CHAPTER 5

SportsStore: A Real Application

In Chapter 2, I built a quick and simple React application. Small and focused examples allow me to demonstrate specific 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 details and place their orders. I will also create an administration area that includes create, read, update, and delete (CRUD) facilities for managing products and orders—and I will protect it so that only logged-in administrators can make changes. Finally, I show you how to prepare a React application for deployment.

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

The SportsStore example is one that I use in all 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* or *Pro Angular 6* book, for example.

The React 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 React works or jump to and from the detail chapters to get into the depth.

Either way, don't expect to understand everything right away—React applications have a lot of moving parts and depend on a lot of packages, and the SportsStore application is intended to show you how they fit together without diving too deeply into the details that the rest of the book describes.

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-react-16`.

A. Freeman, *Pro React 16*, https://doi.org/10.1007/978-1-4842-4451-7_5

Preparing the Project

To create the project, open a new command prompt, navigate to a convenient location, and run the command shown in Listing 5-1.

Listing 5-1. Creating the SportsStore Project

```
npx create-react-app sportsstore
```

The `create-react-app` tool will create a new React project named `sportsstore` with the packages, configuration files, and placeholder content required to start development. The project setup process may take some time to complete because there is a large number of NPM packages to download and install.

■ **Note** When you create a new project, you may see warnings about security vulnerabilities. React development relies on a large number of packages, each of which has its own dependencies, and security issues will inevitably be discovered. For the examples in this book, it is important to use the package versions specified to ensure you get the expected results. For your own projects, you should review the warnings and update to versions that resolve the problems.

Installing Additional NPM Packages

Additional packages are required for the SportsStore project, in addition to the core React libraries and development tools installed by the `create-react-app` package. Run the commands shown in Listing 5-2 to navigate to the `sportsstore` folder and add the packages. (The `npm install` command can be used to add multiple packages in one go, but the result is a long command where it is easy to omit a package. To avoid errors, I add packages individually throughout this book.)

■ **Note** 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 these can be ignored.

Listing 5-2. Installing Additional Packages

```
cd sportsstore
npm install bootstrap@4.1.2
npm install @fortawesome/fontawesome-free@5.6.1
npm install redux@4.0.1
npm install react-redux@6.0.0
npm install react-router-dom@4.3.1
npm install axios@0.18.0
npm install graphql@14.0.2
npm install apollo-boost@0.1.22
npm install react-apollo@2.3.2
```

Don't be put off by the number of additional packages that are required. React focuses on a core set of the features that are required by web applications and relies on supporting packages to create complete applications. To provide some context, the packages added in Listing 5-2 are described in Table 5-1, and I cover them in depth in Part 3 of this book.

Table 5-1. *The Packages Required for the SportsStore Project*

Name	Description
`bootstrap`	This package provides the CSS styles that I used to present HTML content throughout the book.
`fontawesome-free`	This package provides icons that can be included in HTML content. I have used the free package, but there is a more comprehensive paid-for option available, too.
`redux`	This package provides a data store, which simplifies the process of coordinating the different parts of the application. See Chapter 19 for details.
`react-redux`	This package integrates a Redux data store into a React application, as described in Chapters 19 and 20.
`react-router-dom`	This package provides URL routing, which allows the content presented to the user to be selected based on the browser's current URL, as described in Chapters 21 and 22.
`axios`	This package is used to make HTTP requests and will be used to access RESTful and GraphQL services, as described in Chapters 23–25.
`graphql`	This package contains the reference implementation of the GraphQL specification.
`apollo-boost`	This package contains a client used to consume a GraphQL service, as described in Chapter 25.
`react-apollo`	This package is used to integrate the GraphQL client into a React application, as described in Chapter 25.

Further packages are required to create the back-end services that the SportsStore application will consume. Using the command prompt, run the commands shown in Listing 5-3 in the `sportsstore` folder. These packages are installed using the `--save-dev` argument, which indicates they are used during development and will not be part of the SportsStore application when it is deployed.

Listing 5-3. Adding Further Packages

```
npm install --save-dev json-server@0.14.2
npm install --save-dev jsonwebtoken@8.1.1
npm install --save-dev express@4.16.4
npm install --save-dev express-graphql@0.7.1
npm install --save-dev cors@2.8.5
npm install --save-dev faker@4.1.0
npm install --save-dev chokidar@2.0.4
npm install --save-dev npm-run-all@4.1.3
npm install --save-dev connect-history-api-fallback@1.5.0
```

You won't need these packages for applications that consume data from existing services, but I need to create a complete infrastructure for the SportsStore application. Table 5-2 briefly describes the purpose of each package installed in Listing 5-3.

Table 5-2. *The Additional Packages Required by the SportsStore Project*

Name	Description
`json-server`	This package will be used to provide a RESTful web service in Chapter 6.
`jsonwebtoken`	This package will be used to authenticate users in Chapter 8.
`graphql`	This package will be used to define the schema for the GraphQL server in Chapter 7.
`express`	This package will be used to host the back-end servers.
`express-graphql`	This package will be used to create a GraphQL server.
`cors`	This package is used to enable cross-origin request sharing (CORS) requests.
`faker`	This package generates fake data for testing and is used in Chapter 6.
`chokidar`	This package monitors files for changes.
`npm-run-all`	This package is used to run multiple NPM scripts in a single command.
`connect-history-api-fallback`	This package is used to respond to HTTP requests with the `index.html` file and is used in the production server in Chapter 8.

Adding the CSS Stylesheets to the Project

To use the Bootstrap and Font Awesome packages, I need to add `import` statements to the application's `index.js` file. The purpose of the `index.js` file is to start the application, as described in Chapter 9, and adding the import statements shown in Listing 5-4 ensures that the styles I require can be applied to the HTML content presented by the SportsStore application.

Listing 5-4. Adding CSS Stylesheets in the index.js File in the src Folder

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import * as serviceWorker from './serviceWorker';
import "bootstrap/dist/css/bootstrap.css";
import "@fortawesome/fontawesome-free/css/all.min.css";

ReactDOM.render(<App />, document.getElementById('root'));

// If you want your app to work offline and load faster, you can change
// unregister() to register() below. Note this comes with some pitfalls.
// Learn more about service workers: http://bit.ly/CRA-PWA
serviceWorker.unregister();
```

Preparing the Web Service

Once the basic structure of the application is in place, I will add support for consuming data from a web service. In preparation, I added a file called `data.js` to the `sportsstore` folder with the content shown in Listing 5-5.

Listing 5-5. The Contents of the data.js File in the sportsstore Folder

```
module.exports = function () {
    return {
        categories: ["Watersports", "Soccer", "Chess"],
        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: []
    }
}
```

The code in Listing 5-5 creates three data collections that will be used by the application. The `products` collection contains the products for sale to the customer, the `categories` collection contains the set of categories into which the products are organized, and the `orders` collection contains the orders that customers have placed (but is currently empty).

I added a file called `server.js` to the `sportsstore` folder with the code shown in Listing 5-6. This is the code that creates the web service that will provide the application with data. I add features to the back-end server, such as authentication and support for GraphQL, in later chapters.

Listing 5-6. The Contents of the server.js File in the sportsstore Folder

```
const express = require("express");
const jsonServer = require("json-server");
const chokidar = require("chokidar");
const cors = require("cors");
```

```
const fileName = process.argv[2] || "./data.js"
const port = process.argv[3] || 3500;

let router = undefined;

const app = express();

const createServer = () => {
    delete require.cache[require.resolve(fileName)];
    setTimeout(() => {
        router = jsonServer.router(fileName.endsWith(".js")
            ? require(fileName)() : fileName);
    }, 100)
}

createServer();

app.use(cors());
app.use(jsonServer.bodyParser)
app.use("/api", (req, resp, next) => router(req, resp, next));

chokidar.watch(fileName).on("change", () => {
    console.log("Reloading web service data...");
    createServer();
    console.log("Reloading web service data complete.");
});

app.listen(port, () => console.log(`Web service running on port ${port}`));
```

To ensure that the web service is started alongside the React development tools, I changed the `scripts` section of the `package.json` file, as shown in Listing 5-7.

Listing 5-7. Enabling the Web Service in the package.json File in the sportsstore Folder

```
...
"scripts": {
  "start": "npm-run-all --parallel reactstart webservice",
  "reactstart": "react-scripts start",
  "webservice": "node server.js",
  "build": "react-scripts build",
  "test": "react-scripts test",
  "eject": "react-scripts eject"
},
...
```

This change uses the `npm-run-all` package to run the React development server and the web service at the same time.

Running the Example Application

To start the application and the web service, use the command prompt to run the command shown in Listing 5-8 in the `sportsstore` folder.

Listing 5-8. Starting the Application

```
npm start
```

There will be a pause while the initial compilation is completed, and then a new browser window will open displaying the placeholder content shown in Figure 5-1.

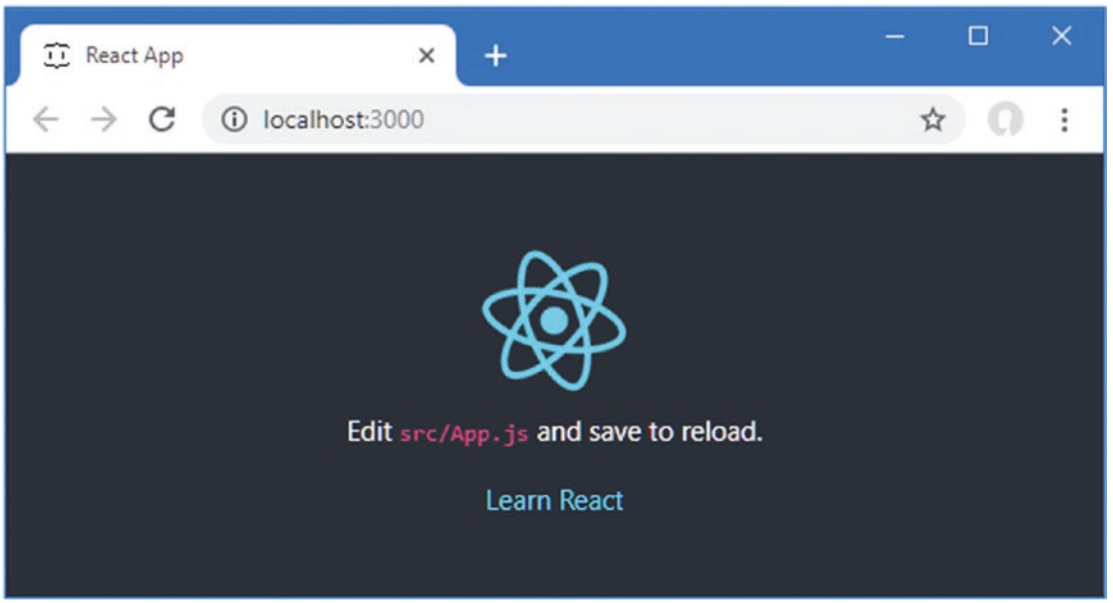

Figure 5-1. *Running the example application*

To make sure that the web service is running, open a new browser window and request the URL `http://localhost:3500/api/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 }
```

Creating the Data Store

The starting point for SportsStore is the data store, which will be the repository for the data presented to the user and the supporting details required to coordinate application features, such as pagination.

I am going to start with a data store that uses local placeholder data. Later, I will add support for getting the data from a web service, but static data is a good place to start because it keeps the focus on the React application. The SportsStore data store will be created using the Redux package, which is the most popular data store for React projects and which I describe in Chapters 19 and 20. To get started, I created the `src/data` folder and added to it a file called `placeholderData.js`, with the content shown in Listing 5-9.

Listing 5-9. The Contents of the placeholderData.js File in the src/data Folder

```
export const data = {
    categories: ["Watersports", "Soccer", "Chess", "Running"],
    products: [
        { id: 1, name: "P1", category: "Watersports",
            description: "P1 (Watersports)", price: 3 },
        { id: 2, name: "P2", category: "Watersports",
           description: "P2 (Watersports)", price: 4 },
        { id: 3, name: "P3", category: "Running",
           description: "P3 (Running)", price: 5 },
        { id: 4, name: "P4", category: "Chess",
           description: "P4 (Chess)", price: 6 },
        { id: 5, name: "P5", category: "Chess",
           description: "P6 (Chess)", price: 7 },
    ]
}
```

Creating the Data Store Actions and Action Creators

Redux data stores separate reading data from the operations that change it. This can feel awkward at first, but it is similar to other parts of React development, such as component state data and using GraphQL, and it quickly becomes second nature.

Actions are objects that are sent to the data store to make changes to the data it contains. Actions have types, and action objects are created using *action creators*. The only action I need at the moment will load the data into the store, initially using the placeholder data defined in Listing 5-9 but eventually from a web service. There are different ways you can structure the actions for a data store, but it is worth identifying the common themes that are shared between different types of data to avoid code duplication later. I added a file called `Types.js` in the `src/data` folder and used it to list the data types in the store and the set of actions that can be performed on them, as shown in Listing 5-10.

Listing 5-10. The Contents of the Types.js File in the src/data Folder

```
export const DataTypes = {
    PRODUCTS: "products",
    CATEGORIES: "categories"
}

export const ActionTypes = {
    DATA_LOAD: "data_load"
}
```

There are two data types—PRODUCTS and CATEGORIES—and a single action, DATA_LOAD, which will populate the data store. There is no requirement to defined action types this way, but using constant values avoids typos when specifying action types elsewhere in the application.

Next, I need to define an *action creator* function, which will create an action object that can be processed by the data store to alter the data it contains. I added a file called ActionCreators.js to the src/data folder, with the code shown in Listing 5-11.

Listing 5-11. The Contents of the ActionCreators.js File in the src/data Folder

```
import { ActionTypes} from "./Types";
import { data as phData} from "./placeholderData";

export const loadData = (dataType) => ({
    type: ActionTypes.DATA_LOAD,
    payload: {
        dataType: dataType,
        data: phData[dataType]
    }
});
```

The use of action creators is described in Chapter 19, but the only requirement for the objects produced by action creators is they must have a type property whose value specifies the type of change required to the data store. It is a good idea to use a common set of properties in action objects so that they can be handled consistently, and the action creator defined in Listing 5-11 returns an action object that has a payload property, which is the convention I will use for all of the SportsStore data store actions.

The payload property for the action object in Listing 5-11 has a dataType property that indicates the type of data that the action relates to and a data property that provides the data to be added to the data store. The value for the data property is obtained from the placeholder data at the moment, but I replace this with data obtained from a web service in Chapter 6.

Actions are processed by data store *reducers*, which are functions that receive the current contents of the data store and an action object and use them to make changes. I added a file called ShopReducer.js to the src/data folder and defined the reducer shown in Listing 5-12.

Listing 5-12. The Contents of the ShopReducer.js File in the src/data Folder

```
import { ActionTypes } from "./Types";
export const ShopReducer = (storeData, action) => {
    switch(action.type) {
        case ActionTypes.DATA_LOAD:
            return {
                ...storeData,
                [action.payload.dataType]: action.payload.data
            };
        default:
            return storeData || {};
    }
}
```

Reducers are required to create and return new objects that incorporate any required changes. If the action type isn't recognized, the reducer must return the data store object it received unchanged. The reducer in Listing 5-12 handles the DATA_LOAD action by creating a new object with all the properties of the old store plus the new data received in the action. Reducers are described in more detail in Chapter 19.

As the final step for creating the data store, I added a file called `DataStore.js` to the `src/data` folder and added the code shown in Listing 5-13.

Listing 5-13. The Contents of the DataStore.js File in the src/data Folder

```
import { createStore } from "redux";
import { ShopReducer } from "./ShopReducer";

export const SportsStoreDataStore = createStore(ShopReducer);
```

The Redux package provides the `createStore` function, which sets up a new data store using a reducer. This is enough to create a data store to get started with, but I will add additional features later so that further operations can be performed and so that data can be loaded from a web service.

Creating the Shopping Features

The first part of the application that will be seen by users is the storefront, which will present the available products in a two-column layout that allows filtering by category, as shown in Figure 5-2.

Figure 5-2. *The basic structure of the application*

I am going to structure the application so that the browser's URL is used to select the content presented to the user. To get started, the application will support the URLs described in Table 5-3, which will allow the user to see the products for sale and filter them by category.

Table 5-3. *The SportsStore URLs*

Name	Description
`/shop/products`	This URL will display all of the products to the user, regardless of category.
`/shop/products/chess`	This URL will display the products in a specific category. In this case, the URL will select the Chess category.

■ **Note** I have adopted the British term *shop* for the part of the application that offers products for sale to customers. I want to avoid confusion between the data store, in which the application's data is kept, and the product store, from which the user makes purchases.

Responding to the browser's URL in the application is known as *URL routing*, which is provided by the React Router package added in Listing 5-2, and which is described in detail in Chapters 21 and 22.

Creating the Product and Category Components

Components are the building blocks for React applications and are responsible for the content presented to the user. I created the `src/shop` folder and added to it a file called `ProductList.js` with the contents shown in Listing 5-14.

Listing 5-14. The Contents of the ProductList.js File in the src/shop Folder

```
import React, { Component } from "react";

export class ProductList extends Component {

    render() {
        if (this.props.products == null || this.props.products.length === 0) {
            return <h5 className="p-2">No Products</h5>
        }
        return this.props.products.map(p =>
                <div className="card m-1 p-1 bg-light" key={ p.id }>
                    <h4>
                        { p.name }
                        <span className="badge badge-pill badge-primary float-right">
                            ${ p.price.toFixed(2) }
                        </span>
                    </h4>
                    <div className="card-text bg-white p-1">
                        { p.description }
                    </div>
                </div>
            )
    }
}
```

Components are created to perform small tasks or display small amounts of content and are combined to create more complex features. The `ProductList` component defined in Listing 5-14 is responsible for displaying details of a list of products, whose details are received through a prop named `product`. Props are used to configure components and allow them to do their work—such as display details of a product—without getting involved in where the data comes from. The `ProductList` component generates HTML content that includes the value of each product's `name`, `price`, and `description` properties, but it doesn't have knowledge of how those products are defined in the application or whether they have been defined locally or retrieved from a remote server.

Next, I added a file called CategoryNavigation.js to the src/shop folder and defined the component shown in Listing 5-15.

Listing 5-15. The Contents of the CategoryNavigation.js File in the src/shop Folder

```
import React, { Component } from "react";
import { Link } from "react-router-dom";

export class CategoryNavigation extends Component {

    render() {
        return <React.Fragment>
            <Link className="btn btn-secondary btn-block"
                to={ this.props.baseUrl }>All</Link>
            { this.props.categories && this.props.categories.map(cat =>
                <Link className="btn btn-secondary btn-block" key={ cat }
                    to={ `${this.props.baseUrl}/${cat.toLowerCase()}`}>
                    { cat }
                </Link>
            )}
        </React.Fragment>
    }
}
```

The selection of a category will be handled by navigating to a new URL, which is done using the Link component provided by the React Router package. When the user clicks a Link, the browser is asked to navigate to a new URL without sending any HTTP requests or reloading the application. The details included in the new URL, such as the selected category in this case, allow different parts of the application to work together.

The CategoryNavigation component receives the array of categories through a prop named categories. The component checks to ensure that the array has been defined and uses the map method to generate the content for each array item. React requires a key prop to be applied to the elements generated by the map method so that changes to the array can be handled efficiently, as explained in Chapter 10. The result is a Link component for each category that is received in the array with an additional Link so that the user can select all products, regardless of category. The Link components are styled so they appear as buttons, and the URLs that the browser will navigate to are the combination of a prop called baseUrl and the name of the category.

To bring together the product table and the category buttons, I added a file called Shop.js to the src/shop folder and added the code shown in Listing 5-16.

Listing 5-16. The Contents of the Shop.js File in the src/shop Folder

```
import React, { Component } from "react";
import { CategoryNavigation } from "./CategoryNavigation";
import { ProductList } from "./ProductList";

export class Shop extends Component {

    render() {
        return <div className="container-fluid">
            <div className="row">
                <div className="col bg-dark text-white">
```

```
                    <div className="navbar-brand">SPORTS STORE</div>
                </div>
            </div>
            <div className="row">
                <div className="col-3 p-2">
                    <CategoryNavigation baseUrl="/shop/products"
                        categories={ this.props.categories } />
                </div>
                <div className="col-9 p-2">
                    <ProductList products={ this.props.products } />
                </div>
            </div>
        </div>
    }
}
```

A component can delegate responsibility for part of its content to other components. In its `render` method, the `Shop` component defined in Listing 5-16 contains HTML elements that set up a grid structure using Bootstrap CSS classes but delegates responsibility for populating some of the grid cells to the `CategoryNavigation` and `ProductList` components. These delegated components are expressed as custom HTML elements in the `render` method, where the element tag matches the name of the component, like this:

```
...
<ProductList products={ this.props.products } />
...
```

A relationship is created between the two components: the `Shop` component is the parent of the `ProductList`, and the `ProductList` component is the child of the `Shop`. Parents configure their child components by providing props, and in Listing 5-16, the `Shop` component passes on the `products` prop it received from its parent to its `ProductList` child component, which will be used to display the list of products to the user. The relationships between components and the ways they can be used to create complex features are described in Part 2 of this book.

Connecting to the Data Store and the URL Router

The `Shop` component and its `CategoryNavigation` and `ProductList` children need access to the data store. To connect these components to the features they require, I added a file called `ShopConnector.js` to the `src/shop` folder with the code shown in Listing 5-17.

Listing 5-17. The Contents of the ShopConnector.js File in the src/shop Folder

```
import React, { Component } from "react";
import { Switch, Route, Redirect }
    from "react-router-dom"
import { connect } from "react-redux";
import { loadData } from "../data/ActionCreators";
import { DataTypes } from "../data/Types";
import { Shop } from "./Shop";
```

```
const mapStateToProps = (dataStore) => ({
    ...dataStore
})

const mapDispatchToProps = {
    loadData
}

const filterProducts = (products = [], category) =>
    (!category || category === "All")
        ? products
        : products.filter(p => p.category.toLowerCase() === category.toLowerCase());

export const ShopConnector = connect(mapStateToProps, mapDispatchToProps)(
    class extends Component {
        render() {
            return <Switch>
                <Route path="/shop/products/:category?"
                    render={ (routeProps) =>
                        <Shop { ...this.props } { ...routeProps }
                            products={ filterProducts(this.props.products,
                                routeProps.match.params.category) } />} />
                <Redirect to="/shop/products" />
            </Switch>
        }

        componentDidMount() {
            this.props.loadData(DataTypes.CATEGORIES);
            this.props.loadData(DataTypes.PRODUCTS);
        }
    }
)
```

Don't worry if the code in Listing 5-17 seems impenetrable at the moment. The code is more complex than earlier listings because this component brings together and consolidates several features so they can be used more easily elsewhere in the project, as shown in Figure 5-3.

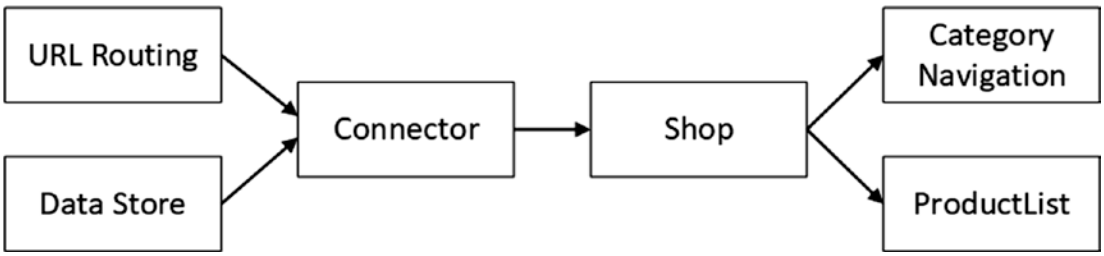

Figure 5-3. *Connecting an application to its services*

The advantage of this approach is that it simplifies adding features or making changes to the application because the components that present content to the user receive their data via props without the need to obtain it directly from the data store or the URL routing system. The disadvantage is that the component that connects the rest of the application to its services can be difficult to write and maintain, as it must combine the features of different packages and present them to its children. The complexity of this component will increase until the end of Chapter 6, when I consolidate the code around the final set of SportsStore shopping features.

The component in Listing 5-17 connects the Redux data store and the URL router to the Shop component. The Redux package provides the `connect` function, which is used to link a component to a data store so that its props are either values from the data store or functions that dispatch data store actions when they are invoked, as described in Chapter 20. It is the `connect` function that has led to much of the code in Listing 5-17 because it requires mappings between the data store and the component's props, which can be verbose. The mappings in Listing 5-17 give the Shop component access to all of the properties defined in the data store, which consists of the product and category data at present but will include other features later.

■ **Tip** You can be more specific in the data store properties you map to props, as demonstrated in Chapter 20, but I have mapped all of the products, which is a useful approach when you start developing a new project because it means you don't have to remember to map new properties each time you enhance the data store.

The product data must be filtered using the selected category, which is accessed through the features provided by the React Router package. A `Route` is used to select the component that will be displayed to the user when the browser navigates to a specific URL. The `Route` in Listing 5-17 matches the URLs from Table 5-3, like this:

```
...
<Route path="/shop/products/:category?" render={ (routeProps) =>
...
```

The `path` prop tells the `Route` to wait until the browser navigates to the `/shop/products` URL. If there is an additional segment in the URL, such as `/shop/products/running`, then the contents of that segment will be assigned to a parameter named `category`, which is how the user's category selection will be determined.

When the browser navigates to a URL that matches the `path` prop, the `Route` displays the content specified by the `render` prop, like this:

```
...
<Route path="/shop/products/:category?" render={ (routeProps) =>
    <Shop { ...this.props } { ...routeProps }
        products={ filterProducts(this.props.products,
            routeProps.match.params.category) } />} />
...
```

This is the point at which the data store and the URL routing features are combined. The Shop component needs to know which category the user has selected, which is available through the argument passed to the `Route` component's render prop. The category is combined with the data from the data store both of which are passed on to the Shop component. The order in which props are applied to a component allows props to be overridden, which I have relied on to replace the `products` data obtained from the data store with the result from the `filterProduct` function, which selects only the products in the category chosen by the user.

The Route is used in conjunction with Switch and Redirect components, both of which are part of the React Router package and which combine to redirect the browser to /shop/products if the browser's current URL isn't matched by the Route.

The ShopConnector component uses the componentDidMount method to load the data into the data store. The componentDidMount method is part of the React component lifecycle, which is described in detail in Chapter 13.

Adding the Shop to the Application

In Listing 5-18, I have set up the data store and the URL routing features and incorporated the ShopConnector component into the application.

Listing 5-18. Adding Routing and a Data Store to the App.js File in the src Folder

```
import React, { Component } from "react";
import { SportsStoreDataStore } from "./data/DataStore";
import { Provider } from "react-redux";
import { BrowserRouter as Router, Route, Switch, Redirect }
    from "react-router-dom";
import { ShopConnector } from "./shop/ShopConnector";

export default class App extends Component {

    render() {
        return <Provider store={ SportsStoreDataStore }>
            <Router>
                <Switch>
                    <Route path="/shop" component={ ShopConnector } />
                    <Redirect to="/shop" />
                </Switch>
            </Router>
        </Provider>
    }
}
```

The data store is applied to the application using a Provider, with the store prop being assigned the data store created in Listing 5-13. The URL routing features are applied to the application using the Router component, which I have supplemented using the Switch, Route, and Redirect components. The Redirect will navigate to the /shop URL, which matches the path prop of the Route and displays the ShopConnector component, producing the result shown in Figure 5-4. Clicking a category button redirects the browser to a new URL, such as /shop/products/watersports, which has the effect of filtering the products that are displayed.

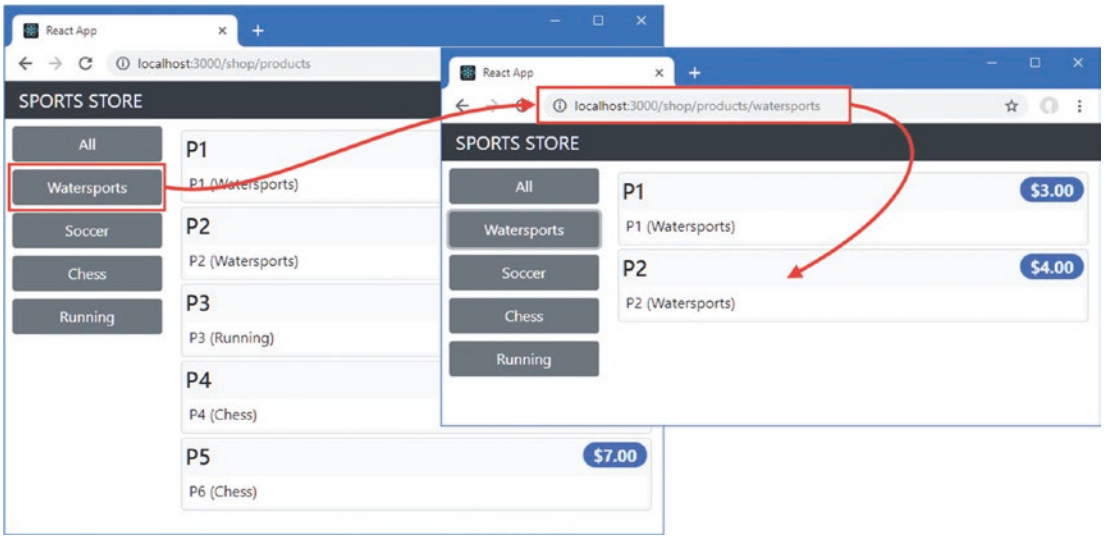

Figure 5-4. *Creating the basic shopping features*

Improving the Category Selection Buttons

The category selection buttons work but don't clearly reflect the current category to the user. To remedy this, I added a file called `ToggleLink.js` to the `src` folder and used it to define the component shown in Listing 5-19.

■ **Tip** I added this component to the `src` folder because I will use it for other parts of the application once the shop has been completed. There are no hard-and-fast rules about how a React project is organized, but I tend to keep related files grouped together in folders.

Listing 5-19. The Contents of the ToggleLink.js File in the src Folder

```
import React, { Component } from "react";
import { Route, Link } from "react-router-dom";

export class ToggleLink extends Component {

    render() {
        return <Route path={ this.props.to } exact={ this.props.exact }
                children={ routeProps => {

            const baseClasses = this.props.className || "m-2 btn btn-block";
            const activeClass = this.props.activeClass || "btn-primary";
            const inActiveClass = this.props.inActiveClass || "btn-secondary"
```

```
            const combinedClasses =
                `${baseClasses} ${routeProps.match ? activeClass : inActiveClass}`

            return <Link to={ this.props.to } className={ combinedClasses }>
                        { this.props.children }
                    </Link>
        }} />
    }
}
```

The React Router package provides a component that indicates when a specific URL has been matched, but it doesn't work well with the Bootstrap CSS classes, as I describe in Chapter 22, where I explain how the `ToggleLink` component works in detail. For this chapter, it is enough to know that the `Route` component can be used to provide access to the URL routing system in order to get details about the current route.
In Listing 5-20, I have updated the `CategoryNavigation` component to use the `ToggleLink` component.

Listing 5-20. Using ToggleLinks in the CategoryNavigation.js File in the src/shop Folder

```
import React, { Component } from "react";
//import { Link } from "react-router-dom";
import { ToggleLink } from "../ToggleLink";

export class CategoryNavigation extends Component {

    render() {
        return <React.Fragment>
            <ToggleLink to={ this.props.baseUrl } exact={ true }>All</ToggleLink>
            { this.props.categories && this.props.categories.map(cat =>
                <ToggleLink key={ cat }
                    to={ `${this.props.baseUrl}/${cat.toLowerCase()}`}>
                    { cat }
                </ToggleLink>
            )}
        </React.Fragment>
    }
}
```

The effect is to clearly indicate which category has been selected, as shown in Figure 5-5.

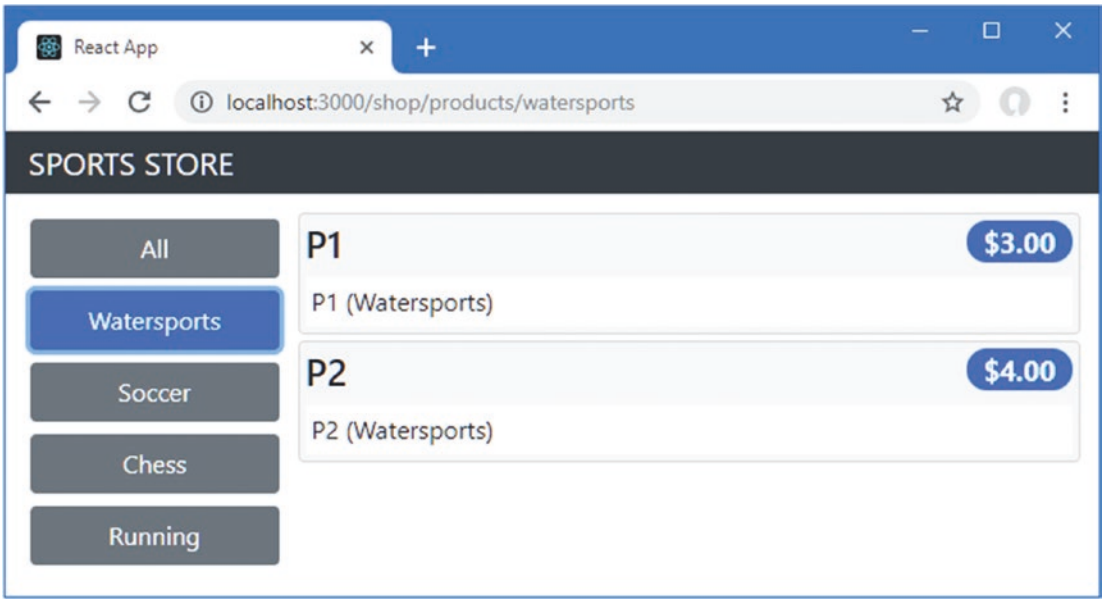

Figure 5-5. *Highlighting the selected component*

Adding the Shopping Cart

The shopping cart will allow the user to select several products in a single purchase before checking out. In the sections that follow, I add extend the data store to keep track of the user's product selections and create components that provide detailed and summary cart views.

Extending the Data Store

To extend the data store to add support for tracking the user's product selections, I added the action types shown in Listing 5-21.

Listing 5-21. Defining Action Types in the Types.js File in the src/data Folder

```
export const DataTypes = {
    PRODUCTS: "products",
    CATEGORIES: "categories"
}

export const ActionTypes = {
    DATA_LOAD: "data_load",
    CART_ADD: "cart_add",
    CART_UPDATE: "cart_update",
    CART_REMOVE: "cart_delete",
    CART_CLEAR: "cart_clear"
}
```

The new actions will allow products to be added and removed from the cart and for the entire cart content to be cleared.

You can define action creators and reducers for different parts of the application in the same file, but breaking them into separate files can make development easier, especially in large projects. I added a file called `CartActionCreators.js` to the `src/data` folder and used it to define action creators for the new action types, as shown in Listing 5-22.

Listing 5-22. The Contents of the CartActionCreators.js File in the src/data Folder

```
import { ActionTypes} from "./Types";

export const addToCart = (product, quantity) => ({
    type: ActionTypes.CART_ADD,
    payload: {
        product,
        quantity: quantity || 1
    }
});

export const updateCartQuantity = (product, quantity) => ({
    type: ActionTypes.CART_UPDATE,
    payload: { product, quantity }
})

export const removeFromCart = (product) => ({
    type: ActionTypes.CART_REMOVE,
    payload: product
})

export const clearCart = () => ({
    type: ActionTypes.CART_CLEAR
})
```

The action objects created by the functions in Listing 5-22 have a `payload` property that carries the data required to execute the action. To define a reducer that will process cart-related actions, I added a file called `CartReducer.js` in the `src/data` folder and defined the function shown in Listing 5-23.

Listing 5-23. The Contents of the CartReducer.js File in the src/data Folder

```
import { ActionTypes } from "./Types";

export const CartReducer = (storeData, action) => {
    let newStore = { cart: [], cartItems: 0, cartPrice: 0, ...storeData }
    switch(action.type) {
        case ActionTypes.CART_ADD:
            const p = action.payload.product;
            const q = action.payload.quantity;

            let existing = newStore.cart.find(item => item.product.id === p.id);
            if (existing) {
                existing.quantity += q;
            } else {
```

```
                newStore.cart = [...newStore.cart, action.payload];
            }
            newStore.cartItems += q;
            newStore.cartPrice += p.price * q;
            return newStore;

        case ActionTypes.CART_UPDATE:
            newStore.cart = newStore.cart.map(item => {
                if (item.product.id === action.payload.product.id) {
                    const diff = action.payload.quantity - item.quantity;
                    newStore.cartItems += diff;
                    newStore.cartPrice+= (item.product.price * diff);
                    return action.payload;
                } else {
                    return item;
                }
            });
            return newStore;

        case ActionTypes.CART_REMOVE:
            let selection = newStore.cart.find(item =>
                item.product.id === action.payload.id);
            newStore.cartItems -= selection.quantity;
            newStore.cartPrice -= selection.quantity * selection.product.price;
            newStore.cart = newStore.cart.filter(item => item !== selection );
            return newStore;

        case ActionTypes.CART_CLEAR:
            return { ...storeData, cart: [], cartItems: 0, cartPrice: 0}

        default:
            return storeData || {};
    }
}
```

The reducer for the cart actions keeps track of the user's product selection by adding a `cart` property to the data store and assigning it an array of objects that have `product` and `quantity` properties. There are also `cartItems` and `cartPrice` properties that keep track of the number of items in the cart and their total price.

■ **Tip** It is important to keep the structure of your data store flat because changes deep in an object hierarchy won't be detected and displayed to the user. It is for this reason that the `cart`, `cartItems`, and `cartPrice` properties are defined alongside the `products` and `categories` properties in the data store, rather than grouped together into a single structure.

By default, the Redux data store uses only one reducer, but it is easy to combine multiple reducers to suit your project. There is built-in support for dividing up responsibilities for the data store between multiple reducers, as described in Chapter 19, but this splits up the data so each reducer can see only part of the model. For the SportsStore application, I want each reducer to have access to the complete data store, so

I added a file called CommonReducer.js to the src/data folder and used it to define the function shown in Listing 5-24.

Listing 5-24. The Contents of the CommonReducer.js File in the src/data Folder

```
export const CommonReducer = (...reducers) => (storeData, action) => {
    for (let i = 0; i < reducers.length; i++ ) {
        let newStore = reducers[i](storeData, action);
        if (newStore !== storeData) {
            return newStore;
        }
    }
    return storeData;
}
```

The commonReducer function combines multiple reducers into a single function and asks each of them to handle actions. Reducers return new objects when they modify the contents of the data store, which makes it easy to detect when an action has been handled. The result is that the SportsStore data store can support multiple reducers where the first to change the data store is considered to have processed the action. In Listing 5-25, I have updated the data store configuration to use the commonReducer function to combine the shop and cart reducers.

Listing 5-25. Combining Reducers in the DataStore.js File in the src/data Folder

```
import { createStore } from "redux";
import { ShopReducer } from "./ShopReducer";
import { CartReducer } from "./CartReducer";
import { CommonReducer } from "./CommonReducer";

export const SportsStoreDataStore
    = createStore(CommonReducer(ShopReducer, CartReducer));
```

Creating the Cart Summary Component

To show the user a summary of their shopping cart, I added a file called CartSummary.js in the src/shop folder and used it to define the component shown in Listing 5-26.

Listing 5-26. The Contents of the CartSummary.js File in the src/shop Folder

```
import React, { Component } from "react";
import { Link } from "react-router-dom";

export class CartSummary extends Component {

    getSummary = () => {
        if (this.props.cartItems > 0) {
            return <span>
                { this.props.cartItems } item(s),
                ${ this.props.cartPrice.toFixed(2)}
            </span>
```

```
        } else {
            return <span>Your cart: (empty) </span>
        }
    }

    getLinkClasses = () => {
        return `btn btn-sm bg-dark text-white
            ${ this.props.cartItems === 0 ? "disabled": ""}`;
    }

    render() {
        return <div className="float-right">
            <small>
                 { this.getSummary() }
                <Link className={ this.getLinkClasses() }
                        to="/shop/cart">
                    <i className="fa fa-shopping-cart"></i>
                </Link>
            </small>
        </div>
    }
}
```

The component defined in Listing 5-26 receives the data it requires through `cartItems` and `cartPrice` props, which are used to create a summary of the component, along with a `Link` that will navigate to the `/shop/cart` URL when clicked. The `Link` is disabled when the value of the `items` prop is zero to prevent the user from progressing without selecting at least one product.

■ **Tip** The `i` element used as the content of the `Link` applies a cart icon from the Font Awesome package added to the project in Listing 5-2. See `https://fontawesome.com` for more details and the full range of icons available.

React handles many aspects of web application development well, but there are some common tasks that are harder to achieve than you might be used to. One example is conditional rendering, where a data value is used to select different content to present to the user or different values for props. The cleanest approach in React is to define a method that uses JavaScript to return a result expressed as HTML, like the `getSummary` and `getLinkClasses` methods in Listing 5-26, which are invoked in the component's `render` method. The other approach is to use the && operator inline, which works well for simple expressions.

In Listing 5-27, I connected the cart-related additions from the data store to the rest of the application, along with the action creator functions.

Listing 5-27. Connecting the Cart in the ShopConnector.js File in the src/shop Folder

```
import React, { Component } from "react";
import { Switch, Route, Redirect }
    from "react-router-dom"
import { connect } from "react-redux";
import { loadData } from "../data/ActionCreators";
import { DataTypes } from "../data/Types";
```

```
import { Shop } from "./Shop";
import { addToCart, updateCartQuantity, removeFromCart, clearCart }
    from "../data/CartActionCreators";

const mapStateToProps = (dataStore) => ({
    ...dataStore
})

const mapDispatchToProps = {
    loadData,addToCart, updateCartQuantity, removeFromCart, clearCart
}

const filterProducts = (products = [], category) =>
    (!category || category === "All")
        ? products
        : products.filter(p => p.category.toLowerCase() === category.toLowerCase());

export const ShopConnector = connect(mapStateToProps, mapDispatchToProps)(
    class extends Component {
        render() {
            return <Switch>
                <Route path="/shop/products/:category?"
                    render={ (routeProps) =>
                        <Shop { ...this.props } { ...routeProps }
                            products={ filterProducts(this.props.products,
                                routeProps.match.params.category) } />} />
                <Redirect to="/shop/products" />
            </Switch>
        }

        componentDidMount() {
            this.props.loadData(DataTypes.CATEGORIES);
            this.props.loadData(DataTypes.PRODUCTS);
        }
    }
)
```

In Listing 5-28, I added a CartSummary to the content rendered by the Shop component, which will ensure that details of the user's selections are shown above the list of products.

Listing 5-28. Adding the Summary in the Shop.js File in the src/shop Folder

```
import React, { Component } from "react";
import { CategoryNavigation } from "./CategoryNavigation";
import { ProductList } from "./ProductList";
import { CartSummary } from "./CartSummary";

export class Shop extends Component {

    render() {
        return <div className="container-fluid">
            <div className="row">
```

```
                <div className="col bg-dark text-white">
                    <div className="navbar-brand">SPORTS STORE</div>
                    <CartSummary { ...this.props } />
                </div>
            </div>
            <div className="row">
                <div className="col-3 p-2">
                    <CategoryNavigation baseUrl="/shop/products"
                        categories={ this.props.categories } />
                </div>
                <div className="col-9 p-2">
                    <ProductList products={ this.props.products }
                        addToCart={ this.props.addToCart } />
                </div>
            </div>
        </div>
    }
}
```

To allow the user to add a product to the cart, I added a button alongside the description of each product produced by the ProductList component, as shown in Listing 5-29.

Listing 5-29. Adding a Button in the ProductList.js File in the src/shop Folder

```
import React, { Component } from "react";

export class ProductList extends Component {

    render() {
        if (this.props.products == null || this.props.products.length === 0) {
            return <h5 className="p-2">No Products</h5>
        }
        return this.props.products.map(p =>
                <div className="card m-1 p-1 bg-light" key={ p.id }>
                    <h4>
                        { p.name }
                        <span className="badge badge-pill badge-primary float-right">
                            ${ p.price.toFixed(2) }
                        </span>
                    </h4>
                    <div className="card-text bg-white p-1">
                        { p.description }
                        <button className="btn btn-success btn-sm float-right"
                            onClick={ () => this.props.addToCart(p) } >
                                Add To Cart
                        </button>
                    </div>
                </div>
            )
    }
}
```

React provides props that are used to register handlers for events, as described in Chapter 12. The handler for the `click` event, which is triggered when an element is clicked, is `onClick`, and the function that is specified invokes the `addToCart` prop, which is mapped to the data store action creator of the same name.

The result is that each product is shown with an Add To Cart button. When the button is clicked, the data store is updated, and the summary of the user's selections reflects the additional item and the new total price, as shown in Figure 5-6.

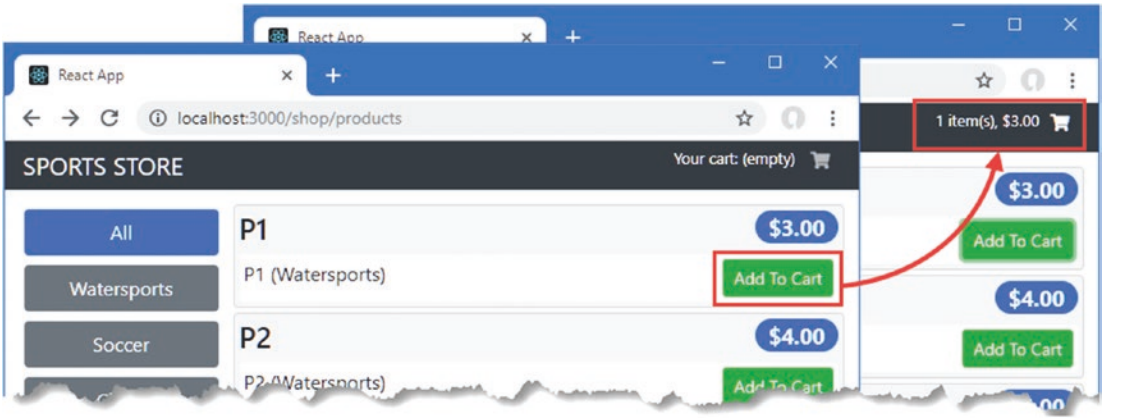

Figure 5-6. *Adding a product to the cart*

Adding the Cart Detail Component

To provide the user with a detailed view of their selections, I added a file called `CartDetails.js` to the `src/shop` folder and used it to define the component shown in Listing 5-30.

Listing 5-30. The Contents of the CartDetails.js File in the src/shop Folder

```
import React, { Component } from "react";
import { Link } from "react-router-dom";
import { CartDetailsRows } from "./CartDetailsRows";

export class CartDetails extends Component {

    getLinkClasses = () => `btn btn-secondary m-1
        ${this.props.cartItems === 0 ? "disabled": ""}`;

    render() {
        return <div className="m-3">
            <h2 className="text-center">Your Cart</h2>
            <table className="table table-bordered table-striped">
                <thead>
                    <tr>
                        <th>Quantity</th>
                        <th>Product</th>
                        <th className="text-right">Price</th>
```

```
                        <th className="text-right">Subtotal</th>
                        <th/>
                    </tr>
                </thead>
                <tbody>
                    <CartDetailsRows cart={ this.props.cart}
                        cartPrice={ this.props.cartPrice }
                        updateQuantity={ this.props.updateCartQuantity }
                        removeFromCart={ this.props.removeFromCart } />
                </tbody>
            </table>
            <div className="text-center">
                <Link className="btn btn-primary m-1" to="/shop">
                    Continue Shopping
                </Link>
                <Link className={ this.getLinkClasses() } to="/shop/checkout">
                    Checkout
                </Link>
            </div>
        </div>
    }
}
```

The `CartDetails` component presents a table to the user, along with `Link` components that return to the product list or navigate to the `/shop/checkout` URL, which starts the checkout process.

The `CartDetails` component relies on a `CartDetailsRows` component to display details of the user's product selection. To create this component, I added a file called `CartDetailsRows.js` to the `src/shop` folder and used it to define the component shown in Listing 5-31.

Listing 5-31. The Contents of the CartDetailsRows.js File in the src/shop Folder

```
import React, { Component } from "react";

export class CartDetailsRows extends Component {

    handleChange = (product, event) => {
        this.props.updateQuantity(product, event.target.value);
    }

    render() {
        if (!this.props.cart || this.props.cart.length === 0) {
            return <tr>
                <td colSpan="5">Your cart is empty</td>
            </tr>
        } else {
            return <React.Fragment>
                { this.props.cart.map(item =>
                    <tr key={ item.product.id }>
                        <td>
                            <input type="number" value={ item.quantity }
                                onChange={ (ev) =>
```

```
                                this.handleChange(item.product, ev) } />
                        </td>
                        <td>{ item.product.name }</td>
                        <td>${ item.product.price.toFixed(2) }</td>
                        <td>${ (item.quantity * item.product.price).toFixed(2) }</td>
                        <td>
                            <button className="btn btn-sm btn-danger"
                                onClick={ () =>
                                        this.props.removeFromCart(item.product)}>
                                    Remove
                                </button>
                        </td>
                    </tr>
                )}
                <tr>
                    <th colSpan="3" className="text-right">Total:</th>
                    <th colSpan="2">${ this.props.cartPrice.toFixed(2) }</th>
                </tr>
            </React.Fragment>
        }
    }
}
```

The render method must return a single top-level element, which is inserted into the HTML in place of the component's element when the HTML document is produced, as explained in Chapter 9. It isn't always possible to return a single HTML element without disrupting the content layout, such as in this example, where multiple table rows are required. For these situations, the `React.Fragment` element is used. This element is discarded when the content is processed and the elements it contains are added to the HTML document.

Adding the Cart URL to the Routing Configuration

In Listing 5-32, I have updated the routing configuration in the `ShopConnector` component to add support for the `/shop/cart` URL.

Listing 5-32. Adding a New URL in the ShopConnector.js File in the src/shop Folder

```
import React, { Component } from "react";
import { Switch, Route, Redirect }
    from "react-router-dom"
import { connect } from "react-redux";
import { loadData } from "../data/ActionCreators";
import { DataTypes } from "../data/Types";
import { Shop } from "./Shop";
import { addToCart, updateCartQuantity, removeFromCart, clearCart }
    from "../data/CartActionCreators";
import { CartDetails } from "./CartDetails";

const mapStateToProps = (dataStore) => ({
    ...dataStore
})
```

```
const mapDispatchToProps = {
    loadData,
    addToCart, updateCartQuantity, removeFromCart, clearCart
}

const filterProducts = (products = [], category) =>
    (!category || category === "All")
        ? products
        : products.filter(p => p.category.toLowerCase() === category.toLowerCase());

export const ShopConnector = connect(mapStateToProps, mapDispatchToProps)(
    class extends Component {
        render() {
            return <Switch>
                <Route path="/shop/products/:category?"
                    render={ (routeProps) =>
                        <Shop { ...this.props } { ...routeProps }
                            products={ filterProducts(this.props.products,
                                routeProps.match.params.category) } />} />
                <Route path="/shop/cart" render={ (routeProps) =>
                        <CartDetails { ...this.props } { ...routeProps }  />} />
                <Redirect to="/shop/products" />
            </Switch>
        }

        componentDidMount() {
            this.props.loadData(DataTypes.CATEGORIES);
            this.props.loadData(DataTypes.PRODUCTS);
        }
    }
)
```

The new Route handles the /shop/cart URL by displaying the CartDetails component, which receives props from both the data store and the routing system. In Listing 5-33, I have updated the Shop component to define a wrapper function around the addToCart action creator that also navigates to the new URL.

Listing 5-33. Navigating to the Cart in the Shop.js File in the src/shop Folder

```
import React, { Component } from "react";
import { CategoryNavigation } from "./CategoryNavigation";
import { ProductList } from "./ProductList";
import { CartSummary } from "./CartSummary";

export class Shop extends Component {

    handleAddToCart = (...args) => {
        this.props.addToCart(...args);
        this.props.history.push("/shop/cart");
    }
```

```
    render() {
        return <div className="container-fluid">
            <div className="row">
                <div className="col bg-dark text-white">
                    <div className="navbar-brand">SPORTS STORE</div>
                    <CartSummary { ...this.props } />
                </div>
            </div>
            <div className="row">
                <div className="col-3 p-2">
                    <CategoryNavigation baseUrl="/shop/products"
                        categories={ this.props.categories } />
                </div>
                <div className="col-9 p-2">
                    <ProductList products={ this.props.products }
                        addToCart={ this.handleAddToCart } />
                </div>
            </div>
        </div>
    }
}
```

The result is that clicking the Add To Cart button for a product displays the updated cart, which provides the user with the choice to return to the product list and make further selections, edit the contents of the cart, or start the checkout process, as shown in Figure 5-7.

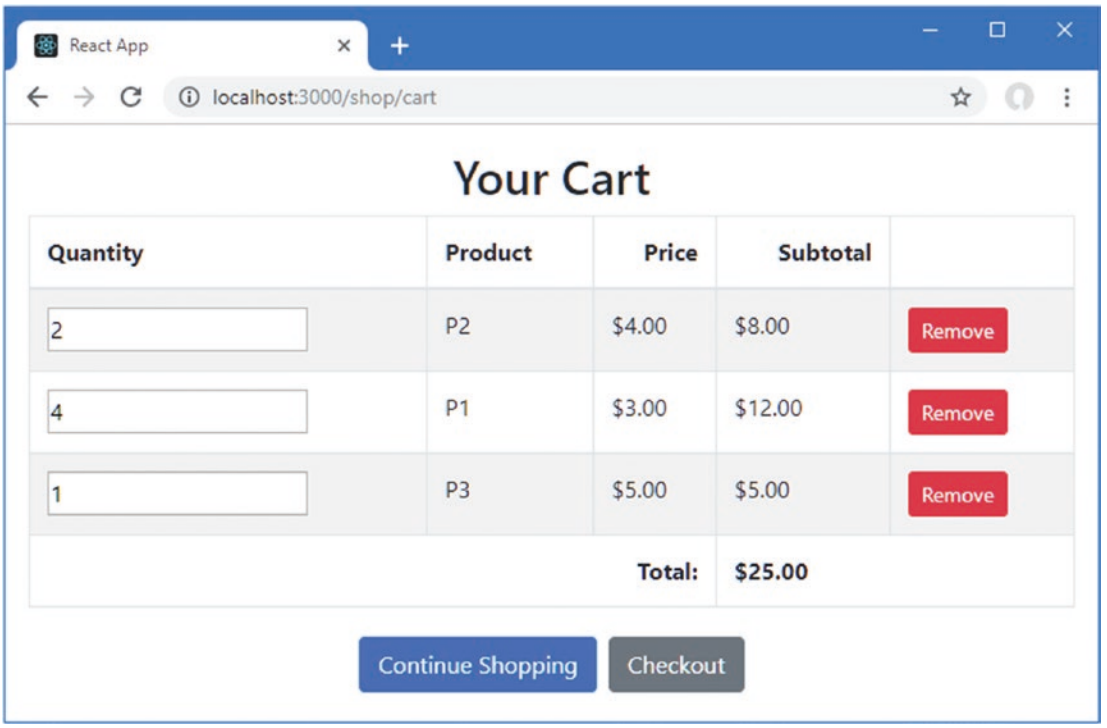

Figure 5-7. *Integrating the cart into the SportsStore project*

The Checkout button returns the user to the `/store/products` URL at the moment, but I add support for checking out in Chapter 6.

Summary

In this chapter, I started development of a realistic React project. The first part of the chapter was spent setting up the Redux data store, which introduces a range of terms—actions, action creators, reducers—that you may not be familiar with but which will soon become second nature. I also set up the React Router package so that the browser's URL can be used to select the content and data that is presented to the user. The foundation these features provides takes time to set up, but you will see that it starts to pay dividends as I add further features to SportsStore. In the next chapter, I add further features to the SportsStore application.