

## CHAPTER 7



# SportsStore: A Real Application

In Chapter 2, I built a quick and simple Angular application. Small and focused examples allow me to demonstrate specific Angular features, but they can lack context. To help overcome this problem, I am going to create a simple but realistic e-commerce application.

My application, called SportsStore, will follow the classic approach taken by online stores everywhere. I will create an online product catalog that customers can browse by category and page, a shopping cart where users can add and remove products, and a checkout where customers can enter their shipping details and place their orders. I will also create an administration area that includes create, read, update, and delete (CRUD) facilities for managing the catalog—and I will protect it so that only logged-in administrators can make changes. Finally, I show you how to prepare and deploy an Angular application.

My goal in this chapter and those that follow is to give you a sense of what real Angular development is like by creating as realistic an example as possible. I want to focus on Angular, of course, and so I have simplified the integration with external systems, such as the data store, and omitted others entirely, such as payment processing.

The SportsStore example is one that I use in a few of my books, not least because it demonstrates the ways in which 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 MVC 2* 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 detail chapters to get into the depth. Either way, don't expect to understand everything right away—Angular has a lot of moving parts, and the SportsStore application is intended to show you how they fit together without diving too deeply into the details that I spend the rest of the book describing.

## Preparing the Project

To create the SportsStore project, open a command prompt, navigate to a convenient location, and run the following command:

---

```
ng new SportsStore
```

---

The `angular-cli` package will create a new project for Angular development, with configuration files, placeholder content, and development tools. The project setup process can take some time since there are many NPM packages to download and install.

---

■ **Tip** You can download the example project for this chapter—and for all the other chapters in this book—from <https://github.com/Apress/pro-angular-6>.

---

## 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@4.1.1
npm install font-awesome@4.7.0
npm install --save-dev json-server@0.12.1
npm install --save-dev jsonwebtoken@8.1.1
```

---

It is important to use the version numbers shown in the listing. You may see warnings about unmet peer dependencies as you add the packages, but you can ignore them. Some of the packages are installed using the `--save-dev` argument, which indicates they are used during development and will not be part of the SportsStore application.

## Adding the CSS Style Sheets to the Application

Once the packages have been installed, add the statements shown in Listing 7-1 to the `angular.json` file to incorporate the CSS files from the Bootstrap CSS framework and the Font Awesome packages into the application. I will use the Bootstrap CSS styles for all the HTML content in the SportsStore application, and I use icons from the Font Awesome package to present a summary of a shopping cart to the user.

**Listing 7-1.** Adding CSS to the `angular.json` File in the SportsStore Folder

```
...
"architect": {
  "build": {
    "builder": "@angular-devkit/build-angular:browser",
    "options": {
      "outputPath": "dist/SportsStore",
      "index": "src/index.html",
      "main": "src/main.ts",
      "polyfills": "src/polyfills.ts",
      "tsConfig": "src/tsconfig.app.json",
      "assets": [
        "src/favicon.ico",
        "src/assets"
      ],
      "styles": [
        "src/styles.css",
        "node_modules/bootstrap/dist/css/bootstrap.min.css",
        "node_modules/font-awesome/css/font-awesome.min.css"
      ]
    }
  }
}
```

```

    "scripts": []
  },
  ...

```

## Preparing the RESTful Web Service

The SportsStore application will use asynchronous HTTP requests to get model data provided by a RESTful web service. As I describe in Chapter 24, 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 that the operation applies to.

I added the `json-server` package to the project in the previous section. This is an excellent package for creating web services from JSON data or JavaScript code. Add the statement shown in Listing 7-2 to the `scripts` section of the `package.json` file so that the `json-server` package can be started from the command line.

**Listing 7-2.** Adding a Script in the `package.json` File in the SportsStore Folder

```

...
"scripts": {
  "ng": "ng",
  "start": "ng serve",
  "build": "ng build",
  "test": "ng test",
  "lint": "ng lint",
  "e2e": "ng e2e",
  "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 7-3, 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 file names 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 7-3.** 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 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 7-4.

**Listing 7-4.** 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";

module.exports = function (req, res, next) {

  if ((req.url == "/api/login" || req.url == "/login")
    && req.method == "POST") {
    if (req.body != null && 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 (((req.url.startsWith("/api/products")
    || req.url.startsWith("/products"))
    || (req.url.startsWith("/api/categories")
    || req.url.startsWith("/categories"))) && req.method != "GET")
    || ((req.url.startsWith("/api/orders")

```

```

    || req.url.startsWith("/orders")) && req.method != "POST")) {
    let token = req.headers["authorization"];
    if (token != null && 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 9](#), including how to authenticate users with Angular.

---

■ **Caution** Don't use the code in [Listing 7-4](#) 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 the development client side with Angular, but this is not suitable for real projects.

---

## Preparing the HTML File

Every Angular web application relies on an HTML file that is loaded by the browser and that loads and starts the application. Edit the `index.html` file in the `SportsStore/src` folder to remove the placeholder content and to add the elements shown in [Listing 7-5](#).

**Listing 7-5.** 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="m-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 8.

## Creating the Folder Structure

An important part of setting up any 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 7-1.

**Table 7-1.** The Additional Folders Required for the SportsStore Project

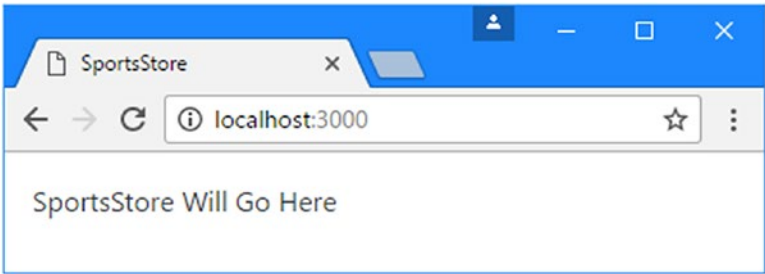
Folder	Description
SportsStore/src/app/model	This folder will contain the code for the data model.
SportsStore/src/app/store	This folder will contain the functionality for basic shopping.
SportsStore/src/app/admin	This folder will contain the functionality for administration.

## Running the Example Application

Make sure that all the changes have been saved, and run the following command in the SportsStore folder:

```
ng serve --port 3000 --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 7-1.



**Figure 7-1.** Running the example application

The development web server will start on port 3000, so the URL for the application will be `http://localhost:3000`. You don't have to include the name of the HTML document because `index.html` is the default file that the server responds with.

## 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 7-3, as follows:

---

```
{
  "id": 1,
  "name": "Kayak",
  "category": "Watersports",
  "description": "A boat for one person",
  "price": 275
}
```

---

## Preparing the Angular Project Features

Every Angular project requires some basic preparation. In the sections that follow, I replace the placeholder content to build the foundation for the SportsStore application.

### Updating the Root Component

The root component is the Angular building block that will manage the contents of the app element in the HTML document from Listing 7-5. 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 7-6.

**Listing 7-6.** 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. The complete set of component properties are described in Chapter 17, 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 8, I use a feature called *URL routing* to adapt the content automatically based on user actions.

## Updating the Root Module

There are two types of Angular module: 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 component file is `app.module.ts`, which is created in the `SportsStore/src/app` folder. No changes are required to this file for the moment, and its initial content is shown in Listing 7-7.

**Listing 7-7.** The Contents of the `app.module.ts` File in the `src/app` Folder

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
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 `AppModule` class. I'll add information to this decorator's properties as I add features to the SportsStore application, but this basic configuration is enough to start the application.

## Inspecting the Bootstrap File

The next piece of plumbing is the bootstrap file, which starts the application. This book is focused on using Angular to create applications that work in web browsers, but the Angular platform can be ported to different environments. The bootstrap file uses the Angular browser platform to load the root module and start the application. No changes are required for the contents of the `main.ts` file, which is in the `SportsStore/src` folder, as shown in Listing 7-8.



**Listing 7-8.** The Contents of the main.ts File in the src Folder

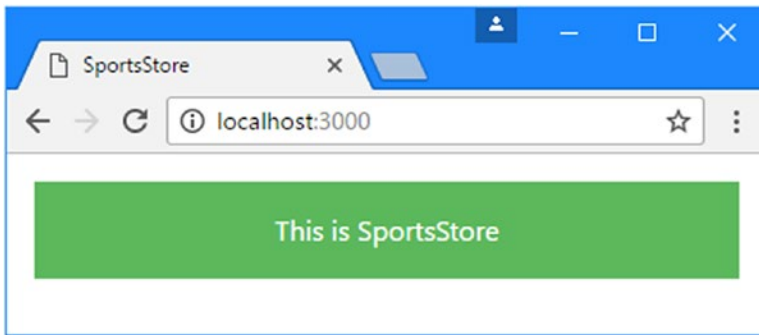
```
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

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

if (environment.production) {
  enableProdMode();
}

platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.log(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 7-2.

**Figure 7-2.** Starting the SportsStore application

## Starting the Data Model

The best place to start any new project is the data model. I want to get to the point where you can see some Angular features at work, so rather than define the data model from end to end, I am going to put some basic functionality in place using dummy data. I'll use this data to create user-facing features and then return to the data model to wire it up to the RESTful web service in Chapter 8.

## 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 7-9.

**Listing 7-9.** 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 in Listing 7-3. The question marks (the ? characters) that follow the parameter names indicate that these are optional parameters that can be omitted when creating new objects using the Product class, which can be useful when writing applications where model object properties will be populated using HTML forms.

## Creating the Dummy Data Source

To prepare for the transition from dummy to real data, I am going to feed the application data using a data source. The rest of the application won't know where the data is coming from, which will make the switch to getting data using HTTP requests seamless.

I added a file called static.datasource.ts to the SportsStore/src/app/model folder and defined the class shown in Listing 7-10.

**Listing 7-10.** The Contents of the static.datasource.ts File in the src/app/model Folder

```
import { Injectable } from "@angular/core";
import { Product } from "../product.model";
import { Observable, from } from "rxjs";

@Injectable()
export class StaticDataSource {
  private products: Product[] = [
    new Product(1, "Product 1", "Category 1", "Product 1 (Category 1)", 100),
    new Product(2, "Product 2", "Category 1", "Product 2 (Category 1)", 100),
    new Product(3, "Product 3", "Category 1", "Product 3 (Category 1)", 100),
    new Product(4, "Product 4", "Category 1", "Product 4 (Category 1)", 100),
    new Product(5, "Product 5", "Category 1", "Product 5 (Category 1)", 100),
    new Product(6, "Product 6", "Category 2", "Product 6 (Category 2)", 100),
    new Product(7, "Product 7", "Category 2", "Product 7 (Category 2)", 100),
    new Product(8, "Product 8", "Category 2", "Product 8 (Category 2)", 100),
    new Product(9, "Product 9", "Category 2", "Product 9 (Category 2)", 100),
    new Product(10, "Product 10", "Category 2", "Product 10 (Category 2)", 100),
    new Product(11, "Product 11", "Category 3", "Product 11 (Category 3)", 100),
    new Product(12, "Product 12", "Category 3", "Product 12 (Category 3)", 100),
    new Product(13, "Product 13", "Category 3", "Product 13 (Category 3)", 100),
    new Product(14, "Product 14", "Category 3", "Product 14 (Category 3)", 100),
    new Product(15, "Product 15", "Category 3", "Product 15 (Category 3)", 100),
  ];
}
```

```

    getProducts(): Observable<Product[]> {
        return from([this.products]);
    }
}

```

The `StaticDataSource` class defines a method called `getProducts`, which returns the dummy data. The result of calling the `getProducts` method is an `Observable<Product[]>`, which is an `Observable` that produces arrays of `Product` objects.

The `Observable` class is provided by the `Reactive Extensions` package, which is used by Angular to handle state changes in applications. I describe the `Observable` class in Chapter 23, but for this chapter, it is enough to know that an `Observable` object represents an asynchronous task that will produce a result at some point in the future. Angular exposes its use of `Observable` objects for some features, including making HTTP requests, and this is why the `getProducts` method returns an `Observable<Product[]>` rather than simply returning the data synchronously.

The `@Injectable` decorator has been applied to the `StaticDataSource` class. This decorator is used to tell Angular that this class will be used as a service, which allows other classes to access its functionality through a feature called *dependency injection*, which is described in Chapters 19 and 20. You'll see how services work as the application takes shape.

---

■ **Tip** Notice that I have to import `Injectable` from the `@angular/core` JavaScript module so that I can apply the `@Injectable` decorator. I won't highlight the all the different Angular classes that I import for the `SportsStore` example, but you can get full details in the chapters that describe the features they relate to.

---

## Creating the Model Repository

The data source is responsible for providing the application with the data it requires, but access to that data is typically mediated by a *repository*, which is responsible for distributing that data to individual application building blocks so that the details of how the data has been obtained are kept hidden. I added a file called `product.repository.ts` in the `SportsStore/src/app/model` folder and defined the class shown in Listing 7-11.

**Listing 7-11.** The Contents of the `product.repository.ts` File in the `src/app/model` Folder

```

import { Injectable } from "@angular/core";
import { Product } from "../product.model";
import { StaticDataSource } from "../static.datasource";

@Injectable()
export class ProductRepository {
    private products: Product[] = [];
    private categories: string[] = [];

    constructor(private dataSource: StaticDataSource) {
        dataSource.getProducts().subscribe(data => {
            this.products = data;
            this.categories = data.map(p => p.category)
                .filter((c, index, array) => array.indexOf(c) == index).sort();
        });
    }
}

```

```

    getProducts(category: string = null): Product[] {
        return this.products
            .filter(p => category == null || category == p.category);
    }

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

    getCategories(): string[] {
        return this.categories;
    }
}

```

When Angular needs to create a new instance of the repository, it will inspect the class and see that it needs a `StaticDataSource` object to invoke the `ProductRepository` constructor and create a new object.

The repository constructor calls the data source's `getProducts` method and then uses the `subscribe` method on the `Observable` object that is returned to receive the product data. See Chapter 23 for details of how `Observable` objects work.

## 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 7-12.

---

■ **Tip** Don't worry if all the file names 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 7-12.** The Contents of the `model.module.ts` File in the `src/app/model` Folder

```

import { NgModule } from "@angular/core";
import { ProductRepository } from "../product.repository";
import { StaticDataSource } from "../static.datasource";

@NgModule({
    providers: [ProductRepository, StaticDataSource]
})
export class ModelModule { }

```

The `@NgModule` decorator is used to create feature modules, and its properties tell Angular how the module should be used. There is only one property in this module, `providers`, and it tells Angular which classes should be used as services for the dependency injection feature, which is described in Chapters 19 and 20. Features modules—and the `@NgModule` decorator—are described in Chapter 21.

## 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 7-3.



**Figure 7-3.** The basic structure of the store

In the sections that follow, I'll use Angular features and the data in the model to create the layout shown in the figure.

## Creating the Store Component and Template

As you become familiar with Angular, you will learn that features can be combined to solve the same problem in different ways. I try to introduce some variety into the SportsStore project to showcase some important Angular features, but I am going to keep things simple for the moment in the interest of being able to get the project started quickly.

With this in mind, the starting point for the store functionality will be a new component, which is a class that provides data and logic to an HTML template, which contains data bindings that generate content dynamically. I created a file called `store.component.ts` in the `SportsStore/src/app/store` folder and defined the class shown in Listing 7-13.

**Listing 7-13.** The Contents of the `store.component.ts` File in the `src/app/store` Folder

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

@Component({
  selector: "store",
  templateUrl: "store.component.html"
})
export class StoreComponent {
```

```

    constructor(private repository: ProductRepository) { }

    get products(): Product[] {
        return this.repository.getProducts();
    }

    get categories(): string[] {
        return this.repository.getCategories();
    }
}

```

The `@Component` decorator has been applied to the `StoreComponent` class, which tells Angular that it is a component. The decorator's properties tell Angular how to apply the component to HTML content (using an element called `store`) and how to find the component's template (in a file called `store.component.html`).

The `StoreComponent` class provides the logic that will support the template content. The constructor receives a `ProductRepository` object as an argument, provided through the dependency injection feature described in Chapters 20 and 21. 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 7-14.

**Listing 7-14.** The Contents of the `store.component.html` File in the `src/app/store` Folder

```

<div class="container-fluid">
  <div class="row">
    <div class="col bg-dark text-white">
      <a class="navbar-brand">SPORTS STORE</a>
    </div>
  </div>
  <div class="row text-white">
    <div class="col-3 bg-info p-2">
      {{categories.length}} Categories
    </div>
    <div class="col-9 bg-success p-2">
      {{products.length}} Products
    </div>
  </div>
</div>

```

The template is simple, just to get started. Most of the elements provide the structure for the store layout and apply some Bootstrap CSS classes. There are only two Angular data bindings at the moment, which are denoted by the `{{` and `}}` characters. These are *string interpolation* bindings, and they tell Angular to evaluate the binding expression and insert the result into the element. The expressions in these bindings display the number of products and categories provided by the store component.

## Creating the Store Feature Module

There isn't much store functionality in place at the moment, 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 7-15.

**Listing 7-15.** 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 working with HTML form elements. The decorator uses the `declarations` property to tell Angular about the `StoreComponent` class, which the `exports` property tells Angular can be also used in other parts of the application, which is important because it will be used by the root module.

## Updating the Root Component and Root Module

Applying the basic model and store functionality requires updating the application's root module to import the two feature modules and also requires updating the root module's template to add the HTML element to which the component in the store module will be applied. Listing 7-16 shows the change to the root component's template.

**Listing 7-16.** 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 7-13. Listing 7-17 shows the change required to the root module so that Angular loads the feature module that contains the store functionality.

**Listing 7-17.** 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({
```

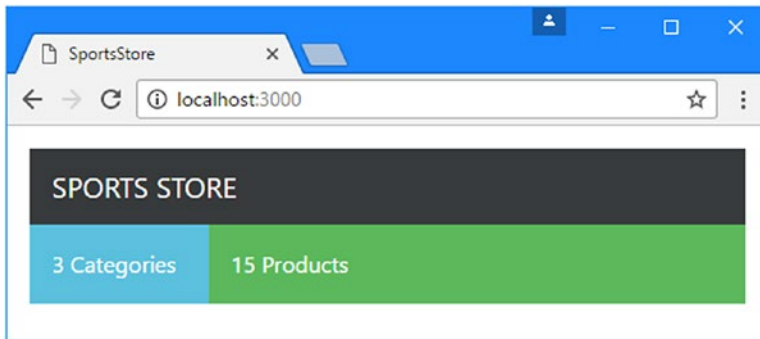
```

imports: [BrowserModule, StoreModule],
  declarations: [AppComponent],
  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 7-4.

All the building blocks created in the previous section work together to display the—admittedly simple—content, which shows how many products there are and how many categories they fit in to.



**Figure 7-4.** Basic features in the SportsStore application

## Adding Store Features the Product Details

The nature of Angular development begins with a slow start as the foundation of the project is put in place and the basic building blocks are created. But once that's done, new features can be created relatively easily. In the sections that follow, I add features to the store so that the user can see the products on offer.

### Displaying the Product Details

The obvious place to start is to display details for the products so that the customer can see what's on offer. Listing 7-18 adds HTML elements to the store component's template with data bindings that generate content for each product provided by the component.

**Listing 7-18.** Adding Elements in the store.component.html File in the src/app/store Folder

```

<div class="container-fluid">
  <div class="row">
    <div class="col bg-dark text-white">
      <a class="navbar-brand">SPORTS STORE</a>
    </div>
  </div>
  <div class="row">
    <div class="col-3 bg-info p-2 text-white">
      {{categories.length}} Categories
    </div>

```



```

<div class="col-9 p-2">
  <div *ngFor="let product of products" class="card m-1 p-1 bg-light">
    <h4>
      {{product.name}}
      <span class="badge badge-pill badge-primary float-right">
        {{ product.price | currency:"USD":"symbol":"2.2-2" }}
      </span>
    </h4>
    <div class="card-text bg-white p-1">{{product.description}}</div>
  </div>
</div>
</div>

```

Most of the elements control the layout and appearance of the content. The most important change is the addition of an Angular data binding expression.

```

...
<div *ngFor="let product of products" class="card m-1 p-1 bg-light">
...

```

This is an example of a directive, which transforms the HTML element it is applied to. This specific directive is called `ngFor`, and it transforms the `div` element by duplicating it for each object returned by the component's `products` property. Angular includes a range of built-in directives that perform the most commonly required tasks, as described in Chapter 13.

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>
...

```

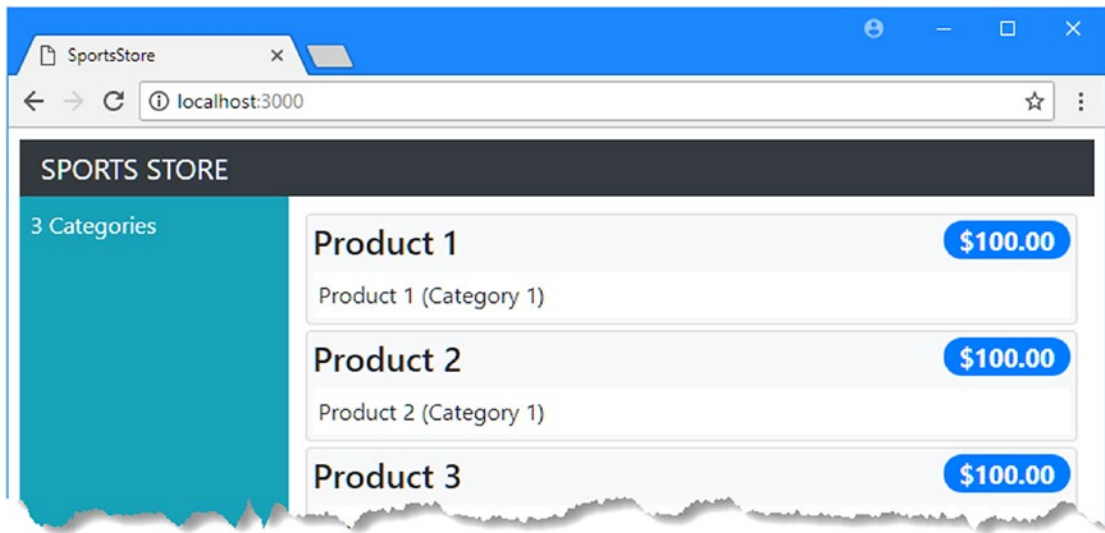
Not all data in an application's data model can be displayed directly to the user. Angular includes a feature called *pipes*, which are classes used to transform or prepare a data value for its use in a data binding. There are several built-in pipes included with Angular, including the currency pipe, 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 7-5.



**Figure 7-5.** *Displaying product information*

## Adding Category Selection

Adding support for filtering the list of products by category requires preparing the store component so that it keeps track of which category the user wants to display and requires changing the way that data is retrieved to use that category, as shown in Listing 7-19.

**Listing 7-19.** Adding Category Filtering in the `store.component.ts` File in the `src/app/store` Folder

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

@Component({
  selector: "store",
  templateUrl: "store.component.html"
})
export class StoreComponent {
  public selectedCategory = null;

  constructor(private repository: ProductRepository) {}

  get products(): Product[] {
    return this.repository.getProducts(this.selectedCategory);
  }

  get categories(): string[] {
    return this.repository.getCategories();
  }

  changeCategory(newCategory?: string) {
```

```

    this.selectedCategory = newCategory;
  }
}

```

The changes are simple because they build on the foundation that took so long to create at the start of the chapter. The `selectedCategory` property is assigned the user's choice of category (where `null` means all categories) and is used in the `updateData` method as an argument to the `getProducts` method, delegating the filtering to the data source. The `changeCategory` method brings these two members together in a method that can be invoked when the user makes a category selection.

Listing 7-20 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 7-20.** Adding Category Buttons in the `store.component.html` File in the `src/app/store` Folder

```

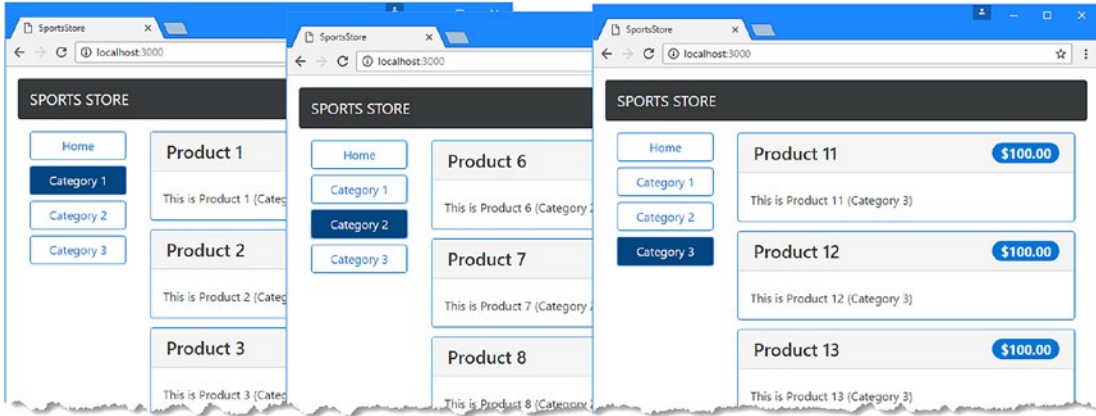
<div class="container-fluid">
  <div class="row">
    <div class="col bg-dark text-white">
      <a class="navbar-brand">SPORTS STORE</a>
    </div>
  </div>
  <div class="row">
    <div class="col-3 p-2">
      <button class="btn btn-block btn-outline-primary" (click)="changeCategory()">
        Home
      </button>
      <button *ngFor="let cat of categories"
        class="btn btn-outline-primary btn-block"
        [class.active]="cat == selectedCategory" (click)="changeCategory(cat)">
        {{cat}}
      </button>
    </div>
    <div class="col-9 p-2">
      <div *ngFor="let product of products" class="card m-1 p-1 bg-light">
        <h4>
          {{product.name}}
          <span class="badge badge-pill badge-primary float-right">
            {{ product.price | currency:"USD":"symbol":"2.2-2" }}
          </span>
        </h4>
        <div class="card-text bg-white p-1">{{product.description}}</div>
      </div>
    </div>
  </div>
</div>

```

There are two new button elements in the template. The first is a Home button, and it has an event binding that invokes the component's `changeCategory` method when the button is clicked. No argument is provided to the method, which has the effect of setting the category to `null` and selecting all the products.

The `ngFor` binding has been applied to the other button element, with an expression that will repeat the element for each value in the array returned by the component's `categories` property. The button has a `click` event binding whose expression calls the `changeCategory` method to select the current category,

which will filter the products displayed to the user. There is also a class binding, which adds the button element to the active class when the category associated with the button is the selected category. This provides the user with visual feedback when the categories are filtered, as shown in Figure 7-6.



**Figure 7-6.** *Selecting product categories*

## Adding Product Pagination

Filtering the products by category has helped make the product list more manageable, but a more typical approach is to break the list into smaller sections and present each of them as a page, along with navigation buttons that move between the pages. Listing 7-21 enhances the store component so that it keeps track of the current page and the number of items on a page.

**Listing 7-21.** Adding Pagination Support in the store.component.ts File in the src/app/store Folder

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

@Component({
  selector: "store",
  templateUrl: "store.component.html"
})
export class StoreComponent {
  public selectedCategory = null;
  public productsPerPage = 4;
  public selectedPage = 1;

  constructor(private repository: ProductRepository) {}

  get products(): Product[] {
    let pageIndex = (this.selectedPage - 1) * this.productsPerPage;
    return this.repository.getProducts(this.selectedCategory)
      .slice(pageIndex, pageIndex + this.productsPerPage);
  }
}
```

```

get categories(): string[] {
    return this.repository.getCategories();
}

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

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

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

get pageNumbers(): number[] {
    return Array(Math.ceil(this.repository
        .getProducts(this.selectedCategory).length / this.productsPerPage))
        .fill(0).map((x, i) => i + 1);
}
}

```

There are two new features in this listing. The first is the ability to get a page of products, and the second is to change the size of the pages, allowing the number of products that each page contains to be altered.

There is an oddity that the component has to work around. There is a limitation in the built-in `ngFor` directive that Angular provides, which can generate content only for the objects in an array or a collection, rather than using a counter. Since I need to generate numbered page navigation buttons, this means I need to create an array that contains the numbers I need, like this:

```

...
return Array(Math.ceil(this.repository.getProducts(this.selectedCategory).length
    / this.productsPerPage)).fill(0).map((x, i) => i + 1);
...

```

This statement creates a new array, fills it with the value 0, and then uses the `map` method to generate a new array with the number sequence. This works well enough to implement the pagination feature, but it feels awkward, and I demonstrate a better approach in the next section. Listing 7-22 shows the changes to the store component's template to implement the pagination feature.

**Listing 7-22.** Adding Pagination in the `store.component.html` File in the `src/app/store` Folder

```

<div class="container-fluid">
  <div class="row">
    <div class="col bg-dark text-white">
      <a class="navbar-brand">SPORTS STORE</a>
    </div>
  </div>
  <div class="row">
    <div class="col-3 p-2">
      <button class="btn btn-block btn-outline-primary" (click)="changeCategory()">

```

```

        Home
    </button>
    <button *ngFor="let cat of categories"
        class="btn btn-outline-primary btn-block"
        [class.active]="cat == selectedCategory" (click)="changeCategory(cat)">
        {{cat}}
    </button>
</div>
<div class="col-9 p-2">
    <div *ngFor="let product of products" class="card m-1 p-1 bg-light">
        <h4>
            {{product.name}}
            <span class="badge badge-pill badge-primary float-right">
                {{ product.price | currency:"USD":"symbol":"2.2-2" }}
            </span>
        </h4>
        <div class="card-text bg-white p-1">{{product.description}}</div>
    </div>

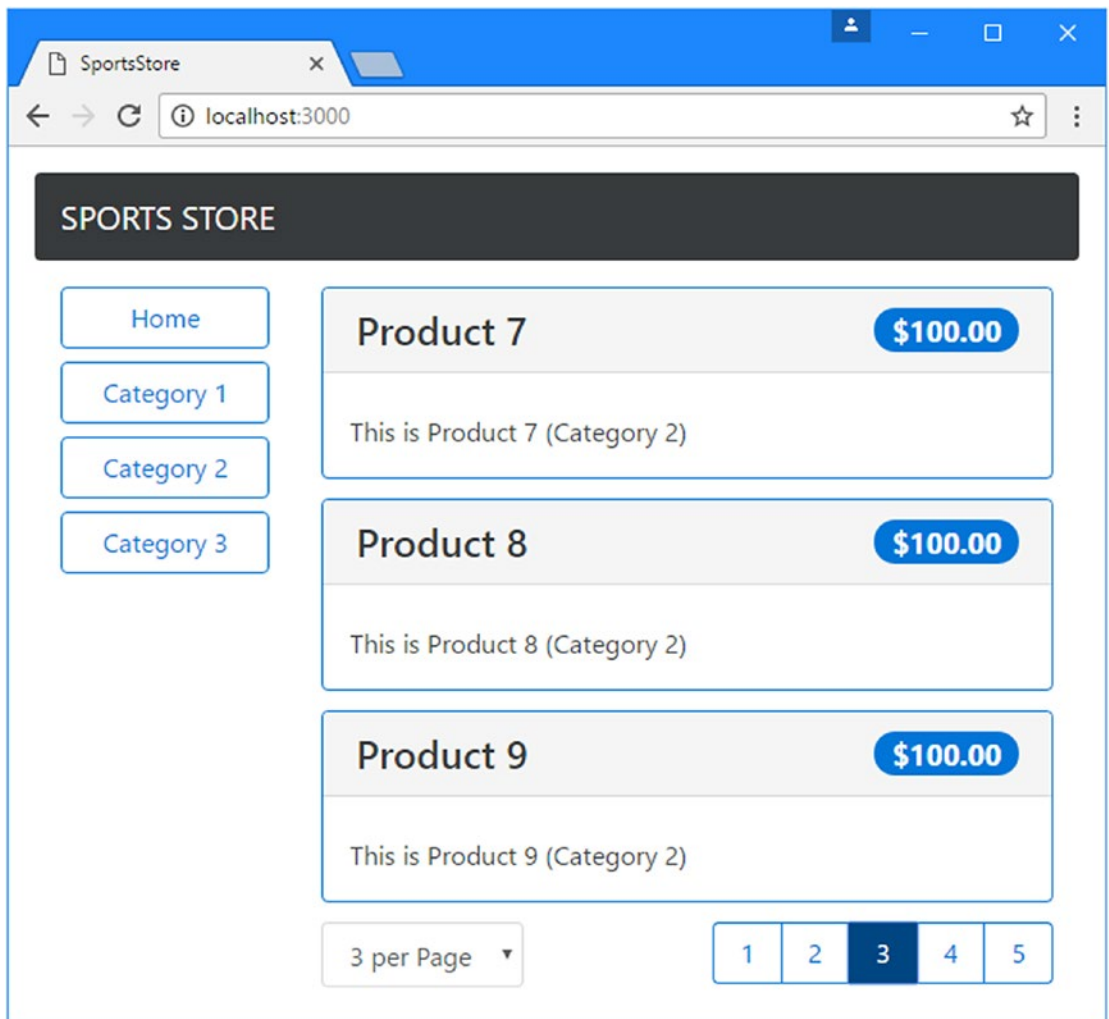
    <div class="form-inline float-left mr-1">
        <select class="form-control" [value]="productsPerPage"
            (change)="changePageSize($event.target.value)">
            <option value="3">3 per Page</option>
            <option value="4">4 per Page</option>
            <option value="6">6 per Page</option>
            <option value="8">8 per Page</option>
        </select>
    </div>

    <div class="btn-group float-right">
        <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 7-7.

■ **Tip** The select element in Listing 7-22 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 7-21. 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 7-7.** *Pagination for products*

## Creating a Custom Directive

In this section, I am going to create a custom directive so that I don't have to generate an array full of numbers to create the page navigation buttons. Angular provides a good range of built-in directives, but it is a simple process to create your own directives to solve problems that are specific to your application or to support features that the built-in directives don't have. I added a file called `counter.directive.ts` in the `src/app/store` folder and used it to define the class shown in Listing 7-23.

**Listing 7-23.** The Contents of the `counter.directive.ts` File in the `src/app/store` Folder

```
import {
  Directive, ViewContainerRef, TemplateRef, Input, Attribute, SimpleChanges
} from "@angular/core";

@Directive({
  selector: "[counterOf]"
})
export class CounterDirective {

  constructor(private container: ViewContainerRef,
    private template: TemplateRef<Object>) {

  }

  @Input("counterOf")
  counter: number;

  ngOnChanges(changes: SimpleChanges) {
    this.container.clear();
    for (let i = 0; i < this.counter; i++) {
      this.container.createEmbeddedView(this.template,
        new CounterDirectiveContext(i + 1));
    }
  }
}

class CounterDirectiveContext {
  constructor(public $implicit: any) { }
}
```

This is an example of a structural directive, which is described in detail in Chapter 16. This directive is applied to elements through a `counter` property and relies on special features that Angular provides for creating content repeatedly, just like the built-in `ngFor` directive. In this case, rather than yield each object in a collection, the custom directive yields a series of numbers that can be used to create the page navigation buttons.

---

■ **Tip** This directive deletes all the content it has created and starts again when the number of pages changes. This can be an expensive process in more complex directives, and I explain how to improve performance in Chapter 16.

---



To use the directive, it must be added to the `declarations` property of its feature module, as shown in Listing 7-24.

**Listing 7-24.** 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 7-25.

**Listing 7-25.** 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="col bg-dark text-white">
      <a class="navbar-brand">SPORTS STORE</a>
    </div>
  </div>
  <div class="row">

    <div class="col-3 p-2">
      <button class="btn btn-block btn-outline-primary" (click)="changeCategory()">
        Home
      </button>
      <button *ngFor="let cat of categories"
        class="btn btn-outline-primary btn-block"
        [class.active]="cat == selectedCategory" (click)="changeCategory(cat)">
        {{cat}}
      </button>
    </div>

    <div class="col-9 p-2">
      <div *ngFor="let product of products" class="card m-1 p-1 bg-light">
        <h4>
          {{product.name}}
          <span class="badge badge-pill badge-primary float-right">
            {{ product.price | currency:"USD":"symbol":"2.2-2" }}
          </span>
        </h4>
        <div class="card-text bg-white p-1">{{product.description}}</div>
      </div>
    </div>
  </div>
```

```

</div>

<div class="form-inline float-left mr-1">
  <select class="form-control" [value]="productsPerPage"
    (change)="changePageSize($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-right">
  <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 7-26, I have replaced the array of numbers with a simple number that provides the expression value.

**Listing 7-26.** Supporting the Custom Directive in the `store.component.ts` File in the `src/app/store` Folder

```

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

@Component({
  selector: "store",
  templateUrl: "store.component.html"
})
export class StoreComponent {
  public selectedCategory = null;
  public productsPerPage = 4;
  public selectedPage = 1;

  constructor(private repository: ProductRepository) {}

  get products(): Product[] {
    let pageIndex = (this.selectedPage - 1) * this.productsPerPage
    return this.repository.getProducts(this.selectedCategory)
      .slice(pageIndex, pageIndex + this.productsPerPage);
  }

  get categories(): string[] {
    return this.repository.getCategories();
  }
}

```

```

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

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

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

get pageCount(): number {
    return Math.ceil(this.repository
        .getProducts(this.selectedCategory).length / this.productsPerPage)
}

//get pageNumbers(): number[] {
//    return Array(Math.ceil(this.repository
//        .getProducts(this.selectedCategory).length / this.productsPerPage))
//        .fill(0).map((x, i) => i + 1);
//}
}

```

There is no visual change to the SportsStore application, but this section has demonstrated this it is possible to supplement the built-in Angular functionality with custom code that is tailored to the needs of a specific project.

## Summary

In this chapter, I started the SportsStore project. The early part of the chapter was spent creating the foundation for the project, including creating the root building blocks for the application and starting work on the feature modules. Once the foundation was in place, I was able to rapidly add features to display the dummy model data to the user, add pagination, and filter the products by category. I finished the chapter by creating a custom directive to demonstrate how the built-in features provided by Angular can be supplemented by custom code. In the next chapter, I continue to build the SportsStore application.