CHAPTER 10

SportsStore: Progressive Features and Deployment

In this chapter, I prepare the SportsStore application for deployment by adding progressive features that will allow it to work while offline and show you how to prepare and deploy the application into a Docker container, which can be used on most hosting platforms.

Preparing the Example Application

No preparation is required for this chapter, which continues using the SportsStore project from Chapter 9.

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`.

Adding Progressive Features

A *progressive web application* (PWA) is one that behaves more like a native application, which means it can continue working when there is no network connectivity, its code and content are cached so it can start immediately, and it can use features such as notifications. Progressive web application features are not specific to Angular, but in the sections that follow, I add progressive features to the SportsStore application to show you how it is done.

Tip The process for developing and testing a PWA can be laborious because it can be done only when the application is built for production, which means that the automatic build tools cannot be used.

Installing the PWA Package

The Angular team provides an NPM package that can be used to bring PWA features to Angular projects. Run the command shown in Listing 10-1 in the `SportsStore` folder to download and install the PWA package.

A. Freeman, *Pro Angular 6*, https://doi.org/10.1007/978-1-4842-3649-9_10

■ **Tip** Notice that this command is `ng add`, rather than the `npm install` command that I use elsewhere for adding packages. The `ng add` command is used specifically to install packages, such as `@angular/pwa`, that have been designed to enhance or reconfigure an Angular project.

Listing 10-1. Installing a Package

```
ng add @angular/pwa
```

Caching the Data URLs

The `@angular/pwa` package configures the application so that HTML, JavaScript, and CSS files are cached, which will allow the application to be started even when there is no network available. I also want the product catalog to be cached so that the application has data to present to the user. In Listing 10-2, I added a new section to the `ngsw-config.json` file, which is used to configure the PWA features for an Angular application and is added to the project by the `@angular/pwa` package.

Listing 10-2. Caching the Data URLs in the ngsw-config.json File in the SportsStore Folder

```
{
  "index": "/index.html",
  "assetGroups": [{
    "name": "app",
    "installMode": "prefetch",
    "resources": {
      "files": [
        "/favicon.ico",
        "/index.html",
        "/*.css",
        "/*.js"
      ]
    }
  }, {
    "name": "assets",
    "installMode": "lazy",
    "updateMode": "prefetch",
    "resources": {
      "files": [
        "/assets/**",
        "/font/*"
      ]
    }
  }],
  "dataGroups": [
    {
        "name": "api-product",
        "urls": ["/api/products"],
```

```
        "cacheConfig" : {
            "maxSize": 100,
            "maxAge": "5d"
        }
    }],
    "navigationUrls": [
      "/**"
    ]
}
```

The PWA's code and content required to run the application are cached and updated when new versions are available, ensuring that updates are applied consistently when they are available, using the configuration in the `assetGroups` section of the configuration file. In addition, I have added an entry so that the files required by the Font Awesome package are cached.

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 5 days. The final configuration section is `navigationUrls`, which specifies the range of URLs that will be directed to the `index.html` file. In this example, I used a wildcard to match all URLs.

■ **Note** I am just touching the surface of the cache features that you can use in a PWA. There are lots of choices available, including the ability to try to connect to the network and then fall back to cached data if there is no connection. See https://angular.io/guide/service-worker-intro for details.

Responding to Connectivity Changes

The SportsStore application isn't an ideal candidate for progressive features because connectivity is required to place an order. To avoid user confusion when the application is running without connectivity, I am going to disable the checkout process. The APIs that are used to add progressive features provide information about the state of connectivity and send events when the application goes offline and online. To provide the application with details of its connectivity, I added a file called `connection.service.ts` to the `src/app/model` folder and used it to define the service shown in Listing 10-3.

Listing 10-3. The Contents of the connection.service.ts File in the src/app/model Folder

```
import { Injectable } from "@angular/core";
import { Observable, Subject } from "rxjs";

@Injectable()
export class ConnectionService {
    private connEvents: Subject<boolean>;

    constructor() {
        this.connEvents = new Subject<boolean>();
        window.addEventListener("online",
            (e) => this.handleConnectionChange(e));
        window.addEventListener("offline",
            (e) => this.handleConnectionChange(e));
    }
```

```
    private handleConnectionChange(event) {
        this.connEvents.next(this.connected);
    }

    get connected() : boolean {
        return window.navigator.onLine;
    }

    get Changes(): Observable<boolean> {
        return this.connEvents;
    }
}
```

This service presets the connection status to the rest of the application, obtaining the status through the browser's `navigator.onLine` property and responding to the `online` and `offline` events, which are triggered when the connection state changes and which are accessed through the `addEventListener` method provided by the browser. In Listing 10-4, I added the new service to the module for the data model.

Listing 10-4. Adding a Service in the model.module.ts File in the src/app/model Folder

```
import { NgModule } from "@angular/core";
import { ProductRepository } from "./product.repository";
import { StaticDataSource } from "./static.datasource";
import { Cart } from "./cart.model";
import { Order } from "./order.model";
import { OrderRepository } from "./order.repository";
import { RestDataSource } from "./rest.datasource";
import { HttpClientModule } from "@angular/common/http";
import { AuthService } from "./auth.service";
import { ConnectionService } from "./connection.service";

@NgModule({
  imports: [HttpClientModule],
  providers: [ProductRepository, Cart, Order, OrderRepository,
    { provide: StaticDataSource, useClass: RestDataSource },
    RestDataSource, AuthService, ConnectionService]
})
export class ModelModule { }
```

To prevent the user from checking out when there is no connection, I updated the cart detail component so that it receives the connection service in its constructor, as shown in Listing 10-5.

Listing 10-5. Receiving a Service in the cartDetail.component.ts File in the src/app/store Folder

```
import { Component } from "@angular/core";
import { Cart } from "../model/cart.model";
import { ConnectionService } from "../model/connection.service";

@Component({
    templateUrl: "cartDetail.component.html"
})
export class CartDetailComponent {
```

```
    public connected: boolean = true;

    constructor(public cart: Cart, private connection: ConnectionService) {
        this.connected = this.connection.connected;
        connection.Changes.subscribe((state) => this.connected = state);
    }
}
```

The component defines a connected property that is set from the service and then updated when changes are received. To complete this feature, I changed the checkout button so that it is disabled when there is no connectivity, as shown in Listing 10-6.

Listing 10-6. Reflecting Connectivity in the cartDetail.component.html File in the src/app/store Folder

```
...
<div class="row">
  <div class="col">
  <div class="text-center">
    <button class="btn btn-primary m-1" routerLink="/store">
        Continue Shopping
    </button>
    <button class="btn btn-secondary m-1" routerLink="/checkout"
            [disabled]="cart.lines.length == 0 && connected">
      {{  connected ?  'Checkout' : 'Offline' }}
    </button>
  </div>
</div>
...
```

Preparing the Application for Deployment

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

Creating the Data File

When I created the RESTful web service in Chapter 8, 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 10-7. When the `json-server` package is configured to use a JSON file, any changes that are made by the application will be persisted.

Listing 10-7. The Contents of the serverdata.json File in the SportsStore Folder

```
{
    "products": [
        { "id": 1, "name": "Kayak", "category": "Watersports",
            "description": "A boat for one person", "price": 275 },
        { "id": 2, "name": "Lifejacket", "category": "Watersports",
            "description": "Protective and fashionable", "price": 48.95 },
        { "id": 3, "name": "Soccer Ball", "category": "Soccer",
```

```
            "description": "FIFA-approved size and weight", "price": 19.50 },
        { "id": 4, "name": "Corner Flags", "category": "Soccer",
            "description": "Give your playing field a professional touch",
            "price": 34.95 },
        { "id": 5, "name": "Stadium", "category": "Soccer",
            "description": "Flat-packed 35,000-seat stadium", "price": 79500 },
        { "id": 6, "name": "Thinking Cap", "category": "Chess",
            "description": "Improve brain efficiency by 75%", "price": 16 },
        { "id": 7, "name": "Unsteady Chair", "category": "Chess",
            "description": "Secretly give your opponent a disadvantage",
            "price": 29.95 },
        { "id": 8, "name": "Human Chess Board", "category": "Chess",
            "description": "A fun game for the family", "price": 75 },
        { "id": 9, "name": "Bling Bling King", "category": "Chess",
            "description": "Gold-plated, diamond-studded King", "price": 1200 }
    ],
    "orders": []
}
```

Creating the Server

When the application is deployed, I am going to use a single HTTP port to handle the requests for the application and its data, rather than the two ports that I have been using in development. Using separate ports is simpler in development because it means that I can use the Angular development HTTP server without having to integrate the RESTful web service. Angular doesn't provide an HTTP server for deployment, and since I have to provide one, I am going to configure it so that it will handle both types of request and include support for HTTP and HTTPS connections, as explained in the sidebar.

USING SECURE CONNECTIONS FOR PROGRESSIVE WEB APPLICATIONS

When you add progressive features to an application, you must deploy it so that it can be accessed over secure HTTP connections. If you do not, the progressive features will not work because the underlying technology—called *service workers*—won't be allowed by the browser over regular HTTP connections.

You can test progressive features using localhost, as I demonstrate shortly, but an SSL/TLS certificate is required when you deploy the application. If you do not have a certificate, then a good place to start is `https://letsencrypt.org`, where you can get one for free, although you should note that you also need to own the domain or hostname that you intend to deploy to generate a certificate. For the purposes of this book, I deployed the SportsStore application with its progressive features to `sportsstore.adam-freeman.com`, which is a domain that I use for development testing and receiving emails. This is not a domain that provides public HTTP services, and you won't be able to access the SportsStore application through this domain.

Run the commands shown in Listing 10-8 in the `SportsStore` folder to install the packages that are required to create the HTTP/HTTPS server.

Listing 10-8. Installing Additional Packages

```
npm install --save-dev express@4.16.3
npm install --save-dev connect-history-api-fallback@1.5.0
npm install --save-dev https@1.0.0
```

I added a file called `server.js` to the SportsStore with the content shown in Listing 10-9, which uses the newly added packages to create an HTTP and HTTPS server that includes the `json-server` functionality that will provide the RESTful web service. (The `json-server` package is specifically designed to be integrated into other applications.)

Listing 10-9. The Contents of the server.js File in the SportsStore Folder

```
const express = require("express");
const https = require("https");
const fs = require("fs");
const history = require("connect-history-api-fallback");
const jsonServer = require("json-server");
const bodyParser = require('body-parser');
const auth = require("./authMiddleware");
const router = jsonServer.router("serverdata.json");

const enableHttps = true;

const ssloptions = {}

if (enableHttps) {
    ssloptions.cert =  fs.readFileSync("./ssl/sportsstore.crt");
    ssloptions.key = fs.readFileSync("./ssl/sportsstore.pem");
}

const app = express();

app.use(bodyParser.json());
app.use(auth);
app.use("/api", router);
app.use(history());
app.use("/", express.static("./dist/SportsStore"));

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

if (enableHttps) {
    https.createServer(ssloptions, app).listen(443,
        () => console.log("HTTPS Server running on port 443"));
} else {
    console.log("HTTPS disabled")
}
```

The server is configured to read the details of the SSL/TLS certificate from files in the `ssl` folder, which is where you should place the files for your certificate. If you do not have a certificate, then you can disable HTTPS by setting the `enableHttps` value to `false`. You will still be able to test the application using the local server, but you won't be able to use the progressive features in deployment.

Changing the Web Service URL in the Repository Class

Now that the RESTful data and the application's JavaScript and HTML content will be delivered by the same server, I need to change the URL that the application uses to get its data, as shown in Listing 10-10.

Listing 10-10. Changing the URL in the rest.datasource.ts File in the src/app/model Folder

```
import { Injectable } from "@angular/core";
import { HttpClient } from "@angular/common/http";
import { Observable } from "rxjs";
import { Product } from "./product.model";
import { Cart } from "./cart.model";
import { Order } from "./order.model";
import { map } from "rxjs/operators";
import { HttpHeaders } from '@angular/common/http';

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

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

    constructor(private http: HttpClient) {
        //this.baseUrl = `${PROTOCOL}://${location.hostname}:${PORT}/`;
        this.baseUrl = "/api/"
    }

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

Building and Testing the Application

To build the application for production, run the command shown in Listing 10-11 in the SportsStore folder.

Listing 10-11. Building the Application for Production

```
ng build --prod
```

This command builds an optimized version of the application without the additions that support the development tools. The output from the build process is placed in the `dist/SportsStore` folder. In addition to the JavaScript files, there is an `index.html` file that has been copied from the `SportsStore/src` folder and modified to use the newly built files.

■ **Note** Angular provides support for server-side rendering, where the application is run in the server, rather than the browser. This is a technique that can improve the perception of the application's startup time and can improve indexing by search engines. This is a feature that should be used with caution because it has serious limitations and can undermine the user experience. For these reasons, I have not covered server-side rendering in this book. You can learn more about this feature at `https://angular.io/guide/universal`.

The build process can take a few minutes to complete. Once the build is ready, run the command shown in Listing 10-12 in the SportsStore folder to start the HTTP server. If you have not configured the server to use a valid SSL/TLS certificate, then you should change the value of the `enableHttps` constant in the `server.js` file and then run the command in Listing 10-12.

Listing 10-12. Starting the Production HTTP Server

```
node server.js
```

Once the server has started, open a new browser window and navigate to `http://localhost`, and you will see the familiar content shown in Figure 10-1.

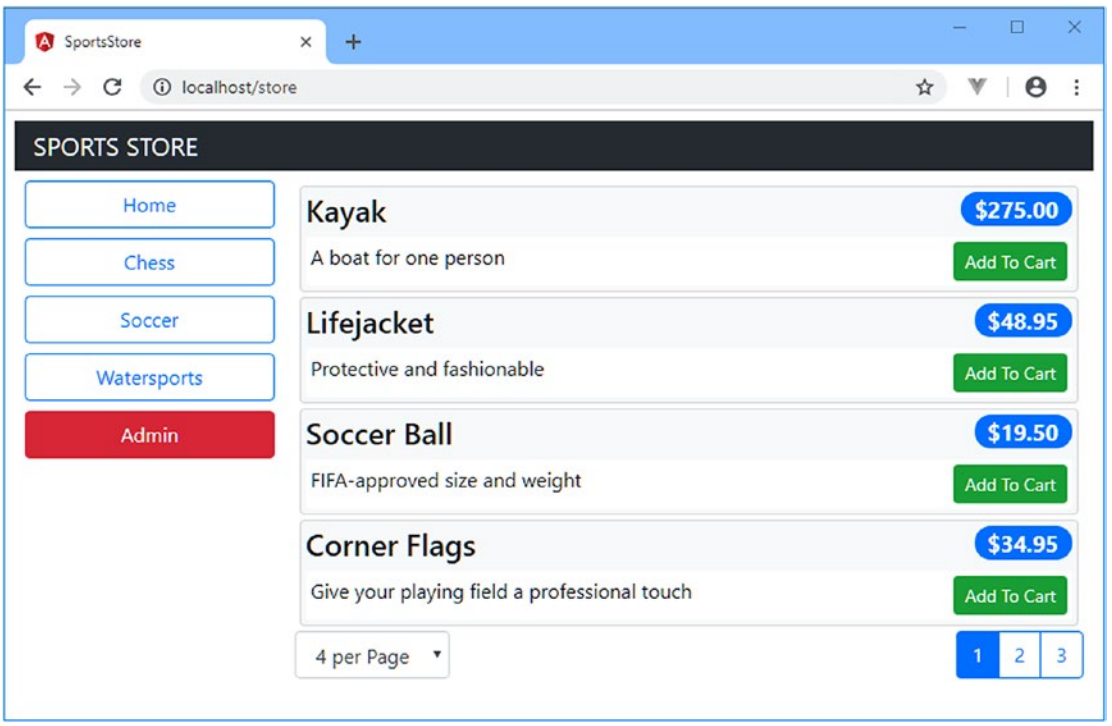

Figure 10-1. *Testing the application*

Testing the Progressive Features

Open the F12 development tools, navigate to the Network tab and check the Offline button, as shown in Figure 10-2. 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.

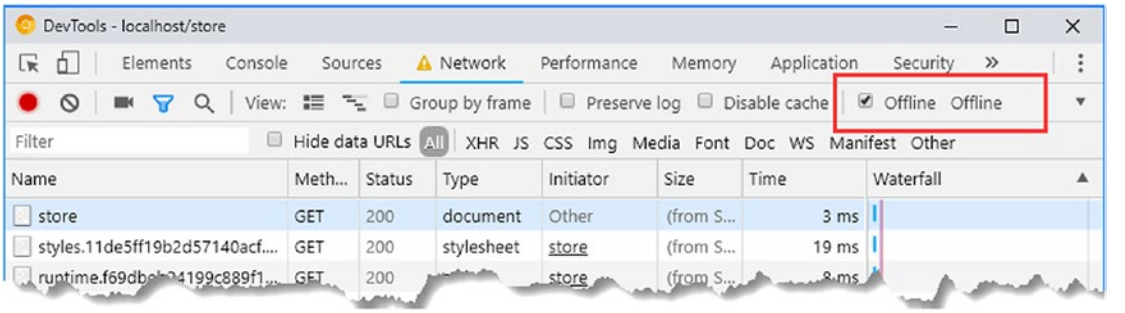

Figure 10-2. *Going offline*

Once the application is offline, click the browser reload button, and 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 10-3. Uncheck the Offline checkbox, and the button's text will change so that the user can place an order.

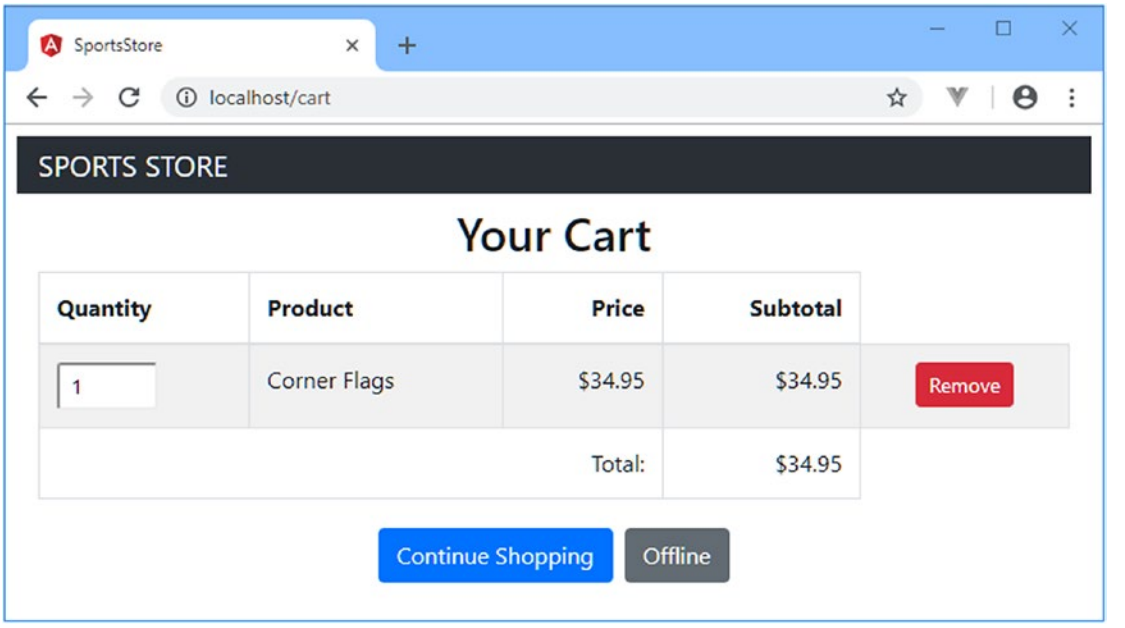

Figure 10-3. *Reflecting the connection status in the application*

Containerizing the SportsStore Application

To complete this chapter, I am going to create a container for the SportsStore application so that it can be deployed into production. At the time of writing, Docker is the most popular way to create containers, which is a pared-down version of Linux with just enough functionality to run the application. Most cloud platforms or hosting engines have support for Docker, and its tools run on the most popular operating systems.

Installing Docker

The first step is to download and install the Docker tools on your development machine, which is available from www.docker.com/products/docker. There are versions for macOS, Windows, and Linux, and there are some specialized versions to work with the Amazon and Microsoft cloud platforms. The free Community edition is sufficient for this chapter.

■ **Caution** One drawback of using Docker is that the company that produces the software has gained a reputation for making breaking changes. This may mean that the example that follows may not work as intended with later versions. If you have problems, check the repository for this book for updates (https://github.com/Apress/pro-angular-6) or contact me at adam@adam-freeman.com.

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 10-13.

Listing 10-13. The Contents of the deploy-package.json File in the SportsStore Folder

```
{
  "dependencies": {
    "bootstrap": "4.1.1",
    "font-awesome": "4.7.0"
  },

  "devDependencies": {
    "json-server": "0.12.1",
    "jsonwebtoken": "8.1.1",
    "express": "^4.16.3",
    "https": "^1.0.0",
    "connect-history-api-fallback": "^1.5.0"
  },

  "scripts": {
    "start":  "node server.js"
  }
}
```

The dependencies section omits Angular and all of the other runtime packages that were added to the package.json file when the project was created because the build process incorporates all of the JavaScript code required by the application into the files in the dist/SportsStore folder. The devDependencies section includes the tools required by the production HTTP/HTTPS server.

The scripts section of the deploy-package.json file is set up so that the npm start command will start the production server, which will provide access to the application and its data.

Creating the Docker Container

To define the container, I added a file called Dockerfile (with no extension) to the SportsStore folder and added the content shown in Listing 10-14.

Listing 10-14. The Contents of the Dockerfile File in the SportsStore Folder

```
FROM node:8.11.2

RUN mkdir -p /usr/src/sportsstore

COPY dist/SportsStore /usr/src/sportsstore/dist/SportsStore
COPY ssl /usr/src/sportsstore/ssl

COPY authMiddleware.js /usr/src/sportsstore/
COPY serverdata.json /usr/src/sportsstore/
COPY server.js /usr/src/sportsstore/server.js
COPY deploy-package.json /usr/src/sportsstore/package.json

WORKDIR /usr/src/sportsstore

RUN npm install

EXPOSE 80

CMD ["node", "server.js"]
```

The contents of the Dockerfile use a base image that has been configured with Node.js and copies the files required to run the application, including the bundle file containing the application and the package.json file that will be used to install the packages required to run the application in deployment.

To speed up the containerization process, I created a file called .dockerignore in the SportsStore folder with the content shown in Listing 10-15. This tells Docker to ignore the node_modules folder, which is not required in the container and takes a long time to process.

Listing 10-15. The Contents of the .dockerignore File in the SportsStore Folder

```
node_modules
```

Run the command shown in Listing 10-16 in the SportsStore folder to create an image that will contain the SportsStore application, along with all of the tools and packages it requires.

Listing 10-16. Building the Docker Image

```
docker build . -t sportsstore  -f  Dockerfile
```

An image is a template for containers. As Docker processes the instructions in the Docker file, the NPM packages will be downloaded and installed, and the configuration and code files will be copied into the image.

Running the Application

Once the image has been created, create and start a new container using the command shown in Listing 10-17.

■ **Tip** Make sure you stop the test server you started in Listing 10-12 before starting the Docker container since both use the same ports to listen for requests.

Listing 10-17. Starting the Docker Container

```
docker run -p 80:80 -p 443:443 sportsstore
```

You can test the application by opening `http://localhost` in the browser, which will display the response provided by the web server running in the container, as shown in Figure 10-4.

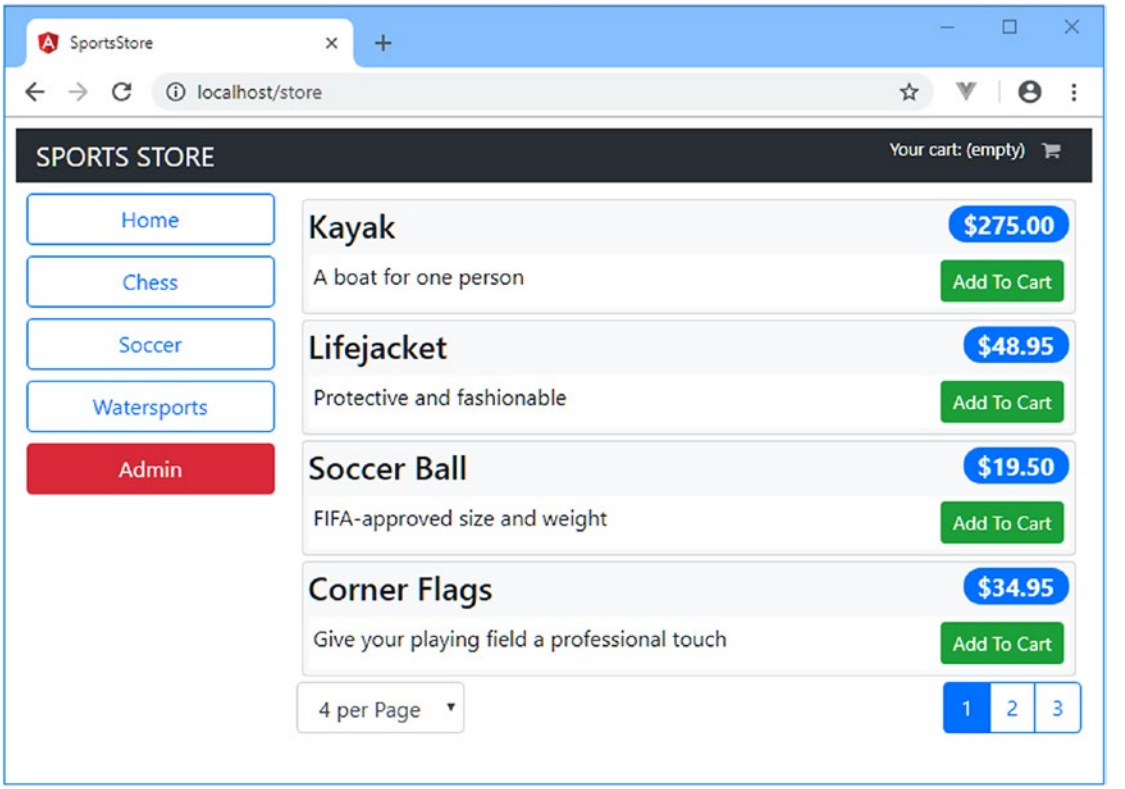

Figure 10-4. *Running the containerized SportsStore application*

To stop the container, run the command shown in Listing 10-18.

Listing 10-18. Listing the Containers

```
docker ps
```

You will see a list of running containers, like this (I have omitted some fields for brevity):

```
CONTAINER ID        IMAGE               COMMAND             CREATED
ecc84f7245d6        sportsstore         "node server.js"    33 seconds ago
```

Using the value in the Container ID column, run the command shown in Listing 10-19.

Listing 10-19. Stopping the Container

```
docker stop ecc84f7245d6
```

The application is ready to deploy to any platform that supports Docker, although the progressive features will work only if you have configured an SSL/TLS certificate for the domain to which the application is deployed.

Summary

This chapter completes the SportsStore application, showing how an Angular application can be prepared for deployment and how easy it is to put an Angular application into a container such as Docker. That's the end of this part of the book. In Part 2, I begin the process of digging into the details and show you how the features I used to create the SportsStore application work in depth.