreModule, MessageModule,
    routing],
  providers: [TermsGuard, LoadGuard, provideClientHydration(),
    PlatformService, BrowserGuard],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Listing 29.3 simplifies the routing configuration so that the new component is always displayed. This is not a requirement for unit testing, but it helps simplify the project for this chapter.

Listing 29.3. Changing the routing configuration in the `app.routing.ts` file in the `src/app` folder

```
import { Routes, RouterModule } from "@angular/router";
import { NotFoundComponent } from "../core/notFound.component";
import { SimpleComponent } from "../simple.component";

const routes: Routes = [
  { path: "", component: SimpleComponent },
  { path: "**", component: NotFoundComponent } ]

export const routing = RouterModule.forRoot(routes, {
  bindToComponentInputs: true
});
```

Open a new command prompt, navigate to the `exampleApp` folder, and run the following command to start the server that provides the RESTful web server:

```
npm run json
```

The RESTful web service isn't used directly in this chapter, but running it prevents errors. Open a separate command prompt, navigate to the `exampleApp` folder, and run the following command to start the Angular development tools:

```
ng serve
```

Open a new browser window and navigate to `http://localhost:4200` to see the content shown in figure 29.1.

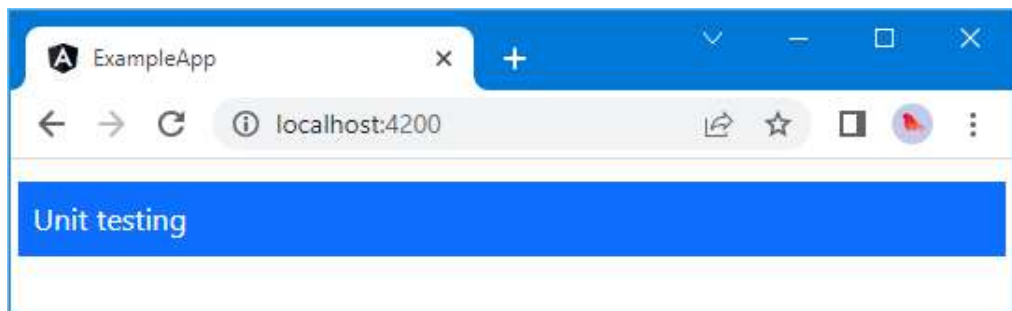


Figure 29.1. Running the example application

29.2 Running a simple unit test

When a new project is created using the `ng new` command, all the packages and tools required for unit testing are installed, based on the Jasmine test framework. To create a simple unit test to confirm that everything is working, I created the `src/app/tests` folder and added to it a file named `app.component.spec.ts` with the contents shown in listing 29.4. The naming convention for unit tests makes it obvious which file the tests apply to.

Listing 29.4. The contents of the `app.component.spec.ts` file in the `src/app/tests` folder

```
describe("Jasmine Test Environment", () => {
  it("is working", () => expect(true).toBe(true));
});
```

I explain the basics of working with the Jasmine API shortly, and you can ignore the syntax for the moment. Using a new command prompt, navigate to the `exampleApp` folder, and run the following command:

```
ng test
```

This command starts the Karma test runner, which opens a new browser tab with the content shown in figure 29.2.



Figure 29.2. Starting the Karma test runner

The browser window is used to run the tests, but the important information is written out to the command prompt used to start the test tools, where you will see a message like this:

```
Chrome 114.0.0.0 (Windows 10):
  Executed 1 of 1 SUCCESS (0.109 secs / 0.003 secs)
TOTAL: 1 SUCCESS
```

This shows that the single unit test in the project has been located and executed successfully. Whenever you make a change that updates one of the JavaScript files in the project, the unit tests will be located and executed, and any problems will be written to the command prompt. To show what an error looks like, listing 29.5 changes the unit test so that it will fail.

Listing 29.5. Making a unit test fail in the app.component.spec.ts file in the src/app/tests folder

```
describe("Jasmine Test Environment", () => {
  it("is working", () => expect(true).toBe(false));
});
```

This test will fail and will result in the following output, which indicates the test that has failed and what went wrong:

```
Chrome 114.0.0.0 (Windows 10) Jasmine Test Environment is working FAILED
  Expected true to be false.
    at <Jasmine>
    at UserContext.apply
      (src/app/tests/app.component.spec.ts:2:41)
    at _ZoneDelegate.invoke
      (node_modules/zone.js/fesm2015/zone.js:368:26)
    at ProxyZoneSpec.onInvoke
      (node_modules/zone.js/fesm2015/zone-testing.js:273:39)
    at _ZoneDelegate.invoke
      (node_modules/zone.js/fesm2015/zone.js:367:52)
Chrome 114.0.0.0 (Windows 10):
  Executed 1 of 1 (1 FAILED) (0.008 secs / 0.003 secs)
TOTAL: 1 FAILED, 0 SUCCESS
```

29.3 Working with Jasmine

The API that Jasmine provides chains together JavaScript methods to define unit tests. You can find the full documentation for Jasmine at <http://jasmine.github.io>, but table 29.3 describes the most useful functions for Angular testing.

Table 29.3. Useful Jasmine methods

Name	Description
describe(description, function)	This method is used to group a set of related tests.
beforeEach(function)	This method is used to specify a task that is performed before each unit test.
afterEach(function)	This method is used to specify a test that is performed after each unit test.
it(description, function)	This method is used to perform the test action.
expect(value)	This method is used to identify the result of the test.
toBe(value)	This method specifies the expected value of the test.

You can see how the methods in table 29.3 were used to create the unit test in listing 29.5.

```
...
describe("Jasmine Test Environment", () => {
  it("is working", () => expect(true).toBe(false));
});
...
```

You can also see why the test has failed since the `expect` and `toBe` methods have been used to check that `true` and `false` are equal. Since this cannot be the case, the test fails.

The `toBe` method isn't the only way to evaluate the result of a unit test. Table 29.4 shows other evaluation methods provided by Angular.

Table 29.4. Useful Jasmine evaluation methods

Name	Description
<code>toBe(value)</code>	This method asserts that a result is the same as the specified value (but need not be the same object).
<code>toEqual(object)</code>	This method asserts that a result is the same object as the specified value.
<code>toMatch(regex)</code>	This method asserts that a result matches the specified regular expression.
<code>toBeDefined()</code>	This method asserts that the result has been defined.
<code>toBeUndefined()</code>	This method asserts that the result has not been defined.
<code>toBeNull()</code>	This method asserts that the result is null.
<code>toBeTruthy()</code>	This method asserts that the result is truthy, as described in chapter 3.
<code>toBeFalsy()</code>	This method asserts that the result is falsy, as described in chapter 3.
<code>toContain(substring)</code>	This method asserts that the result contains the specified substring.
<code>toBeLessThan(value)</code>	This method asserts that the result is less than the specified value.
<code>toBeGreaterThan(value)</code>	This method asserts that the result is more than the specified value.

Listing 29.6 shows how these evaluation methods can be used in tests, replacing the failing test from the previous section.

Listing 29.6. Replacing the unit test in the `app.component.spec.ts` file in the `src/app/tests` folder

```
describe("Jasmine Test Environment", () => {
  it("test numeric value", () => expect(12).toBeGreaterThan(10));
  it("test string value", () => expect("London").toMatch("^Lon"));
});
```

When you save the changes to the file, the tests will be executed, and the results will be shown in the command prompt.

29.4 Testing an Angular component

The building blocks of an Angular application can't be tested in isolation because they depend on the underlying features provided by Angular and by the other parts of the project, including the services, directives, templates, and modules it contains. As a consequence, testing a building block such as a component means using testing utilities that are provided by Angular to re-create enough of the application to let the component function so that tests can be performed against it. In this section, I walk through the process of performing a unit test on the `SimpleComponent` component. This component is so simple that it doesn't have functionality of its own to test, but it is enough to demonstrate how the test process is applied.

29.4.1 Working with the `TestBed` class

At the heart of Angular unit testing is a class called `TestBed`, which is responsible for simulating the Angular application environment so that tests can be performed. Table 29.5 describes the most useful methods provided by the `TestBed` method, all of which are static.

Table 29.5. Useful `TestBed` methods

Name	Description
<code>configureTestingModule</code>	This method is used to configure the Angular testing module.
<code>createComponent</code>	This method is used to create an instance of the component.
<code>compileComponents</code>	This method is used to compile components, as described in the "Testing a Component with an External Template" section.

The `configureTestingModule` method is used to configure the Angular module that is used in testing, using the same properties supported by the `@NgModule` decorator. Just like in a real application, a component cannot be used in a unit test unless it has been added to the `declarations` property of the module. This means that the first step in most unit tests is to configure the testing module. To demonstrate, I added a file named `simple.component.spec.ts` to the `src/app/tests` folder with the content shown in listing 29.7.

Listing 29.7. The contents of the `simple.component.spec.ts` file in the `src/app/tests` folder

```
import { TestBed } from "@angular/core/testing";
import { SimpleComponent } from "../simple.component";

describe("SimpleComponent", () => {

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [SimpleComponent]
    });
  });

});
```

The `TestBed` class is defined in the `@angular/core/testing` module, and the `configureTestingModule` accepts an object whose `declarations` property tells the test module that the `SimpleComponent` class is going to be used.

TIP Notice that the `TestBed` class is used within the `beforeEach` function. You will receive an error if you try to use the `TestBed` outside of this function.

The next step is to create a new instance of the component so that it can be used in tests. This is done using the `createComponent` method, as shown in listing 29.8.

Listing 29.8. Instantiating a component in the `simple.component.spec.ts` file in the `src/app/tests` folder

```
import { TestBed, ComponentFixture } from "@angular/core/testing";
import { SimpleComponent } from "../simple.component";

describe("SimpleComponent", () => {

  let fixture: ComponentFixture<SimpleComponent>;
  let component: SimpleComponent;

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [SimpleComponent]
    });
    fixture = TestBed.createComponent(SimpleComponent);
    component = fixture.componentInstance;
  });

  it("is defined", () => {
    expect(component).toBeDefined()
  });
});
```

The argument to the `createComponent` method tells the test bed which component type it should instantiate, which is `SimpleComponent` in this case. The result is a `ComponentFixture<SimpleComponent>` object, which provides features for testing a component, using the methods and properties described in table 29.6.

Table 29.6. Useful `ComponentFixture` methods and properties

Name	Description
<code>componentInstance</code>	This property returns the component object.
<code>debugElement</code>	This property returns the test host element for the component.
<code>nativeElement</code>	This property returns the DOM object representing the host element for the component.

<code>detectChanges()</code>	This method causes the test bed to detect state changes and reflect them in the component's template.
<code>whenStable()</code>	This method returns a <code>Promise</code> that is resolved when the effect of an operation has been fully applied.

In the listing, I use the `componentInstance` property to get the `SimpleComponent` object that has been created by the test bed and perform a simple test to ensure that it has been created by using the `expect` method to select the `component` object as the target of the test and the `toBeDefined` method to perform the test. I demonstrate the other methods and properties in the sections that follow.

CONFIGURING THE TEST BED FOR DEPENDENCIES

One of the most important features of Angular applications is dependency injection, which allows components and other building blocks to receive services by declaring dependencies on them using constructor parameters. Listing 29.9 adds a dependency on the data model repository service to the `SimpleComponent` class.

Listing 29.9. Adding a service dependency in the `simple.component.ts` file in the `src/app` folder

```
import { Component } from "@angular/core";
import { Model } from "../model/repository.model";
import { Product } from "../model/product.model";

@Component({
  selector: "simple",
  template: `<div class="bg-primary text-white p-1">
    There are
    <span class="strong"> {{getProducts().length}} </span>
    products
  </div>`
})
export class SimpleComponent {

  constructor(private repository: Model) {}

  category: string = "Soccer";

  getProducts(): Product[] {
    return this.repository.Products()
      .filter(p => p.category == this.category);
  }
}
```

The component uses the repository to provide a filtered collection of `Product` objects, which are exposed through a method called `getProducts` and filtered using a `category` property. The inline template has a corresponding data binding that displays the number of products that the `getProducts` method returns.

Being able to unit test the component means providing it with a repository service. The Angular test bed will take care of resolving dependencies as long as they are configured

through the test module. Effective unit testing generally requires components to be isolated from the rest of the application, which means that mock or fake objects (also known as *test doubles*) are used as substitutes for real services in unit tests. Listing 29.10 configures the test bed so that a fake repository is used to provide the component with its service.

Listing 29.10. Providing a service in the `simple.component.spec.ts` file in the `src/app/tests` folder

```
import { TestBed, ComponentFixture } from "@angular/core/testing";
import { SimpleComponent } from "../simple.component";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";
import { signal } from "@angular/core";

describe("SimpleComponent", () => {

  let fixture: ComponentFixture<SimpleComponent>;
  let component: SimpleComponent;

  let mockRepository = {
    Products: signal([
      new Product(1, "test1", "Soccer", 100),
      new Product(2, "test2", "Chess", 100),
      new Product(3, "test3", "Soccer", 100)
    ])
  }

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [SimpleComponent],
      providers: [
        { provide: Model, useValue: mockRepository }
      ]
    });
    fixture = TestBed.createComponent(SimpleComponent);
    component = fixture.componentInstance;
  });

  it("filters categories", () => {
    component.category = "Chess"
    expect(component.getProducts().length).toBe(1);
    component.category = "Soccer";
    expect(component.getProducts().length).toBe(2);
    component.category = "Running";
    expect(component.getProducts().length).toBe(0);
  });
});
```

The `mockRepository` variable is assigned an object that provides a `Products` property that returns fixed data that can be used to test for known outcomes. To provide the component with the service, the `providers` property for the object passed to the `TestBed.configureTestingModule` method is configured in the same way as a real Angular module, using the value provider to resolve dependencies on the `Model` class using the `mockRepository` variable. The test invokes the component's `getProducts` method and

compares the results with the expected outcome, changing the value of the `category` property to check different filters.

29.4.2 Testing data bindings

The previous example showed how a component's properties and methods can be used in a unit test. This is a good start, but many components will also include small fragments of functionality in the data binding expressions contained in their templates, and these should be tested as well. Listing 29.11 checks that the data binding in the component's template correctly displays the number of products in the mock data model.

Listing 29.11. Testing a binding in the `simple.component.spec.ts` file in the `src/app/tests` folder

```
import { TestBed, ComponentFixture } from "@angular/core/testing";
import { SimpleComponent } from "../simple.component";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";
import { signal } from "@angular/core";
import { DebugElement } from "@angular/core";
import { By } from "@angular/platform-browser";

describe("SimpleComponent", () => {

  let fixture: ComponentFixture<SimpleComponent>;
  let component: SimpleComponent;
  let debugElement: DebugElement;
  let bindingElement: HTMLSpanElement;

  let mockRepository = {
    Products: signal([
      new Product(1, "test1", "Soccer", 100),
      new Product(2, "test2", "Chess", 100),
      new Product(3, "test3", "Soccer", 100)
    ])
  }

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [SimpleComponent],
      providers: [
        { provide: Model, useValue: mockRepository }
      ]
    });
    fixture = TestBed.createComponent(SimpleComponent);
    component = fixture.componentInstance;
    debugElement = fixture.debugElement;
    bindingElement = debugElement.query(By.css("span")).nativeElement;
  });

  it("filters categories", () => {
    component.category = "Chess"
    fixture.detectChanges();
    expect(component.getProducts().length).toBe(1);
    expect(bindingElement.textContent).toContain("1");
  });
});
```



```

    component.category = "Soccer";
    fixture.detectChanges();
    expect(component.getProducts().length).toBe(2);
    expect(bindingElement.textContent).toContain("2");

    component.category = "Running";
    fixture.detectChanges();
    expect(component.getProducts().length).toBe(0);
    expect(bindingElement.textContent).toContain("0");
  });
});

```

The `ComponentFixture.debugElement` property returns a `DebugElement` object that represents the root element from the component's template, and table 29.7 lists the most useful methods and properties described by the `DebugElement` class.

TIP If you don't see the test output, then restart the `ng test` command.

Table 29.7. Useful `DebugElement` properties and methods

Name	Description
<code>nativeElement</code>	This property returns the object that represents the HTML element in the DOM.
<code>children</code>	This property returns an array of <code>DebugElement</code> objects representing the children of this element.
<code>query(selectorFunction)</code>	The <code>selectorFunction</code> is passed a <code>DebugElement</code> object for each HTML element in the component's template, and this method returns the first <code>DebugElement</code> for which the function returns <code>true</code> .
<code>queryAll(selectorFunction)</code>	This is similar to the <code>query</code> method, except the result is all the <code>DebugElement</code> objects for which the function returns <code>true</code> .
<code>triggerEventHandler(name, event)</code>	This method triggers an event. See the "Testing Component Events" section for details.

Locating elements is done through the `query` and `queryAll` methods, which accept functions that inspect `DebugElement` objects and return `true` if they should be included in the results. The `By` class, defined in the `@angular/platform-browser` module, makes it easier to locate elements in the component's template through the static methods described in table 29.8.

Table 29.8. The `By` methods

Name	Description
------	-------------

<code>By.all()</code>	This method returns a function that matches any element.
<code>By.css(selector)</code>	This method returns a function that uses a CSS selector to match elements.
<code>By.directive(type)</code>	This method returns a function that matches elements to which the specified directive class has been applied, as demonstrated in the “Testing Input Properties” section.

In the listing, I use the `By.css` method to locate the first `span` element in the template and access the DOM object that represents it through the `nativeElement` property so that I can check the value of the `textContent` property in the unit tests.

Notice that after each change to the component’s `category` property, I call the `ComponentFixture` object’s `detectChanges` method, like this:

```
...
component.category = "Soccer";
fixture.detectChanges();
expect(component.getProducts().length).toBe(2);
expect(bindingElement.textContent).toContain("2");
...
```

This method tells the Angular testing environment to process any changes and evaluate the data binding expressions in the template. Without this method call, the change to the value of the `category` component would not be reflected in the template, and the test would fail.

29.4.3 Testing a component with an external template

Angular components are compiled into factory classes, either within the browser or by the ahead-of-time compiler that I demonstrated in chapter 8. As part of this process, Angular processes any external templates and includes them as text in the JavaScript code that is generated, similar to an inline template. When unit testing a component with an external template, the compilation step must be performed explicitly. In listing 29.12, I changed the `@Component` decorator applied to the `SimpleComponent` class so that it specifies an external template.

Listing 29.12. Specifying a template in the `simple.component.ts` file in the `src/app` folder

```
import { Component } from "@angular/core";
import { Model } from "../model/repository.model";
import { Product } from "../model/product.model";

@Component({
  selector: "simple",
  templateUrl: "simple.component.html"
})
export class SimpleComponent {

  constructor(private repository: Model) {}

  category: string = "Soccer";

  getProducts(): Product[] {
```

```

        return this.repository.Products()
            .filter(p => p.category == this.category);
    }
}

```

To provide the template, I created a file called `simple.component.html` in the `src/app/` folder and added the elements shown in listing 29.13.

Listing 29.13. The `simple.component.html` file in the `src/app` folder

```

<div class="bg-primary text-white p-2">
  There are
    <span class="strong"> {{getProducts().length}} </span>
  products
</div>

```

This is the same content that was previously defined inline. Listing 29.14 updates the unit test for the component to deal with the external template by explicitly compiling the component.

Listing 29.14. Compiling a component in the `simple.component.spec.ts` file in the `src/app/tests` folder

```

import { TestBed, ComponentFixture, waitForAsync }
    from "@angular/core/testing";
import { SimpleComponent } from "../simple.component";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";
import { signal } from "@angular/core";
import { DebugElement } from "@angular/core";
import { By } from "@angular/platform-browser";

describe("SimpleComponent", () => {

    let fixture: ComponentFixture<SimpleComponent>;
    let component: SimpleComponent;
    let debugElement: DebugElement;
    let bindingElement: HTMLSpanElement;

    let mockRepository = {
        Products: signal([
            new Product(1, "test1", "Soccer", 100),
            new Product(2, "test2", "Chess", 100),
            new Product(3, "test3", "Soccer", 100)
        ])
    };

    beforeEach(waitForAsync(() => {
        TestBed.configureTestingModule({
            declarations: [SimpleComponent],
            providers: [
                { provide: Model, useValue: mockRepository }
            ]
        });
        TestBed.compileComponents().then(() => {
            fixture = TestBed.createComponent(SimpleComponent);
            component = fixture.componentInstance;

```

```

        debugElement = fixture.debugElement;
        bindingElement
            = debugElement.query(By.css("span")).nativeElement;
    });
});

it("filters categories", () => {
    component.category = "Chess"
    fixture.detectChanges();
    expect(component.getProducts().length).toBe(1);
    expect(bindingElement.textContent).toContain("1");

    component.category = "Soccer";
    fixture.detectChanges();
    expect(component.getProducts().length).toBe(2);
    expect(bindingElement.textContent).toContain("2");

    component.category = "Running";
    fixture.detectChanges();
    expect(component.getProducts().length).toBe(0);
    expect(bindingElement.textContent).toContain("0");
});
});

```

Components are compiled using the `TestBed.compileComponents` method. The compilation process is asynchronous, and the `compileComponents` method returns a `Promise`, which must be used to complete the test setup when the compilation is complete. To make it easier to work with asynchronous operations in unit tests, the `@angular/core/testing` module contains a function called `waitForAsync`, which is used with the `beforeEach` method.

29.4.4 Testing component events

To demonstrate how to test for a component's response to events, I defined a new property in the `SimpleComponent` class and added a method to which the `@HostBinding` decorator has been applied, as shown in listing 29.15.

Listing 29.15. Adding event handling in the `simple.component.ts` file in the `src/app` folder

```

import { Component, HostListener } from "@angular/core";
import { Model } from "../model/repository.model";
import { Product } from "../model/product.model";

@Component({
    selector: "simple",
    templateUrl: "simple.component.html"
})
export class SimpleComponent {

    constructor(private repository: Model) {}

    category: string = "Soccer";
    highlighted: boolean = false;

```

```

getProducts(): Product[] {
    return this.repository.Products()
        .filter(p => p.category == this.category);
}

@HostListener("mouseenter", ['$event.type'])
@HostListener("mouseleave", ['$event.type'])
setHighlight(type: string) {
    this.highlighted = type == "mouseenter";
}
}

```

The `setHighlight` method has been configured so that it will be invoked when the host element's `mouseenter` and `mouseleave` events are triggered. Listing 29.16 updates the component's template so that it uses the new property in a data binding.

Listing 29.16. Binding to a property in the `simple.component.html` file in the `src/app` folder

```

<div class="bg-primary text-white p-2" [class.bg-success]="highlighted">
  There are
  <span class="strong"> {{getProducts().length}} </span>
  products
</div>

```

Events can be triggered in unit tests through the `triggerEventHandler` method defined by the `DebugElement` class, as shown in listing 29.17.

Listing 29.17. Triggering events in the `simple.component.spec.ts` file in the `src/app/tests` folder

```

import { TestBed, ComponentFixture, waitForAsync }
    from "@angular/core/testing";
import { SimpleComponent } from "../simple.component";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";
import { signal } from "@angular/core";
import { DebugElement } from "@angular/core";
import { By } from "@angular/platform-browser";

describe("SimpleComponent", () => {

    let fixture: ComponentFixture<SimpleComponent>;
    let component: SimpleComponent;
    let debugElement: DebugElement;
    //let bindingElement: HTMLSpanElement;
    let divElement: HTMLDivElement;

    let mockRepository = {
        Products: signal([
            new Product(1, "test1", "Soccer", 100),
            new Product(2, "test2", "Chess", 100),
            new Product(3, "test3", "Soccer", 100)
        ])
    }

    beforeEach(waitForAsync(() => {

```

```

TestBed.configureTestingModule({
  declarations: [SimpleComponent],
  providers: [
    { provide: Model, useValue: mockRepository }
  ]
});
TestBed.compileComponents().then(() => {
  fixture = TestBed.createComponent(SimpleComponent);
  component = fixture.componentInstance;
  debugElement = fixture.debugElement;
  // bindingElement
  //   = debugElement.query(By.css("span")).nativeElement;
  divElement = debugElement.children[0].nativeElement;
});
});

// it("filters categories", () => {
//   component.category = "Chess"
//   fixture.detectChanges();
//   expect(component.getProducts().length).toBe(1);
//   expect(bindingElement.textContent).toContain("1");

//   component.category = "Soccer";
//   fixture.detectChanges();
//   expect(component.getProducts().length).toBe(2);
//   expect(bindingElement.textContent).toContain("2");

//   component.category = "Running";
//   fixture.detectChanges();
//   expect(component.getProducts().length).toBe(0);
//   expect(bindingElement.textContent).toContain("0");
// });

it("handles mouse events", () => {
  expect(component.highlighted).toBeFalsy();
  expect(divElement.classList.contains("bg-success")).toBeFalsy();
  debugElement.triggerEventHandler("mouseenter",
    new Event("mouseenter"));
  fixture.detectChanges();
  expect(component.highlighted).toBeTruthy();
  expect(divElement.classList.contains("bg-success")).toBeTruthy();
  debugElement.triggerEventHandler("mouseleave",
    new Event("mouseleave"));
  fixture.detectChanges();
  expect(component.highlighted).toBeFalsy();
  expect(divElement.classList.contains("bg-success")).toBeFalsy();
});
});

```

The test in this listing checks the initial state of the component and the template and then triggers the `mouseenter` and `mouseleave` events, checking the effect that each has.

29.4.5 Testing output properties

Testing output properties is a simple process because the `EventEmitter` objects used to implement them are `Observable` objects that can be subscribed to in unit tests. Listing 29.18 adds an output property to the component under test.

Listing 29.18. Adding an output property in the `simple.component.ts` file in the `src/app` folder

```
import { Component, EventEmitter, HostListener, Output }
  from "@angular/core";
import { Model } from "../model/repository.model";
import { Product } from "../model/product.model";

@Component({
  selector: "simple",
  templateUrl: "simple.component.html"
})
export class SimpleComponent {

  constructor(private repository: Model) {}

  category: string = "Soccer";
  highlighted: boolean = false;

  @Output("pa-highlight")
  change = new EventEmitter<boolean>();

  getProducts(): Product[] {
    return this.repository.Products()
      .filter(p => p.category == this.category);
  }

  @HostListener("mouseenter", ["$event.type"])
  @HostListener("mouseleave", ["$event.type"])
  setHighlight(type: string) {
    this.highlighted = type == "mouseenter";
    this.change.emit(this.highlighted);
  }
}
```

The component defines an output property called `change`, which is used to emit an event when the `setHighlight` method is called. Listing 29.19 shows a unit test that targets the output property.

Listing 29.19. Testing an output property in the `simple.component.spec.ts` file in the `src/app/tests` folder

```
import { TestBed, ComponentFixture, waitForAsync }
  from "@angular/core/testing";
import { SimpleComponent } from "../simple.component";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";
import { signal } from "@angular/core";
import { DebugElement } from "@angular/core";
import { By } from "@angular/platform-browser";

describe("SimpleComponent", () => {

  let fixture: ComponentFixture<SimpleComponent>;
  let component: SimpleComponent;
  let debugElement: DebugElement;
  let divElement: HTMLDivElement;
```

```

let mockRepository = {
  Products: signal([
    new Product(1, "test1", "Soccer", 100),
    new Product(2, "test2", "Chess", 100),
    new Product(3, "test3", "Soccer", 100)
  ])
}

beforeEach(waitForAsync(() => {
  TestBed.configureTestingModule({
    declarations: [SimpleComponent],
    providers: [
      { provide: Model, useValue: mockRepository }
    ]
  });
  TestBed.compileComponents().then(() => {
    fixture = TestBed.createComponent(SimpleComponent);
    component = fixture.componentInstance;
    debugElement = fixture.debugElement;
    divElement = debugElement.children[0].nativeElement;
  });
}));

// it("handles mouse events", () => {
//   expect(component.highlighted).toBeFalsy();
//   expect(divElement.classList.contains("bg-success")).toBeFalsy();
//   debugElement.triggerEventHandler("mouseenter",
//     new Event("mouseenter"));
//   fixture.detectChanges();
//   expect(component.highlighted).toBeTruthy();
//   expect(divElement.classList
//     .contains("bg-success")).toBeTruthy();
//   debugElement.triggerEventHandler("mouseleave",
//     new Event("mouseleave"));
//   fixture.detectChanges();
//   expect(component.highlighted).toBeFalsy();
//   expect(divElement.classList.contains("bg-success")).toBeFalsy();
// });

it("implements output property", () => {
  let highlighted: boolean = false;
  component.change.subscribe(value => highlighted = value);
  debugElement.triggerEventHandler("mouseenter",
    new Event("mouseenter"));
  expect(highlighted).toBeTruthy();
  debugElement.triggerEventHandler("mouseleave",
    new Event("mouseleave"));
  expect(highlighted).toBeFalsy();
});
});

```

I could have invoked the component's `setHighlight` method directly in the unit test, but I have chosen to trigger the `mouseenter` and `mouseleave` events, which will activate the output property indirectly. Before triggering the events, I use the `subscribe` method to

receive the event from the `output` property, which is then used to check for the expected outcomes.

29.4.6 Testing input properties

The process for testing input properties requires a little extra work. To get started, I added an input property to the `SimpleComponent` class that is used to receive the data model repository, replacing the service that was received by the constructor, as shown in listing 29.20. I have also removed the host event bindings and the output property to keep the example simple.

Listing 29.20. Adding an input property in the `simple.component.ts` file in the `src/app` folder

```
import { Component, EventEmitter, HostListener, Input, Output }
  from "@angular/core";
import { Model } from "../model/repository.model";
import { Product } from "../model/product.model";

@Component({
  selector: "simple",
  templateUrl: "simple.component.html"
})
export class SimpleComponent {

  //constructor(private repository: Model) {}

  category: string = "Soccer";
  highlighted: boolean = false;

  // @Output("pa-highlight")
  // change = new EventEmitter<boolean>();

  getProducts(): Product[] {
    return this.model == undefined ? [] :
      this.model.Products()
        .filter(p => p.category == this.category);
  }

  // @HostListener("mouseenter", ["$event.type"])
  // @HostListener("mouseleave", ["$event.type"])
  // setHighlight(type: string) {
  //   this.highlighted = type == "mouseenter";
  //   this.change.emit(this.highlighted);
  // }

  @Input("pa-model")
  model?: Model;
}
```

The input property is set using an attribute called `pa-model` and is used within the `getProducts` method. Listing 29.21 shows how to write a unit test that targets the input property.

Listing 29.21. Testing an input property in the simple.component.spec.ts file in the src/app/tests folder

```

import { TestBed, ComponentFixture, waitForAsync }
    from "@angular/core/testing";
import { SimpleComponent } from "../simple.component";
import { Product } from "../model/product.model";
import { Model } from "../model/repository.model";
import { signal } from "@angular/core";
import { DebugElement } from "@angular/core";
import { By } from "@angular/platform-browser";
import { Component, ViewChild } from "@angular/core";

@Component({
    template: `<simple [pa-model]="model"></simple>`
})
class TestComponent {

    constructor(public model: Model) { }

    @ViewChild(SimpleComponent)
    SimpleComponent!: SimpleComponent;
}

describe("SimpleComponent", () => {

    let fixture: ComponentFixture<TestComponent>;
    let component: SimpleComponent;
    let debugElement: DebugElement;
    let divElement: HTMLDivElement;

    let mockRepository = {
        Products: signal([
            new Product(1, "test1", "Soccer", 100),
            new Product(2, "test2", "Chess", 100),
            new Product(3, "test3", "Soccer", 100)
        ])
    }

    beforeEach(waitForAsync(() => {
        TestBed.configureTestingModule({
            declarations: [SimpleComponent, TestComponent],
            providers: [
                { provide: Model, useValue: mockRepository }
            ]
        });
        TestBed.compileComponents().then(() => {
            fixture = TestBed.createComponent(TestComponent);
            fixture.detectChanges();
            component = fixture.componentInstance.SimpleComponent;
            debugElement = fixture.debugElement
                .query(By.directive(SimpleComponent));
        });
    }));

    // it("implements output property", () => {
    //     let highlighted: boolean = false;

```

```

//      component.change.subscribe(value => highlighted = value);
//      debugElement.triggerEventHandler("mouseenter",
//          new Event("mouseenter"));
//      expect(highlighted).toBeTruthy();
//      debugElement.triggerEventHandler("mouseleave",
//          new Event("mouseleave"));
//      expect(highlighted).toBeFalsy();
//  });

it("receives the model through an input property", () => {
    component.category = "Chess";
    fixture.detectChanges();
    let products = mockRepository.Products()
        .filter(p => p.category == component.category);
    let componentProducts = component.getProducts();
    for (let i = 0; i < componentProducts.length; i++) {
        expect(componentProducts[i]).toEqual(products[i]);
    }
    expect(debugElement.query(By.css("span"))
        .nativeElement.textContent)
        .toContain(products.length);
});
});

```

The trick here is to define a component that is only required to set up the test and whose template contains an element that matches the selector of the component you want to target. In this example, I defined a component class called `TestComponent` with an inline template defined in the `@Component` decorator that contains a `first` element with a `pa-model` attribute, which corresponds to the `@Input` decorator applied to the `SimpleComponent` class.

The test component class is added to the `declarations` array for the testing module, and an instance is created using the `TestBed.createComponent` method. I used the `@ViewChild` decorator in the `TestComponent` class so that I can get hold of the `SimpleComponent` instance I require for the test. To get the `SimpleComponent` root element, I used the `DebugElement.query` method with the `By.directive` method.

The result is that I can access both the component and its root element for the test, which sets the `category` property and then validates the results both from the component and via the data binding in its template.

29.5 Testing an Angular directive

The process for testing directives is similar to the one required to test input properties, in that a test component and template are used to create an environment for testing in which the directive can be applied. To have a directive to test, I added a file called `attr.directive.ts` to the `src/app` folder and added the code shown in listing 29.22.

NOTE I have shown an attribute directive in this example, but the technique in this section can be used to test structural directives equally well.

Listing 29.22. The contents of the `attr.directive.ts` file in the `src/app` folder

```
import {
```

```

    Directive, ElementRef, Input, SimpleChange
  } from "@angular/core";

  @Directive({
    selector: "[pa-attr]"
  })
  export class PaAttrDirective {

    constructor(private element: ElementRef) { }

    @Input("pa-attr")
    bgClass?: string;

    ngOnChanges(changes: { [property: string]: SimpleChange }) {
      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);
      }
    }
  }
}

```

This is an attribute directive based on an example from part 2 of this book. To create a unit test that targets the directive, I added a file called `attr.directive.spec.ts` to the `src/app/tests` folder and added the code shown in listing 29.23.

Listing 29.23. The contents of the `attr.directive.spec.ts` file in the `src/app/tests` folder

```

import { TestBed, ComponentFixture } from "@angular/core/testing";
import { Component, DebugElement, ViewChild } from "@angular/core";
import { By } from "@angular/platform-browser";
import { PaAttrDirective } from "../attr.directive";

@Component({
  template: `<div><span [pa-attr]="className">Test Content</span></div>`
})
class TestComponent {
  className = "initialClass"

  @ViewChild(PaAttrDirective)
  attrDirective!: PaAttrDirective;
}

describe("PaAttrDirective", () => {

  let fixture: ComponentFixture<TestComponent>;
  let directive: PaAttrDirective;
  let spanElement: HTMLSpanElement;

  beforeEach(() => {
    TestBed.configureTestingModule({
      declarations: [TestComponent, PaAttrDirective],
    });
  });

```

```

        fixture = TestBed.createComponent(TestComponent);
        fixture.detectChanges();
        directive = fixture.componentInstance.attrDirective;
        spanElement =
            fixture.debugElement.query(By.css("span")).nativeElement;
    });

    it("generates the correct number of elements", () => {
        fixture.detectChanges();
        expect(directive.bgClass).toBe("initialClass");
        expect(spanElement.className).toBe("initialClass");

        fixture.componentInstance.className = "nextClass";
        fixture.detectChanges();
        expect(directive.bgClass).toBe("nextClass");
        expect(spanElement.className).toBe("nextClass");
    });
});

```

The text component has an inline template that applies the directive and a property that is referred to in the data binding. The `@ViewChild` decorator provides access to the directive object that Angular creates when it processes the template, and the unit test can check that changing the value used by the data binding has an effect on the directive object and the element to which it has been applied. (You may need to restart the `ng test` command to see the effect of the new unit test).

29.6 Summary

In this chapter, I demonstrated the different ways in which Angular components and directives can be unit tested. I explained the process of installing the test framework and tools and how to create the testbed through which tests are applied. I demonstrated how to test the different aspects of components and how the same techniques can be applied to directives as well.

- Unit testing is performed with Jasmine, which provides an API for describing the expected outcome from unit tests.
- The Angular framework includes classes for testing directives and components in isolation.
- Unit tests can be performed on most Angular features, including input and output properties, dependency injection, and event handling.
- Unit tests can be performed on the HTML content generated by a component or directive, although this can require some awkward interrogation of the DOM API to find the elements to be inspected.

That is all I have to teach you about Angular. I started by creating a simple application and then took you on a comprehensive tour of the different building blocks in the framework, showing you how they can be created, configured, and applied to create web applications.

I wish you every success in your Angular projects, and I can only hope that you have enjoyed reading this book as much as I enjoyed writing it.

index

SYMBOLS

- @angular/cli, 213
 - installing, 11
 - ng new, 12
 - ng serve command, 14
- @angular/forms module, 342
- @angular/forms package, 342

A

- Ajax. *see* Web services
- Angular
 - data flow, 250
 - data bindings, 250
 - template, 250
 - user interaction, 252
- Angular Material, 166
 - installing, 22
- Applications
 - round-trip, 2
 - single-page, 2
- Authentication. *See* SportsStore: authentication

B

- Bootstrap CSS
 - framework, 48
- Browser
 - choosing, 12
- Building for production, 207

C

- Cascading Style Sheets (CSS), 48
- Change detection
 - ChangeDetectorRef class, 433
- Change detection, 254
 - expression
 - evaluation, 256
 - signals, 258
 - strategy, 256
- Change detection
 - NgZone class, 664
- Component libraries
 - additional styles, 783
 - Angular Material, 779
 - choosing, 779
 - Covalent, 779
 - data APIs, 790
 - feature modules, 784
 - installing, 778
 - Material Design, 779
 - mixing styles
 - packages, 783
 - ng-bootstrap, 779
 - ngx-bootstrap, 779
 - Sass, 793
 - sass files, 793
 - scss files, 793
 - themes, 793
 - using components, 781
- Components, 237
 - @Component
 - decorator, 440
 - application structure, 438

- content projection, 453
- creating, 440
- decorator, 237
- dynamic, 455
- input properties, 447
- lifecycle methods
 - ngAfterViewChecked, 461
 - ngAfterViewInit, 461
- output properties, 451
- styles
 - external, 458
 - inline, 457
- template queries, 460
 - @ViewChild
 - decorator, 461
 - @ViewChildren
 - decorator, 461
- templates
 - data bindings, 446
 - external, 445
 - inline, 444
- Cross-origin HTTP requests (CORS), 658
- CSS stylesheets
 - configuring, 227
 - style bundle, 227

D

- Data bindings, 17
 - attribute bindings, 278, 281, 284
 - class bindings, 278
 - classes, 285
 - directive, 277

- event binding, 330
 - brackets, 331
 - event data, 334
 - expression, 331
 - filtering key
 - events, 339
 - host element, 331
 - template
 - references
 - variables, 338
- expressions, 277
- expressions, 279
- host element, 277, 281
- one-way bindings, 275
 - structure, 276
- property bindings, 278, 281
- restrictions, 322
 - limited expression
 - context, 324
- square brackets, 277, 280
- string interpolation, 283
- style bindings, 278
- styles, 285
- target, 277
- two-way, 30
- two-way bindings, 340
 - ngModel directive, 343
- Data model, 100
- Dependency injection.
 - See Services
- Development
 - environment, 4
- Directives
 - host element content
 - change detection, 433
 - ChangeDetectorRef
 - class, 433
- Directives, 277
 - @Directive decorator, 403
 - @Input decorator, 378
 - @Output decorator, 384

- attribute directives, 371
 - data-bound inputs, 377
 - host element
 - attributes, 374
 - required inputs, 382
- built-in directives, 300
- custom directive, 116
- custom events, 384
 - emit method, 386
- host element
 - bindings, 388
- host element content, 426
 - @ContentChild
 - decorator, 427
 - @ContentChildren
 - decorator, 431
- lifecycle hooks, 379
- micro-templates, 302
- ngClass, 278
- ngClass directive, 289
- ng-container element, 321
- ngFor, 278
- ngFor directive, 306
 - even variable, 308
 - expanding micro-template
 - syntax, 312
 - first variable, 308
 - index variable, 308
 - last variable, 308
 - let keyword, 307
 - minimizing
 - changes, 314
 - odd variable, 308
 - of keyword, 307
 - trackBy, 317
 - using variables in
 - child elements, 307
- ngIf, 278
- ngIf directive, 301
 - using literal
 - values, 305
- ngModel directive, 343
- ngStyle, 278
- ngStyle directive, 292

- ngSwitch, 278
- ngSwitch directive, 303
- ngTemplateOutlet, 278
- ngTemplateOutlet
 - directive, 318
 - context data, 319
 - ng-template
 - element, 319
- structural directives, 401
 - collection changes, 416
 - concise syntax, 406
 - context data, 411
 - detecting changes, 403
 - iterating directives, 407
 - ngDoCheck
 - method, 417
 - ng-template
 - element, 405
 - property changes, 414
 - ViewContainerRef
 - class, 403

- Directives
 - using services, 526
- Docker containers, 208
- DOM Events
 - common properties, 335

E

- Editor
 - choosing, 11
- Errata, reporting, 5
- Events, 330
 - event binding, 330

F

- Forms, 345
 - API, 580
 - dynamic forms, 615
 - FormArray class, 615
 - adding controls, 621
 - methods, 615
 - properties, 615

- removing controls, 621
- validating controls, 623
- FormControl class, 582
 - change frequency, 586
 - constructor, 586
 - events, 586
 - state, 587
 - updateOn
 - property, 586
- formControl directive, 581
- formControlName directive, 602
- FormGroup class, 598
 - resetting, 600
 - setting values, 600
- formGroup directive, 602
- observable
 - properties, 584
- reactive forms, 580
- ReactiveFormsModule, 581
- validation, 348, 589, 607
 - asynchronous, 640
 - custom, 628
 - multiple fields, 634
- directives, 630
 - registering
 - services, 631
- validation classes, 349, 364
- whole-form
 - validation, 357

H

HTML

- attributes, 45
 - literal values, 46
 - without values, 45
- document object
 - model, 47
- document structure, 47
- elements, 44
 - content, 46

- hierarchy, 46
- tags, 45
- void elements, 45

J

- JavaScript, 50
 - access control, 85
 - arrays, 76
 - built-in methods, 78
 - enumerating, 77
 - modifying, 77
 - reading, 77
 - spread operator, 78
- boolean type, 53
- classes, 82
 - inheritance, 86
- closures, 75
- coalescing values, 66
- conditional
 - statements, 62
- constructor, 85
- functions
 - as arguments to
 - other functions, 74
 - default
 - parameters, 72
 - defining, 71
 - optional
 - parameters, 72
 - rest parameters, 73
 - results, 73
- literal values in
 - directive
 - expressions, 305
- modules, 88
 - export keyword, 88
 - import keyword, 88
 - NPM packages, 89
 - resolution, 89
- null, 53
- null coalescing
 - operator, 66
- nullish coalescing
 - operator, 66
- number type, 53
- objects

- literal syntax, 80
- optional
 - properties, 82
- operators, 61
 - equality versus
 - identity
 - operator, 62
- optional chaining
 - operator, 67
- primitive types, 53
- statements
 - conditional, 62
- string type, 53
- template strings, 60
- truthy and falsy
 - values, 63, 288
- types, 59
 - booleans, 59
 - converting
 - explicitly, 64
 - null, 61
 - numbers, 60
 - strings, 59
 - template strings, 60
 - undefined, 61
 - undefined, 53
 - variable closure, 75
 - variables and
 - constants, 57
- JSON Web Token, 157

L

- Linting
 - ESLint, 230
- Listings
 - complete, 5
 - interleaved, 6
 - partial, 6

M

- Material Design, 779
- Micro-templates
 - use by directives, 302
- Modules
 - @NgModule
 - decorator, 543
 - bootstrap property, 544
 - declarations property, 544

- dynamic. *see* URL routing
- dynamic loading
 - SportsStore, 152
- feature modules, creating, 547
- imports property, 543
- providers property, 544
- root module, 541
- using with JavaScript modules, 552

N

- ng add Command, 222
- ng command, 12
- ng config Command, 227
- ng lint command, 230
- ng new command, 213
- ng serve command, 223
- ng-container element, 321
- NgZone class, 664
- Node Package Manager, 11
- Node.js
 - installing, 10
 - NPM, 11
 - package manager, 11

P

- Pipes
 - @Pipe decorator, 470
 - applying, 467
 - arguments, 468
 - async pipe, 505
 - combining, 473
 - creating, 469
 - formatting currency amounts, 483
 - formatting dates, 488
 - formatting numbers, 479
 - formatting
 - percentages, 486
 - formatting string
 - case, 494
 - impure pipes, 474
 - JSON serialization, 497
 - key/value pairs, 500

- pluralizing values, 503
- pure pipes, 474
- selecting values, 501, 503
- slicing arrays, 498
- using services, 524
- Polyfills, 227
- Prerendering, 773
 - routes file, 773
- Progressive Web Applications, 199
- Projects
 - .editorconfig file, 216
 - .gitignore file, 216
 - ahead-of-time compilation, 239
 - angular.json file, 216
 - AoT compilation, 239
 - build process, 225
 - bundles, 225
 - components, 237
 - contents, 215
 - data model, 240
 - development tools, 223
 - hot reloading, 226
 - HTML document, 234
 - node_modules folder, 216, 218
 - package.json file, 216
 - packages, 218, 222
 - global packages, 220
 - scripts, 221
 - versions, 219
 - root module, 236
 - src folder, 216, 217
 - src/app folder, 217
 - src/assets folder, 217
 - src/index.html file, 217
 - src/main.ts file, 217
 - src/polyfills.ts file, 217
 - src/styles.css file, 217
 - src/tests.ts file, 217
 - structure, 215
 - tsconfig.json file, 216
 - tslint.json file, 216
 - webpack, 225

R

- React, 3
- Reactive extensions, 267
 - async pipe, 505
 - Observable
 - subscribe method, 267
- Reactive forms, 580
- Rehydration, 759
- REST. *see* Web services
- RESTful web services.
 - see* Web services
 - SportsStore example, 146
- Root module, 236
- Round-trip applications, 2
- RxJS, 267

S

- Sass, 793
- Schematics API, 222
- Server-side rendering, 756
 - browser APIs, 761
 - commands, 762
 - guarding, 767
 - installing, 756
 - navigation, 764
 - preparing the application, 758
 - prerendering, 773
 - rehydration, 759
- Services
 - @Injectable
 - decorator, 517
 - component isolation, 531
 - dependency injection, 517
 - providers property, 520
 - receiving services, 518
 - registering services, 520
 - registering validators, 631
 - services in directives, 526
 - services in pipes, 524

- shared object
 - problem, 511
- Signals, 100, 258
 - computed, 103
 - computed signals, 260
 - dependencies, 260
 - effects, 262
 - functions, 258
 - computed function, 258
 - effect function, 258
 - signal function, 258
 - working with
 - observables, 269
 - toObservable
 - function, 574
 - writable signals, 258
 - methods, 259
- Single-page applications, 2
- SportsStore
 - additional packages, 92
 - Angular Material, 166
 - authentication, 157
 - JSON Web Token, 157
 - bootstrap file, 99
 - cart, 120
 - summary
 - component, 123
 - category selection, 109
 - component library, 166
 - containerizing, 208
 - creating the
 - container, 210
 - creating the
 - image, 210
 - deployment
 - packages, 208
 - Dockerfile, 209
 - stopping the
 - container, 211
 - creating the project, 92
 - data model, 100
 - data source, 100
 - signals, 100

- data source, 100
- displaying products, 108
- dynamic module, 152
- navigation, 132
- orders, 139
- pagination, 112
- persistent data, 204
- prerendering, 192
 - detecting, 193
 - packages, 192
- production build, 207
- progressive features, 199
 - caching, 199
 - connectivity, 200
- project structure, 96
- REST data, 146
- root component, 97
- root module, 98
- route guard, 134
- server-side rendering, 192
- URL routing, 128
- web service, 93
- String interpolation, 283

T

- Templates
 - variables, 35
- TypeScript, 50
 - any type, 52
 - concise constructor, 16
 - specific types, 53
 - type annotation, 52
 - type union, 54
 - variables and
 - constants, 57

U

- Unit testing
 - components
 - configuring
 - dependencies, 813
 - data bindings, 815
 - events, 819
 - input properties, 824
 - output properties, 821

- templates, 817
- directives, 826
- Jasmine, 808
 - methods, 809
- Karma test runner, 808
- ng test command, 808
- TestBed class, 811
- URL routing, 128, 669
 - ActivatedRoute class, 678
 - basic configuration, 669
 - child routes, 715
 - parameters, 718
 - route outlets, 716
 - dynamic modules, 749
 - guarding, 753
 - specifying, 751
 - using, 752
 - guarding, 134
 - guards, 723
 - displaying a
 - loading
 - message, 729
 - mapToCanActivate
 - function, 735
 - preventing
 - navigation, 732
 - preventing route
 - activation, 733
 - resolvers, 724
 - mapToResolve
 - function, 728
 - navigating within a
 - component, 709
 - navigation events, 691
 - navigation links, 673
 - optional URL
 - segments, 687
 - programmatic
 - navigation, 678, 689
 - redirections, 707
 - route parameters, 682
 - routerLink directive, 673
 - router-outlet element, 671

- Routes class, 670
- styles for active elements, 710
- wildcard routes, 705

V

- Vue.js, 3

W

- Web services, 648
 - cross-origin requests, 658
 - errors, 661
 - HTTP verbs, 649
 - HttpClient class, 650

- consolidating requests, 656
 - methods, 650
 - responses, 651
- NgZone class, 664
- request headers, 659