CHAPTER 8

SportsStore: Authentication and Deployment

In this chapter, I add authentication to the SportsStore application to protect the administration features from unauthorized use. I also prepare the SportsStore application for deployment into a Docker container, which can be used on most hosting platforms.

Preparing for This Chapter

To prepare for this chapter, I am going to add support for authentication and authorization to the simple server that provides the RESTful web service and GraphQL service. At the moment, any client can perform any operation, which means that shoppers could change prices, create products, and perform other tasks that should be restricted to administrators. Table 8-1 lists the combination of HTTP methods and URLs that should be publicly accessible; everything else will be protected, including all GraphQL queries and mutations.

Table 8-1. *The Publicly Accessible HTTP Methods and URL Combinations*

HTTP Method	URL	Description
`GET`	`/api/products`	This combination is used to request pages of products for shoppers.
`GET`	`/api/categories`	This combination is used to request the set of categories and is used to provide shoppers with navigation buttons.
`POST`	`/api/orders`	This combination is used to submit orders.
`POST`	`/login`	This combination will be used to submit a username and password for authentication.

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`.

© Adam Freeman 2019
A. Freeman, *Pro React 16*, https://doi.org/10.1007/978-1-4842-4451-7_8

To implement the authentication and provide the means for authorization, I added a file called `authMiddleware.js` to the `sportsstore` folder and added the code shown in Listing 8-1.

Listing 8-1. The Contents of the authMiddleware.js File in the sportsstore Folder

```
const jwt = require("jsonwebtoken");

const APP_SECRET = "myappsecret", USERNAME = "admin", PASSWORD = "secret";

const anonOps = [{ method: "GET", urls: ["/api/products", "/api/categories"]},
                 { method: "POST", urls: ["/api/orders"]}]

module.exports = function (req, res, next) {
    if (anonOps.find(op => op.method === req.method
            && op.urls.find(url => req.url.startsWith(url)))) {
        next();
    } else if (req.url === "/login" && req.method === "POST") {
        if (req.body.username === USERNAME && req.body.password === PASSWORD) {
            res.json({
                success: true,
                token: jwt.sign({ data: USERNAME, expiresIn: "1h" }, APP_SECRET)
            });
        } else {
            res.json({ success: false });
        }
        res.end();
    } else {
        let token = req.headers["authorization"];
        if (token != null && token.startsWith("Bearer<")) {
            token = token.substring(7, token.length - 1);
            jwt.verify(token, APP_SECRET);
            next();
        } else {
            res.statusCode = 401;
            res.end();
        }
    }
}
```

The code in Listing 8-1 will inspect each request received by the HTTP server that delivers the RESTful web service and the GraphQL service. A 401 unauthorized response is returned if a request isn't for one of the unsecured combinations of HTTP method and URL. The `/login` URL is used for authentication, with the hardwired credentials shown in Table 8-2.

Table 8-2. *The Credentials Used by the SportsStore Application*

Name	Description
`name`	`admin`
`password`	`secret`

■ **Caution** All of the server-side code in the SportsStore project can be used for real projects except Listing 8-1, which contains hard-coded credentials and is unsuitable for anything other than basic development and testing.

To add the middleware to the server, I added the statements shown in Listing 8-2 to the `server.js` file.

Listing 8-2. Adding Middleware in the server.js File in the sportsstore Folder

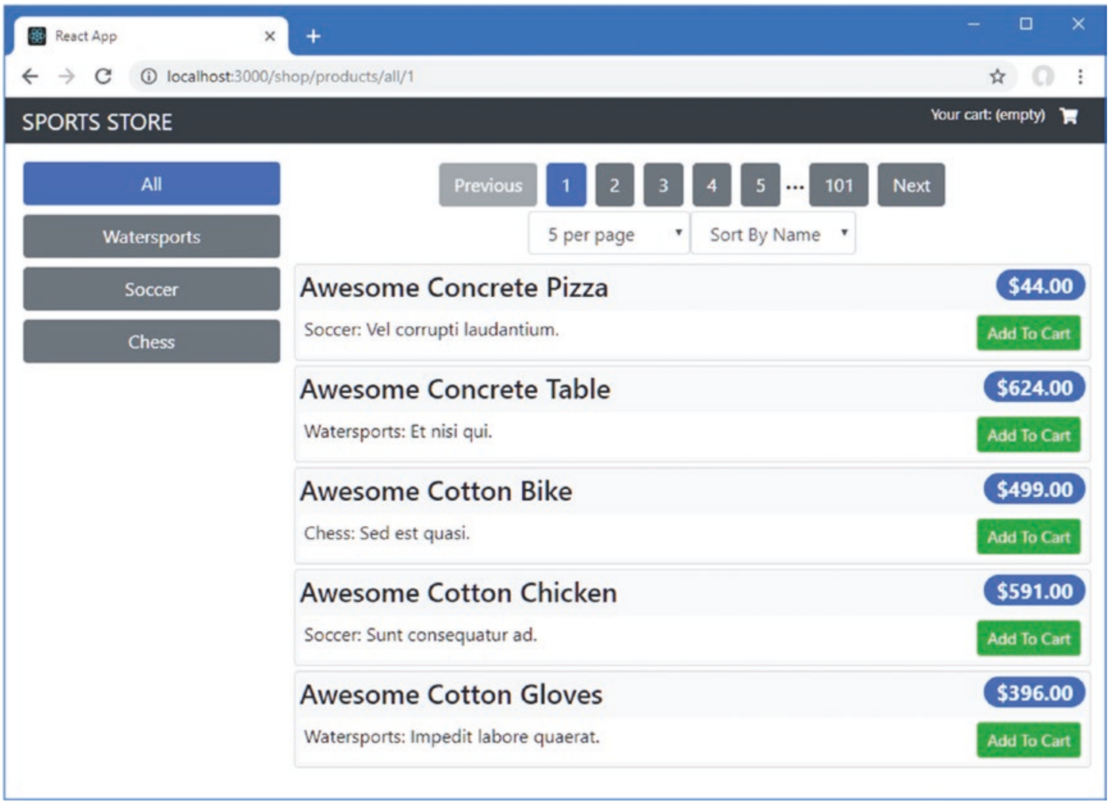
```
const express = require("express");
const jsonServer = require("json-server");
const chokidar = require('chokidar');
const cors = require("cors");
const fs = require("fs");
const { buildSchema } = require("graphql");
const graphqlHTTP = require("express-graphql");
const queryResolvers  = require("./serverQueriesResolver");
const mutationResolvers = require("./serverMutationsResolver");
const auth = require("./authMiddleware");

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

let router = undefined;
let graph = undefined;

const app = express();

const createServer = () => {
    delete require.cache[require.resolve(fileName)];
    setTimeout(() => {
        router = jsonServer.router(fileName.endsWith(".js")
                ? require(fileName)() : fileName);
        let schema =  fs.readFileSync("./serverQueriesSchema.graphql", "utf-8")
            + fs.readFileSync("./serverMutationsSchema.graphql", "utf-8");
        let resolvers = { ...queryResolvers, ...mutationResolvers };
        graph = graphqlHTTP({
            schema: buildSchema(schema), rootValue: resolvers,
            graphiql: true, context: { db: router.db }
        })
    }, 100)
}

createServer();

app.use(cors());
app.use(jsonServer.bodyParser)
app.use(auth);
app.use("/api", (req, resp, next) => router(req, resp, next));
app.use("/graphql", (req, resp, next) => graph(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}`));
```

Open a new command prompt, navigate to the `sportsstore` folder, and run the command shown in Listing 8-3 to start the React development tools, the RESTful web service, and the GraphQL service.

Listing 8-3. Starting the Development Tool and Web Services

```
npm start
```

Once the project has been compiled, a new browser window will open and show the SportsStore shopping features, as shown in Figure 8-1.



Awesome Concrete Pizza $44.00
Soccer: Vel corrupti laudantium.

Awesome Concrete Table $624.00
Watersports: Et nisi qui.

Awesome Cotton Bike $499.00
Chess: Sed est quasi.

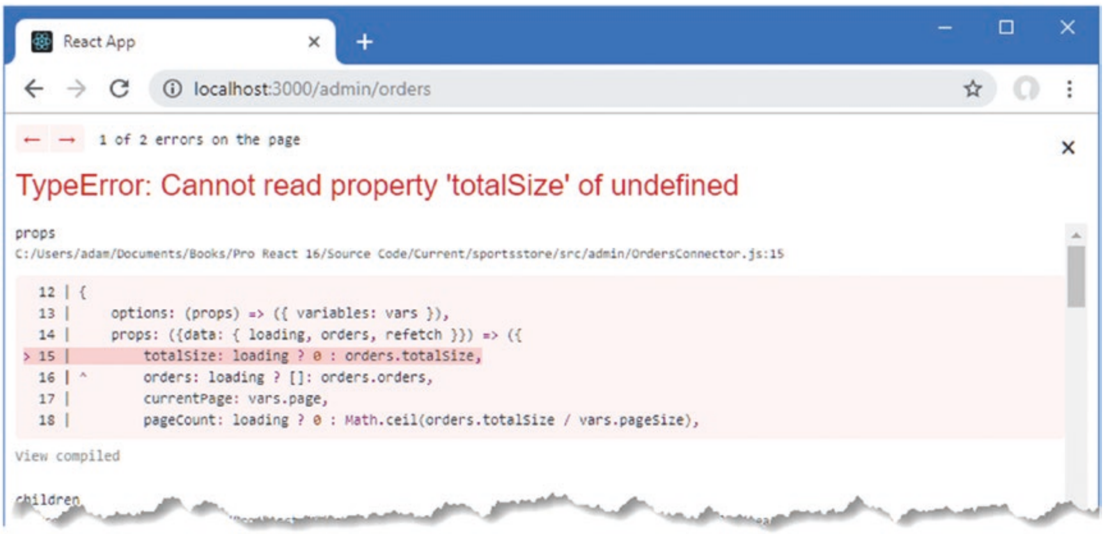
Awesome Cotton Chicken $591.00
Soccer: Sunt consequatur ad.

Awesome Cotton Gloves $396.00
Watersports: Impedit labore quaerat.

Figure 8-1. *Running the example application*

Adding Authentication for GraphQL Requests

The introduction of the authentication middleware has broken the administration features, which rely on HTTP requests that are no longer publicly accessible. If you navigate to `http://localhost:3000/admin`, you will see the effect of the 401 – Not Authorized response that the server makes to the GraphQL HTTP requests, as shown in Figure 8-2.



```
← →  1 of 2 errors on the page

TypeError: Cannot read property 'totalSize' of undefined

props
C:/Users/adam/Documents/Books/Pro React 16/Source Code/Current/sportsstore/src/admin/OrdersConnector.js:15

  12 | {
  13 |     options: (props) => ({ variables: vars }),
  14 |     props: ({data: { loading, orders, refetch }}) => ({
> 15 |         totalSize: loading ? 0 : orders.totalSize,
  16 | ^       orders: loading ? []: orders.orders,
  17 |         currentPage: vars.page,
  18 |         pageCount: loading ? 0 : Math.ceil(orders.totalSize / vars.pageSize),

View compiled

children
```

Figure 8-2. *Encountering an error*

In the sections that follow, I explain how the SportsStore application will authenticate users and implement the required features to prevent the error shown in the figure and restore the administration features for authenticated users.

Understanding the Authentication System

When the server authenticates a user, it will return a JSON Web Token (JWT) that the application must include in subsequent HTTP requests to show that authentication has been successfully performed. You can read the JWT specification at `https://tools.ietf.org/html/rfc7519`, but for the purposes of the SportsStore project, it is enough to know that the application can authenticate the user by sending a POST request to the `/login` URL, including a JSON-formatted object in the request body that contains name and password properties. There is only one set of valid credentials in the authentication code defined in Listing 8-1, which I have repeated in Table 8-3. You should not hard-code credentials in real projects, but this is the username and password that you will need for the SportsStore application.

Table 8-3. *The Authentication Credentials Supported by the RESTful Web Service*

Username	Password
`admin`	`secret`

If the correct credentials are sent to the /login URL, then the response from the server will contain a JSON object like this:

```
{
  "success": true,
  "token":"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJkYXRhIjoiYWRtaW4iLCJleHBpcmVz
           SW4iOiIxaCIsImlhdCI6MTQ3ODk1NjI1Mn0.lJaDDrSu-bHBtdWrz0312p_DG5tKypGv6cA
           NgOyzlg8"
}
```

The success property describes the outcome of the authentication operation, and the token property contains the JWT, which should be included in subsequent requests using the Authorization HTTP header in this format:

```
Authorization: Bearer<eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJkYXRhIjoiYWRtaW4iLC
                     JleHBpcmVzSW4iOiIxaCIsImlhdCI6MTQ3ODk1NjI1Mn0.lJaDDrSu-bHBtd
                     Wrz0312p_DG5tKypGv6cANgOyzlg8>
```

I configured the JWT tokens returned by the server so they expire after one hour.

If the wrong credentials are sent to the server, then the JSON object returned in the response will contain just a success property set to false, like this:

```
{
  "success": false
}
```

Creating the Authentication Context

The SportsStore application needs to be able to determine whether the user has been authenticated and keep track of the web token that must be included in HTTP requests, ensuring that the administration features are shown only after successful authentication.

This is the type of information that is often required in multiple places in an application, to ensure that components can easily collaborate. For the SportsStore application, I am going to use the React *context* feature, which allows functionality to be easily shared between components in a simple and lightweight way and which is described in Chapter 14. I created the src/auth folder and added to it a file called AuthContext.js with the code shown in Listing 8-4.

Listing 8-4. The Contents of the AuthContext.js File in the src/auth Folder

```
import React from "react";

export const AuthContext = React.createContext({
    isAuthenticated: false,
    webToken: null,
    authenticate: (username, password) => {},
    signout: () => {}
})
```

The React.createContext method is used to create a context, and the object it receives is used for default values, which is why the authenticate and signout functions are empty. The real functionality for a context is provided by a provider component, which I defined by creating a file called `AuthProviderImpl.js` in the `src/auth` folder and adding the code shown in Listing 8-5.

Listing 8-5. The Contents of the AuthProviderImpl.js File in the src/auth Folder

```
import React, { Component } from "react";
import Axios from "axios";
import { AuthContext } from "./AuthContext";
import { authUrl } from "../data/Urls";

export class AuthProviderImpl extends Component {

    constructor(props) {
        super(props);
        this.state = {
            isAuthenticated: false,
            webToken: null
        }
    }

    authenticate = (credentials) => {
        return Axios.post(authUrl,  credentials).then(response => {
            if (response.data.success === true) {
                this.setState({
                    isAuthenticated: true,
                    webToken:response.data.token
                })
                return true;
            } else {
                throw new Error("Invalid Credentials");
            }
        })
    }

    signout = () => {
        this.setState({ isAuthenticated: false, webToken: null });
    }

    render = () =>
        <AuthContext.Provider value={ {...this.state,
                authenticate: this.authenticate, signout: this.signout}}>
            { this.props.children }
        </AuthContext.Provider>
}
```

This component uses the React context feature in its render method to provide an implementation of the `AuthContext` properties and functions, which it does through the `value` prop of the special `AuthContext.Provider` element. The effect is to share access to the state data and the `authenticate` and `signout` methods directly to any descendant component that applies the corresponding `AuthContext.Consumer` element, which I will use shortly.

The implementation of the `authenticate` method uses the Axios package to send a POST request to validate credentials that will be obtained from the user. The result of the authenticate method is a

Promise that will be resolved when the server responds to confirm the credentials and will be rejected if the credentials are incorrect.

To define the URL used to perform authentication, I added the URL shown in Listing 8-6.

Listing 8-6. Adding a URL in the Urls.js File in the src/data Folder

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

const protocol = "http";
const hostname = "localhost";
const port = 3500;

export const RestUrls = {
    [DataTypes.PRODUCTS]: `${protocol}://${hostname}:${port}/api/products`,
    [DataTypes.CATEGORIES]: `${protocol}://${hostname}:${port}/api/categories`,
    [DataTypes.ORDERS]: `${protocol}://${hostname}:${port}/api/orders`
}

export const GraphQlUrl = `${protocol}://${hostname}:${port}/graphql`;

export const authUrl = `${protocol}://${hostname}:${port}/login`;
```

To apply the context to the SportsStore application, I made the changes shown in Listing 8-7 to the `App.js` file.

Listing 8-7. Adding a Context Provider 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";
import { Admin } from "./admin/Admin";
import { AuthProviderImpl } from "./auth/AuthProviderImpl";

export default class App extends Component {

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

To make it easier to consume the features defined by the `AuthContext`, I added a file called `AuthWrapper.js` to the `src/auth` folder and defined the higher-order component shown in Listing 8-8.

Listing 8-8. The Contents of the AuthWrapper.js File in the src/auth Folder

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

export const authWrapper = (WrappedComponent) =>
    class extends Component {
        render = () =>
            <AuthContext.Consumer>
                { context =>
                    <WrappedComponent { ...this.props } { ...context } />
                }
            </AuthContext.Consumer>
    }
```

The context features rely on a render prop function, which can be difficult to integrate directly into components. Using the `authWrapper` function will allow a component to receive the features defined by the `AuthContext` as props. (Higher-order components and render prop functions are both described in Chapter 14.)

Creating the Authentication Form

To allow the user to provide their credentials, I added a file called `AuthPrompt.js` to the `src/auth` folder and used it to define the component shown in Listing 8-9.

Listing 8-9. The Contents of the AuthPrompt.js File in the src/auth Folder

```
import React, { Component } from "react";
import { withRouter } from "react-router-dom";
import { authWrapper } from "./AuthWrapper";
import { ValidatedForm } from "../forms/ValidatedForm";

export const AuthPrompt = withRouter(authWrapper(class extends Component {

    constructor(props) {
        super(props);
        this.state = {
            errorMessage: null
        }
        this.defaultAttrs = { required: true };
        this.formModel = [
            { label: "Username", attrs: { defaultValue: "admin"}},
            { label: "Password", attrs: { type: "password"} },
        ];
    }

    authenticate = (credentials) => {
        this.props.authenticate(credentials)
            .catch(err => this.setState({ errorMessage: err.message}))
            .then(this.props.history.push("/admin"));
    }
    render = () =>
        <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 m-2">
                    { this.state.errorMessage != null &&
                        <h4 className="bg-danger text-center text-white m-1 p-2">
                            { this.state.errorMessage }
                        </h4>
                    }
                    <ValidatedForm formModel={ this.formModel }
                        defaultAttrs={ this.defaultAttrs }
                        submitCallback={ this.authenticate }
                        submitText="Login"
                        cancelCallback={ () => this.props.history.push("/")}
                        cancelText="Cancel"
                    />
                </div>
            </div>
        </div>
}))
```

This component receives routing features from the `withRouter` function and authentication features from the `authWrapper` function, both of which will be presented through the component's props. The `ValidatedForm` I defined in Chapter 6 is used to present the user with username and password fields, both of which require values. When the form data is submitted, the `authenticate` method forwards the details for authentication. If authentication is successful, then the `history` object provided by the URL routing system (described in Chapters 21 and 22) is used to redirect the user to the `/admin` URL. An error message is displayed if authentication fails.

Guarding the Authentication Features

To prevent access to the administration features until the user has been authenticated, I made the changes shown in Listing 8-10 to the `Admin` component.

Listing 8-10. Guarding Features in the Admin.js File in the src/admin Folder

```
import React, { Component } from "react";
import  ApolloClient from "apollo-boost";
import { ApolloProvider} from "react-apollo";
import { GraphQlUrl } from "../data/Urls";
import { OrdersConnector } from "./OrdersConnector"
import { Route, Redirect, Switch } from "react-router-dom";
import { ToggleLink } from "../ToggleLink";
import { ConnectedProducts } from "./ProductsConnector";
import { ProductEditor } from "./ProductEditor";
import { ProductCreator } from "./ProductCreator";
import { AuthPrompt } from "../auth/AuthPrompt";
import { authWrapper } from "../auth/AuthWrapper";

// const graphQlClient = new ApolloClient({
//     uri: GraphQlUrl
// });

export const Admin = authWrapper(class extends Component {

    constructor(props) {
        super(props);
        this.client = new ApolloClient({
            uri: GraphQlUrl,
            request: gqloperation => gqloperation.setContext({
                headers: {
                    Authorization: `Bearer<${this.props.webToken}>`
                },
            })
        })
    }

    render() {
        return <ApolloProvider client={ this.client }>
            <div className="container-fluid">
                <div className="row">
                <div className="col bg-info text-white">
                    <div className="navbar-brand">SPORTS STORE</div>
                </div>
            </div>
            <div className="row">
                <div className="col-3 p-2">
                    <ToggleLink to="/admin/orders">Orders</ToggleLink>
                    <ToggleLink to="/admin/products">Products</ToggleLink>
                    { this.props.isAuthenticated &&
                        <button onClick={ this.props.signout }
                            className=
                                "btn btn-block btn-secondary m-2 fixed-bottom col-3">
                            Log Out
                        </button>
                    }
                </div>
                <div className="col-9 p-2">
                    <Switch>
                        {
                            !this.props.isAuthenticated &&
                                <Route component={ AuthPrompt } />
                        }
                        <Route path="/admin/orders" component={ OrdersConnector } />
                        <Route path="/admin/products/create"
                            component={ ProductCreator} />
                        <Route path="/admin/products/:id"
```

```
                            component={ ProductEditor} />
                        <Route path="/admin/products"
                            component={ ConnectedProducts } />
                        <Redirect to="/admin/orders" />
                    </Switch>
                </div>
            </div>
        </div>
        </ApolloProvider>
    }
})
```

The Admin component is wrapped with the authWrapper function so it has access to the authentication features. The ApolloClient object is created in the constructor so that I can add a function that modifies each request to add an Authorization header to each GraphQL HTTP request.

There are two new code fragments in the render method. The first displays a logout button if the user is authenticated. The second fragment checks the authentication status and produces a Route component that displays the AuthPrompt component, regardless of the URL. (A Route component without a path property will always display its component and can be used with a Switch to prevent other Route components from being evaluated.)

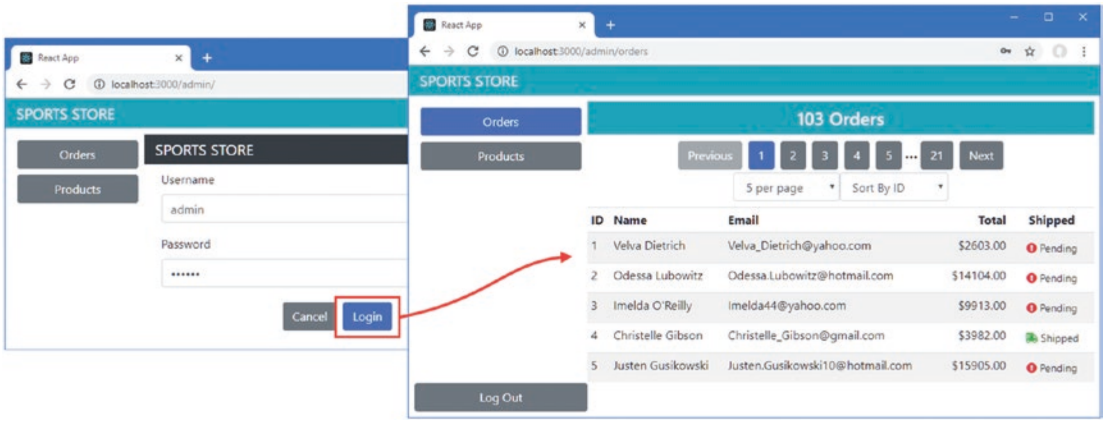
Adding a Navigation Link for the Administration Features

To make it easier to use the administration features, I added a Link to the CategoryNavigation component, as shown in Listing 8-11.

Listing 8-11. Adding a Link in the CategoryNavigation.js File in the src/shop Folder

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

export class CategoryNavigation extends Component {

    render() {
        return <React.Fragment>
            <ToggleLink to={ `${this.props.baseUrl}/all` } exact={ false }>
                All
            </ToggleLink>
            { this.props.categories && this.props.categories.map(cat =>
                <ToggleLink key={ cat }
                    to={ `${this.props.baseUrl}/${cat.toLowerCase()}`}>
                    { cat }
                </ToggleLink>
            )}
            <Link className="btn btn-block btn-secondary fixed-bottom m-2 col-3"
                to="/admin">
                Administration
            </Link>
        </React.Fragment>
    }
}
```

To see the authentication feature, navigate to `http://localhost:3000` and click the new Administration button. The guard will ensure that the authentication form is displayed. Enter **secret** into the password field and click the Login button to perform authentication, which will then display the administration features, as shown in Figure 8-3. Click the Log Out button to return to the unauthenticated state.



Username: admin
Password: ••••••

ID	Name	Email	Total	Shipped
1	Velva Dietrich	Velva_Dietrich@yahoo.com	$2603.00	Pending
2	Odessa Lubowitz	Odessa.Lubowitz@hotmail.com	$14104.00	Pending
3	Imelda O'Reilly	Imelda44@yahoo.com	$9913.00	Pending
4	Christelle Gibson	Christelle_Gibson@gmail.com	$3982.00	Shipped
5	Justen Gusikowski	Justen.Gusikowski10@hotmail.com	$15905.00	Pending

Figure 8-3. *Authenticating to use the administration features*

Preparing the Application for Deployment

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

Enabling Lazy Loading for the Administration Features

When the application is deployed, the individual JavaScript files will be combined into a single file that the browser can download more efficiently. Most of the users will be shoppers, which means they are unlikely to require the administration features. To prevent them from downloading code that is unlikely to be used, I have enabled lazy loading on the `import` statement that incorporates the top-level administration component into the rest of the application, as shown in Listing 8-12.

Listing 8-12. Using Lazy Loading in the App.js File in the src Folder

```
import React, { Component, lazy, Suspense } 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";
//import { Admin } from "./admin/Admin";
import { AuthProviderImpl } from "./auth/AuthProviderImpl";

const Admin = lazy(() => import("./admin/Admin"));

export default class App extends Component {
    render() {
        return <Provider store={ SportsStoreDataStore }>
            <AuthProviderImpl>
                <Router>
                    <Switch>
                        <Route path="/shop" component={ ShopConnector } />
                        <Route path="/admin" render={
                            routeProps =>
                                <Suspense fallback={ <h3>Loading...</h3> }>
                                    <Admin { ...routeProps } />
                                </Suspense>
                            } />
                        <Redirect to="/shop" />
                    </Switch>
                </Router>
            </AuthProviderImpl>
        </Provider>
    }
}
```

The Suspense component is used to denote content that should be loaded only when it is required and is combined with the lazy function. Together, these ensure that the Admin component will not be loaded until it is required. The lazy loading feature is a recent addition to React and, at the time of writing, doesn't support lazily loading named exports from files. To accommodate this requirement, I changed the definition of the Admin component as shown in Listing 8-13.

Listing 8-13. Changing the Export in the Admin.js File in the src/admin Folder

```
import React, { Component } from "react";
import  ApolloClient from "apollo-boost";
import { ApolloProvider} from "react-apollo";
import { GraphQlUrl } from "../data/Urls";
import { OrdersConnector } from "./OrdersConnector"
import { Route, Redirect, Switch } from "react-router-dom";
import { ToggleLink } from "../ToggleLink";
import { ConnectedProducts } from "./ProductsConnector";
import { ProductEditor } from "./ProductEditor";
import { ProductCreator } from "./ProductCreator";
import { AuthPrompt } from "../auth/AuthPrompt";
import { authWrapper } from "../auth/AuthWrapper";

export default authWrapper(class extends Component {

    // ...constructor and render method omitted for brevity...

})
```

Creating the Data File

The data file that is used by the RESTful and GraphQL services uses JavaScript to generate the same data each time the server is started. This has been useful during development because it has made it easy to return to a known state, but it isn't suitable for a production application.

The `json-server` package will create a persistent database when it is provided with a JSON file, so I added a file called `productionData.json` to the `sportstore` folder with the content shown in Listing 8-14.

Listing 8-14. The Contents of the productionData.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 }
    ],
    "categories": ["Watersports", "Soccer", "Chess"],
    "orders": []
}
```

Configuring the Request URLs

When I deploy the application, I will replace the React development HTTP server with one that combines serving static HTML and JavaScript files, as well as providing the RESTful and GraphQL services. To prepare for combining all the services on a single port, I changed the format of the URLs that the SportsStore uses, as shown in Listing 8-15.

Listing 8-15. Changing URLs in the Urls.js File in the src/data Folder

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

// const protocol = "http";
// const hostname = "localhost";
// const port = 3500;
```

```
export const RestUrls = {
    [DataTypes.PRODUCTS]: `/api/products`,
    [DataTypes.CATEGORIES]: `/api/categories`,
    [DataTypes.ORDERS]: `/api/orders`
}

export const GraphQlUrl = `/graphql`;
export const authUrl = `/login`;
```

Building the Application

To create the optimized version of the application suitable for production use, open a new command prompt, navigate to the `sportsstore` folder, and run the command shown in Listing 8-16.

Listing 8-16. Building the Application for Deployment

```
npm run build
```

The build process can take a moment to complete, and the result is an optimized set of files in the `build` folder.

Creating the Application Server

The React development HTTP server isn't suitable for production. In Listing 8-17, I have extended the server that has been providing the RESTful and GraphQL services so that it will also serve the files from the `build` folder.

Listing 8-17. Configuring the Server in 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 fs = require("fs");
const { buildSchema } = require("graphql");
const graphqlHTTP = require("express-graphql");
const queryResolvers  = require("./serverQueriesResolver");
const mutationResolvers = require("./serverMutationsResolver");
const auth = require("./authMiddleware");
const history = require("connect-history-api-fallback");

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

let router = undefined;
let graph = undefined;

const app = express();
const createServer = () => {
    delete require.cache[require.resolve(fileName)];
    setTimeout(() => {
        router = jsonServer.router(fileName.endsWith(".js")
                ? require(fileName)() : fileName);
        let schema =  fs.readFileSync("./serverQueriesSchema.graphql", "utf-8")
            + fs.readFileSync("./serverMutationsSchema.graphql", "utf-8");
        let resolvers = { ...queryResolvers, ...mutationResolvers };
        graph = graphqlHTTP({
            schema: buildSchema(schema), rootValue: resolvers,
            graphiql: true, context: { db: router.db }
        })
    }, 100)
}

createServer();

app.use(history());
app.use("/", express.static("./build"));
app.use(cors());
app.use(jsonServer.bodyParser)
app.use(auth);
app.use("/api", (req, resp, next) => router(req, resp, next));
app.use("/graphql", (req, resp, next) => graph(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}`));
```

The `connect-history-api-fallback` package responds to any HTTP request with the contents of the `index.html` file. This is useful for applications that use URL routing because it means that users can navigate directly to the URLs to which the application navigates using the HTML5 History API.

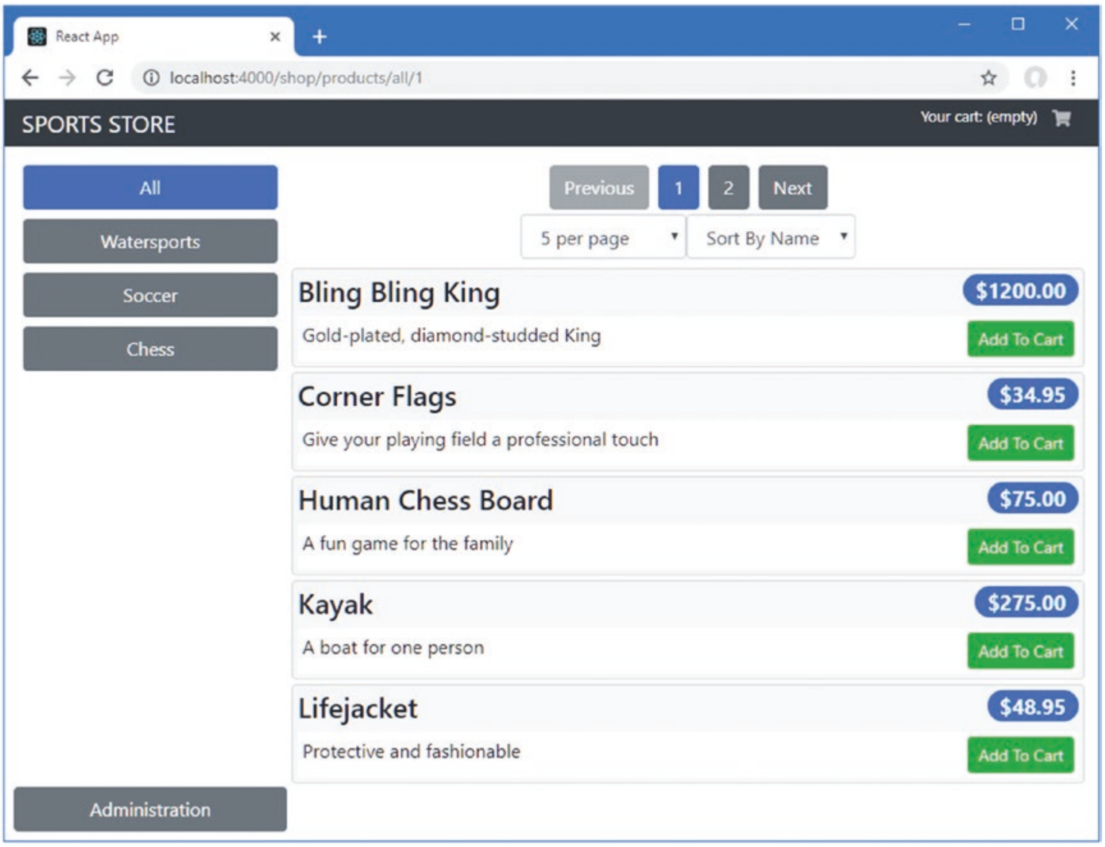
Testing the Production Build and Server

To ensure that the production build is working and that the server has been configured correctly, run the command shown in Listing 8-18 in the `sportsstore` folder.

Listing 8-18. Testing the Production Build

```
node server.js ./productionData.json 4000
```

Once the server has started, open a new browser window and navigate to `http://localhost:4000`; you will see the familiar content shown in Figure 8-4.



Bling Bling King $1200.00
Gold-plated, diamond-studded King

Corner Flags $34.95
Give your playing field a professional touch

Human Chess Board $75.00
A fun game for the family

Kayak $275.00
A boat for one person

Lifejacket $48.95
Protective and fashionable

Figure 8-4. *Testing 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-react-16`) 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 8-19.

Listing 8-19. The Contents of the deploy-package.json File in the sportsstore Folder

```
{
    "name": "sportsstore",
    "description": "SportsStore",
    "repository": "https://github.com/Apress/pro-react-16",
    "license": "0BSD",

    "devDependencies": {
      "graphql": "^14.0.2",
      "chokidar": "^2.0.4",
      "connect-history-api-fallback": "^1.5.0",
      "cors": "^2.8.5",
      "express": "^4.16.4",
      "express-graphql": "^0.7.1",
      "json-server": "^0.14.2",
      "jsonwebtoken": "^8.1.1"
    }
}
```

The `devDependencies` section species the packages required to run the application in the container. All of the packages that are used in the browser have been included in the JavaScript files produced by the `build` command. The other fields describe the application, and their main use is to prevent warning when the container is created.

Creating the Docker Container

To define the container, I added a file called `Dockerfile` (with no extension) to the `sportsstore` folder and added the content shown in Listing 8-20.

Listing 8-20. The Contents of the Dockerfile File in the sportsstore Folder

```
FROM node:10.14.1

RUN mkdir -p /usr/src/sportsstore

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

COPY serverQueriesSchema.graphql /usr/src/sportsstore/
COPY serverQueriesResolver.js /usr/src/sportsstore/
COPY serverMutationsSchema.graphql /usr/src/sportsstore/
COPY serverMutationsResolver.js /usr/src/sportsstore/

WORKDIR /usr/src/sportsstore

RUN echo 'package-lock=false' >> .npmrc

RUN npm install

EXPOSE 80

CMD ["node", "server.js", "./productionData.json", "80"]
```

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 file that will be used to install the NPM packages required to run the application in deployment.

To speed up the containerization process, I created a file called `.dockerignore` in the `sportsstore` folder with the content shown in Listing 8-21. This tells Docker to ignore the `node_modules` folder, which is not required in the container and takes a long time to process.

Listing 8-21. The Contents of the .dockerignore File in the sportsstore Folder

```
node_modules
```

Run the command shown in Listing 8-22 in the `sportsstore` folder to create an image that will contain the SportsStore application, along with all the packages it requires.

Listing 8-22. 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.

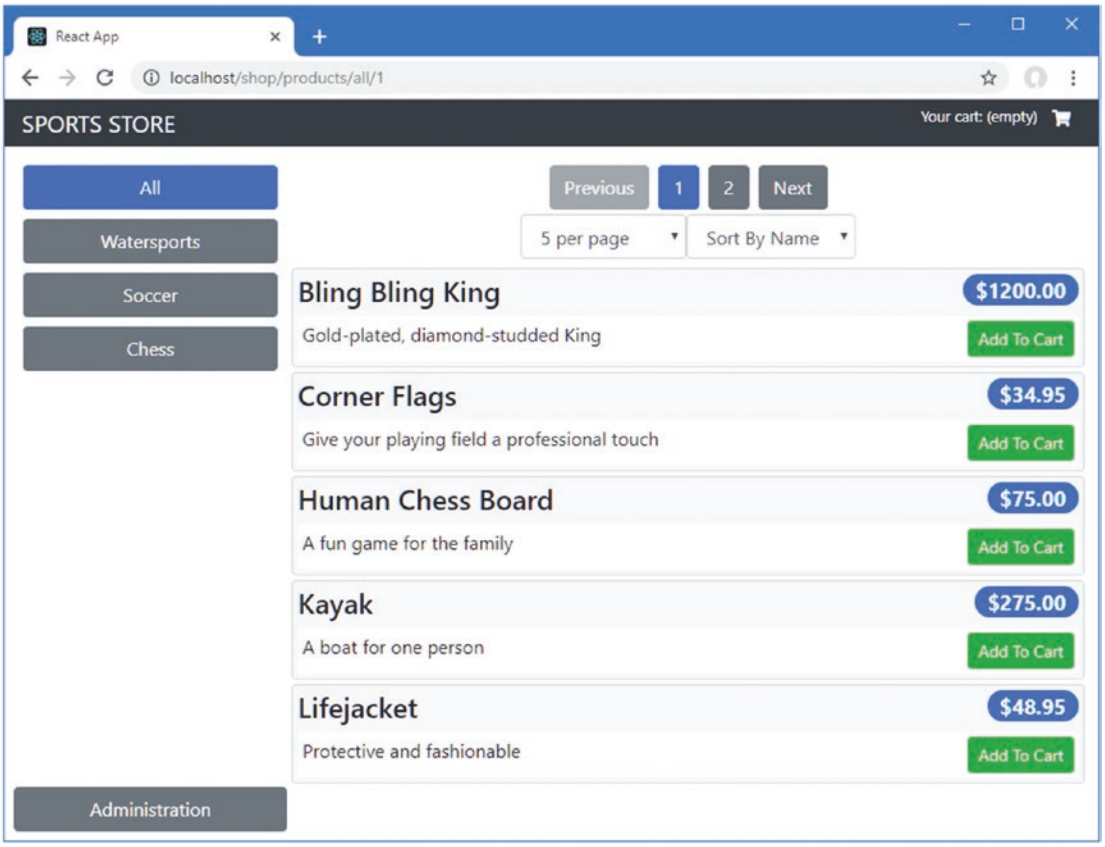
Running the Application

Once the image has been created, create and start a new container using the command shown in Listing 8-23.

Listing 8-23. Starting the Docker Container

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

You can test the application by opening `http://localhost` in the browser, which will display the response provided by the web server running in the container, as shown in Figure 8-5.



Bling Bling King $1200.00
Gold-plated, diamond-studded King

Corner Flags $34.95
Give your playing field a professional touch

Human Chess Board $75.00
A fun game for the family

Kayak $275.00
A boat for one person

Lifejacket $48.95
Protective and fashionable

Figure 8-5. *Running the containerized SportsStore application*

To stop the container, run the command shown in Listing 8-24.

Listing 8-24. 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 8-25.

Listing 8-25. Stopping the Container

```
docker stop ecc84f7245d6
```

The application is ready to deploy to any platform that supports Docker.

Summary

This chapter completes the SportsStore application, showing how a React application can be prepared for deployment and how easy it is to put a React 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.