CHAPTER 7

SportsStore: Administration

In this chapter, I add the administration features to the SportsStore application, providing the tools required to manage orders and products. I use GraphQL in this chapter rather than expanding the RESTful web service I used for the customer-facing part of SportsStore. GraphQL is an alternative to conventional web services that puts the client in control of the data it receives, although it requires more initial setup and can be more complex to use.

Preparing for This Chapter

This chapter builds on the SportsStore project created in Chapter 5 and modified in Chapter 6. To prepare for this chapter, I am going to generate a number of fake orders so there is data to work with, as shown in Listing 7-1.

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.

Listing 7-1. Altering the Application Data in the data.js File in the sportsstore Folder

```
var faker = require("faker");
faker.seed(100);
var categories = ["Watersports", "Soccer", "Chess"];
var products = [];
for (let i = 1; i <= 503; i++) {
    var category = faker.helpers.randomize(categories);
    products.push({
        id: i,
        name: faker.commerce.productName(),
        category: category,
        description: `${category}: ${faker.lorem.sentence(3)}`,
        price: Number(faker.commerce.price())
    })
}
var orders = [];
for (let i = 1; i <= 103; i++) {
    var fname = faker.name.firstName(); var sname = faker.name.lastName();
```


A. Freeman, *Pro React 16*, https://doi.org/10.1007/978-1-4842-4451-7_7

```
    var order = {
        id: i, name: `${fname} ${sname}`,
        email: faker.internet.email(fname, sname),
        address: faker.address.streetAddress(), city: faker.address.city(),
        zip: faker.address.zipCode(), country: faker.address.country(),
        shipped: faker.random.boolean(), products:[]
    }
    var productCount = faker.random.number({min: 1, max: 5});
    var product_ids = [];
    while (product_ids.length < productCount) {
        var candidateId = faker.random.number({ min: 1, max: products.length});
        if (product_ids.indexOf(candidateId) === -1) {
            product_ids.push(candidateId);
        }
    }
    for (let j = 0; j < productCount; j++) {
        order.products.push({
            quantity: faker.random.number({min: 1, max: 10}),
            product_id: product_ids[j]
        })
    }
    orders.push(order);
}

module.exports = () => ({ categories, products, orders })
```

Running the Example Application

Open a new command prompt, navigate to the `sportsstore` folder, and run the command shown in Listing 7-2.

Listing 7-2. Running the Example Application

```
npm start
```

The React development tools and the RESTful web service will start. Once the development tools have compiled the SportsStore application, a new browser window will open and display the content shown in Figure 7-1.

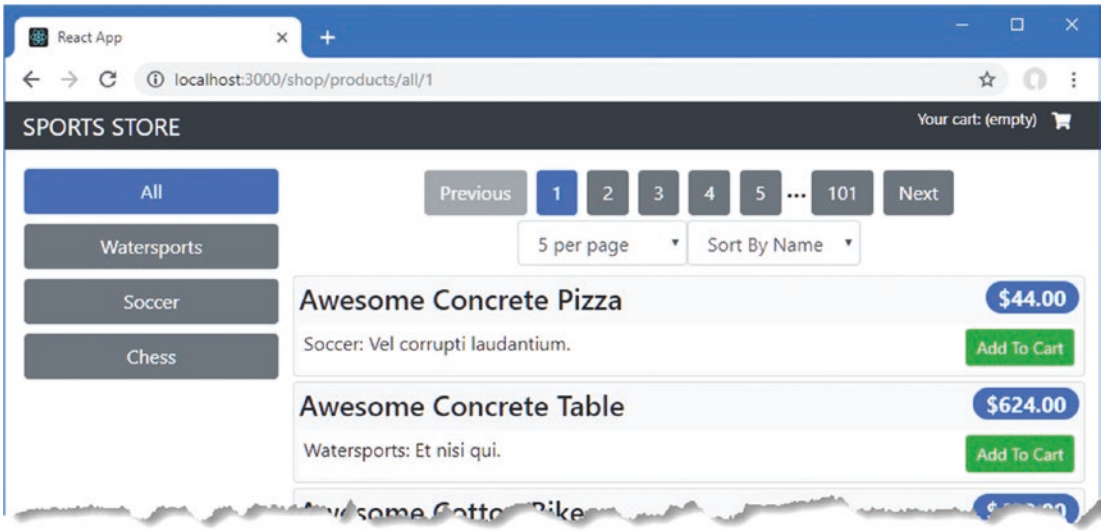

Figure 7-1. *Running the example application*

Creating a GraphQL Service

The administration features that I add to the SportsStore application in this chapter will use GraphQL instead of a RESTful web service. Few real applications would need to mix REST and GraphQL for the same data, but I want to demonstrate both approaches to remote services.

GraphQL isn't specific to React development, but it is so closely associated with React that I included an introduction to GraphQL in Chapter 24 and demonstrated the different ways a GraphQL service can be consumed by a React application in Chapter 25.

■ **Tip** I am going to create a custom GraphQL server for the SportsStore application so that I can share data with the RESTful web service provided by the excellent `json-server` package. As I explain in Chapter 24, there are open source and commercial GraphQL servers available.

Defining the GraphQL Schema

GraphQL requires that all of its operations are defined in a schema. To define the schema for the queries the service will support, I created a file called `serverQueriesSchema.graphql` in the `sportsstore` folder with the content shown in Listing 7-3.

Listing 7-3. The Contents of serverQueriesSchema.graphql in the sportsstore Folder

```
type product { id: ID!, name: String!, description: String! category: String!
               price: Float! }

type productPage { totalSize: Int!, products(sort: String, page: Int, pageSize: Int): [product]}

type orderPage { totalSize: Int, orders(sort: String, page: Int, pageSize: Int): [order]}
```

```
type order {
    id: ID!, name: String!, email: String!, address: String!, city: String!,
    zip: String!, country: String!, shipped: Boolean, products: [productSelection]
}

type productSelection { quantity: Int!, product: product }

type Query {
    product(id: ID!): product
    products(category: String, sort: String, page: Int, pageSize: Int): productPage
    categories: [String]
    orders(onlyUnshipped: Boolean): orderPage
}
```

The GraphQL specification includes a schema language used to define the features that a service provides. The schema in Listing 7-3 defines queries for products, categories, and orders. The `product` and `order` queries support pagination and return results that include a `totalSize` property that reports the number of items available so the client can present the user with pagination controls. The products can be filtered by category, and the orders can be filtered so that only unshipped orders are shown.

In GraphQL, changes are performed using *mutations*, following the theme of separating operations to read and write data that is common to much of React development. I added a file called `serverMutationsSchema.graphql` to the `sportsstore` folder and used it to define the mutations shown in Listing 7-4.

Listing 7-4. The Contents of the serverMutationsSchema.graphql File in the sportsstore Folder

```
input productStore {
    name: String!, description: String!, category: String!, price: Float!
}

input productUpdate {
    id: ID!, name: String, description: String, category: String, price: Float
}

type Mutation {
    storeProduct(product: productStore): product
    updateProduct(product: productUpdate): product
    deleteProduct(id: ID!): product
    shipOrder(id: ID!, shipped: Boolean!): order
}
```

The schema in Listing 7-4 defines mutations for storing new products, updating and deleting existing products, and marking orders as shipped or unshipped.

Defining the GraphQL Resolvers

The schema in a GraphQL service is implemented by a resolver. To provide the resolver for the queries, I added a file called `serverQueriesResolver.js` in the `sportsstore` folder with the code shown in Listing 7-5.

Listing 7-5. The Contents of the serverQueriesResolver.js File in the sportsstore Folder

```
const paginateQuery = (query, page = 1, pageSize = 5) =>
    query.drop((page - 1) * pageSize).take(pageSize);

const product = ({id}, {db}) => db.get("products").getById(id).value();

const products = ({ category }, { db}) => ({
    totalSize: () => db.get("products")
        .filter(p => category ? new RegExp(category, "i").test(p.category) : p)
        .size().value(),
    products: ({page, pageSize, sort}) => {
        let query = db.get("products");
        if (category) {
            query = query.filter(item =>
                new RegExp(category, "i").test(item.category))
        }
        if (sort) { query = query.orderBy(sort) }
        return paginateQuery(query, page, pageSize).value();
    }
})

const categories = (args, {db}) => db.get("categories").value();

const resolveProducts = (products, db) =>
    products.map(p => ({
        quantity: p.quantity,
        product: product({ id: p.product_id} , {db})
    }))

const resolveOrders = (onlyUnshipped, { page, pageSize, sort}, { db }) => {
    let query = db.get("orders");
    if (onlyUnshipped) { query = query.filter({ shipped: false}) }
    if (sort) { query = query.orderBy(sort) }
    return paginateQuery(query, page, pageSize).value()
        .map(order => ({ ...order, products: () =>
            resolveProducts(order.products, db) }));
}

const orders = ({onlyUnshipped = false}, {db}) => ({
    totalSize: () => db.get("orders")
        .filter(o => onlyUnshipped ? o.shipped === false : o).size().value(),
    orders: (...args) => resolveOrders(onlyUnshipped, ...args)
})

module.exports = { product, products, categories, orders }
```

The code in Listing 7-5 implements the queries defined in Listing 7-3. You can see an example of a stand-alone custom GraphQL server in Chapter 24, but the code in Listing 7-5 relies on the Lowdb database that the `json-server` package uses for data storage and that is described in detail at https://github.com/typicode/lowdb.

Each query is resolved using a series of functions invoked when the client requests specific fields, ensuring that the server has to load and process only the data that is needed. For the `orders` query, for example, the chain of functions ensures that the server only has to query the database for the related `product` objects if the client asks for them, avoiding retrieving data that is not required.

To implement the mutations, I added a file called `serverMutationsResolver.js` to the `sportsstore` folder and added the code shown in Listing 7-6.

Listing 7-6. The Contents of the serverMutationsResolver.js File in the sportsstore Folder

```
const storeProduct = ({ product}, {db }) =>
    db.get("products").insert(product).value();

const updateProduct = ({ product }, { db }) =>
    db.get("products").updateById(product.id, product).value();

const deleteProduct = ({ id }, { db }) => db.get("products").removeById(id).value();

const shipOrder = ({ id, shipped }, { db }) =>
    db.get("orders").updateById(id, { shipped: shipped}).value()

module.exports = {
    storeProduct, updateProduct, deleteProduct, shipOrder
}
```

Each of the functions defined in Listing 7-6 corresponds to a mutation defined in Listing 7-4. The code required to implement the mutation is simpler than the queries because the queries required additional statements to filter and page data.

Updating the Server

In Chapter 5, I added the packages required to create a GraphQL server to the SportsStore project. In Listing 7-7, I have used these packages to add support for GraphQL to the back-end server that has been providing the SportsStore application with its RESTful web service.

Listing 7-7. Adding GraphQL 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 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("/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 additions load the schema and resolvers and use them to create a GraphQL service that shares a database with the existing RESTful web service. Stop the development tools and run the command shown in Listing 7-8 in the `sportsstore` folder to start them again, which will also start the GraphQL server.

Listing 7-8. Starting the Development Tools and Services

```
npm start
```

To make sure that the GraphQL server is running, navigate to `http://localhost:3500/graphql`, which will display the tool shown in Figure 7-2.

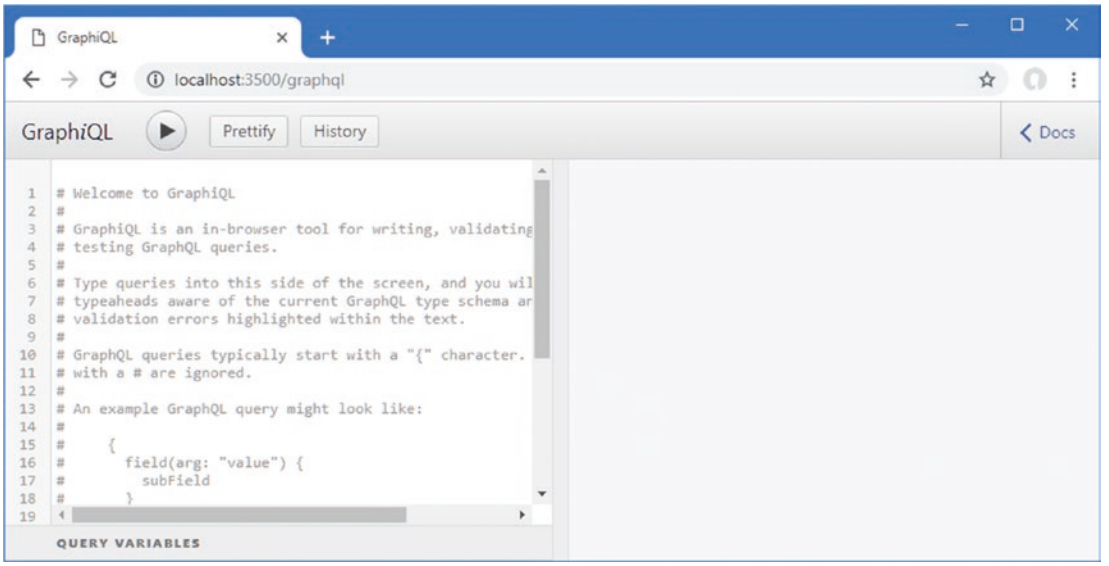

Figure 7-2. *The GraphiQL browser*

The package I used to create the GraphQL server includes the GraphiQL browser, which makes it easy to explore a GraphQL service. Replace the welcome message in the left part of the window with the GraphQL mutation shown in Listing 7-9.

■ **Note** The data used by the RESTful web service and GraphQL service is reset each time the `npm start` command 2 is used, which means that the change made by the mutation in Listing 7-9 will be lost when you next start the server. I convert the SportsStore application to a persistent database as part of the deployment preparations in Chapter 8.

Listing 7-9. A GraphQL Mutation

```
mutation {
    updateProduct(product: {
        id: 272, price: 100
    }) { id, name, category, price }
}
```

Click the Execute Query button to send the mutation to the GraphQL server, which will update a product in the database and produce the following result:

```
...
{
  "data": {
    "updateProduct": {
      "id": "272",
      "name": "Awesome Concrete Pizza",
      "category": "Soccer",
      "price": 100
    }
  }
}
...
```

Navigate back to `http://localhost:3000` (or reload the browser tab if it is still open), and you will see that the price of the first product shown in the list has changed, as shown in Figure 7-3.

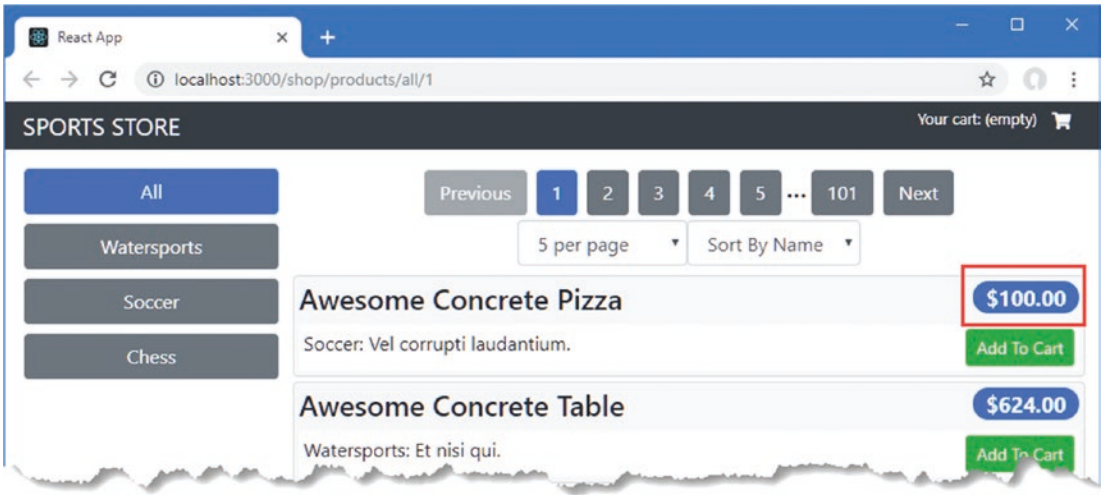

Figure 7-3. *The effect of a GraphQL mutation*

Creating the Order Administration Features

GraphQL requires more work at the server to create the schema and write the resolvers, but the benefit is that the client can be much simpler than one that uses a RESTful web service. In part, this is because of the way that GraphQL uses well-defined but flexible queries, but it is also because the GraphQL client package provides a lot of useful features that I had to create manually in Chapters 5 and 6.

■ **Note** The way that I use GraphQL in the SportsStore chapter is the simplest approach, but it hides the detail of how GraphQL works. In Chapter 25, I demonstrate how to use GraphQL directly over HTTP and also how to integrate GraphQL into an application that uses a data store.

Defining the Order Table Component

I am going to start by creating a display of the orders. To define the component that displays the order data, I added a file called `OrdersTable.js` in the `src/admin` folder and added the code shown in Listing 7-10.

Listing 7-10. The Contents of the OrdersTable.js File in the src/admin Folder

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

export class OrdersTable extends Component {

    render = () =>
        <div>
            <h4 className="bg-info text-white text-center p-2">
                { this.props.totalSize } Orders
            </h4>

            <PaginationControls keys={["ID", "Name"]}
                { ...this.props } />

            <table className="table table-sm table-striped">
                <thead>
                    <tr><th>ID</th>
                        <th>Name</th><th>Email</th>
                        <th className="text-right">Total</th>
                        <th className="text-center">Shipped</th>
                    </tr>
                </thead>
                <tbody>
                    { this.props.orders.map(order =>
                        <OrdersRow key={ order.id }
                            order={ order} toggleShipped={ () =>
                                this.props.toggleShipped(order.id, !order.shipped) }
                        />
                    )}
                </tbody>
            </table>
        </div>
}
```

The `OrdersTable` component displays the total number of orders and renders a table where responsibility for each row is delegated to the `OrdersRow` component, which I defined by adding a file called `OrdersRow.js` to the `src/admin` folder with the code shown in Listing 7-11.

Listing 7-11. The Contents of the OrdersRow.js File in the src/admin Folder

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

export class OrdersRow extends Component {

    calcTotal = (products) => products.reduce((total, p) =>
        total += p.quantity * p.product.price, 0).toFixed(2)

    getShipping = (order) => order.shipped
        ? <i className="fa fa-shipping-fast text-success" />
        : <i className="fa fa-exclamation-circle text-danger" />

    render = () =>
        <tr>
            <td>{ this.props.order.id }</td>
            <td>{this.props.order.name}</td>
            <td>{ this.props.order.email }</td>
            <td className="text-right">
                ${ this.calcTotal(this.props.order.products) }
            </td>
            <td className="text-center">
                <button className="btn btn-sm btn-block bg-muted"
                        onClick={ this.props.toggleShipped }>
                    { this.getShipping(this.props.order )}
                    <span>
                        { this.props.order.shipped
                            ? " Shipped" : " Pending"}
                    </span>
                </button>
            </td>
        </tr>
}
```

Defining the Connector Component

When a GraphQL client queries its server, it provides values for any parameters the query defines and specifies the data fields that it wants to receive. This is the biggest difference from most RESTful web services, and it means that GraphQL clients receive only the data values they require. It does mean, however, that a client-side query has to be defined before data can be retrieved from the server. I like to define queries separately from components, and I added a file called `clientQueries.js` to the `src/admin` folder with the content shown in Listing 7-12.

Listing 7-12. The Contents of the clientQueries.js File in the src/admin Folder

```
import gql from "graphql-tag";

export const ordersSummaryQuery = gql`
    query($onlyShipped: Boolean, $page:Int, $pageSize:Int, $sort:String) {
        orders(onlyUnshipped: $onlyShipped) {
            totalSize,
```

```
            orders(page: $page, pageSize: $pageSize, sort: $sort) {
                id, name, email, shipped
                products {
                    quantity, product { price }
                }
            }
        }
    }`
```

GraphQL queries are defined as JavaScript string literals in the client application but must be processed using the `gql` function from the `graphql-tag` package. The query in Listing 7-12 targets the server's `orders` query and will accept variables that are used for the query's `onlyShipped`, `page`, `pageSize`, and `sort` parameters. The client query selects only the fields it requires and incorporates details of the product data related to each order, which is included in the query results generated by the server's resolver for the `orders` query.

The GraphQL client package, `React-Apollo`, provides the `graphql` function, which is the counterpart to the `connect` and `withRouter` functions used earlier and which connects a component to the GraphQL features by creating a *higher-order component*, which is a function that provides features to a component, as described in Chapter 14. To create the connection between the `OrdersTable` component and the query defined in Listing 7-12, I added a file called `OrdersConnector.js` to the `src/admin` folder and added the code shown in Listing 7-13.

Listing 7-13. The Contents of the OrdersConnector.js File in the src/admin Folder

```
import { graphql } from "react-apollo";
import { ordersSummaryQuery } from "./clientQueries";
import { OrdersTable } from "./OrdersTable";

const vars = {
    onlyShipped: false, page: 1, pageSize: 10, sort: "id"
}

export const OrdersConnector = graphql(ordersSummaryQuery,
    {
        options: (props) => ({ variables: vars }),
        props: ({data: { loading, orders, refetch }}) => ({
            totalSize: loading ? 0 : orders.totalSize,
            orders: loading ? []: orders.orders,
            currentPage: vars.page,
            pageCount: loading ? 0 : Math.ceil(orders.totalSize / vars.pageSize),
            navigateToPage: (page) => { vars.page = Number(page); refetch(vars)},
            pageSize: vars.pageSize,
            setPageSize: (size) => { vars.pageSize = Number(size); refetch(vars)},
            sortKey: vars.sort,
            setSortProperty: (key) => { vars.sort = key; refetch(vars)},
        })
    }
)(OrdersTable)
```

The graphql function accepts arguments for the query and a configuration object and returns a function that is used to wrap a component and provide it access to the query features. There are many properties supported by the configuration object, but I require only two. The first is the options property, which is used to create the set of variables that will be applied to the GraphQL query, using a function that receives the props applied by the parent component.

■ **Tip** The Apollo GraphQL client caches the results from queries so that it doesn't send duplicate requests to the server, which is useful when using components with routing, for example.

The second is the props property, which is used to create the props that will be passed to the display component and is provided with a data object that combines details of the query progress, the response from the server, and the functions used to refresh the query.

I selected three properties from the data object and used them to create the props for the OrdersTable component. The loading property is true while the query is sent to the server and the response is awaited, which allows me to use placeholder values until the GraphQL response is received. The results of the query are assigned to a property given the query name, which is orders in this case. The response from a query is structured like this:

```
...
{ "orders":
  { "totalSize":103,
    "orders":[
      {"id":"1","name":"Velva Dietrich","email":"Velva_Dietrich@yahoo.com",
       "shipped":false, "products":[{"quantity":8,"product":{"price":84 },
      {"quantity":7,"product":{"price":125}, {"quantity":3,"product":{"price":352}
          ...other data values omitted for brevity...
  }
}
...
```

To get the total number of available orders, for example, I read the value of the orders.totalSize property, like this:

```
...
totalSize: loading ? 0 : orders.totalSize,
...
```

The value of the totalSize prop is zero until the result from the server has been received and is then assigned the value of orders.totalSize.

The third property I selected from the data object is refetch, which is a function that resends the query and which I use to respond to pagination changes.

```
...
navigateToPage: (page) => { vars.page = Number(page); refetch(vars)},
...
```

I pass all of the query variables to the refetch function for brevity, but any values the function receives are merged with the original variables, which can be useful for more complex queries.

■ **Tip** There is also a `fetchMore` function available that can be used to retrieve data and merge it with existing results, which is useful for components that gradually build up the data they present to the user. I have taken a simpler approach for the SportsStore application, and each page of data replaces the previous query results.

Configuring the GraphQL Client

Access to the GraphQL client features is provided through the `ApolloProvider` component. To configure the GraphQL client and to create a convenient placeholder for other administration features, I created the `src/admin` folder and added to it a file called `Admin.js`, which I used to define the component shown in Listing 7-14.

Listing 7-14. The Contents of 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"

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

export class Admin extends Component {

    render() {
        return <ApolloProvider client={ graphQlClient }>
            <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 p-2">
                    <OrdersConnector />
                </div>
            </div>
        </div>
        </ApolloProvider>
    }
}
```

To get started with the administration features, I am going to display an `OrdersTable` component, which I will create in the next section. I'll return to `Admin` and use URL routing to display additional features. To set the URL that will be used to communicate with the GraphQL server, I added the statement shown in Listing 7-15 to the `Urls.js` file.

Listing 7-15. 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`;
```

GraphQL requires only one URL because, unlike REST, it doesn't use the URL or the HTTP method to describe an operation. In Chapter 8, I will change the URLs used by the application as I prepare the project for deployment.

To incorporate the new features into the application, I added the route shown in Listing 7-16 to the App component.

Listing 7-16. Adding a Route in 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";

export default class App extends Component {

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

Save the changes to the files and navigate to `http://localhost:3000/admin`, and you will see the results shown in Figure 7-4.

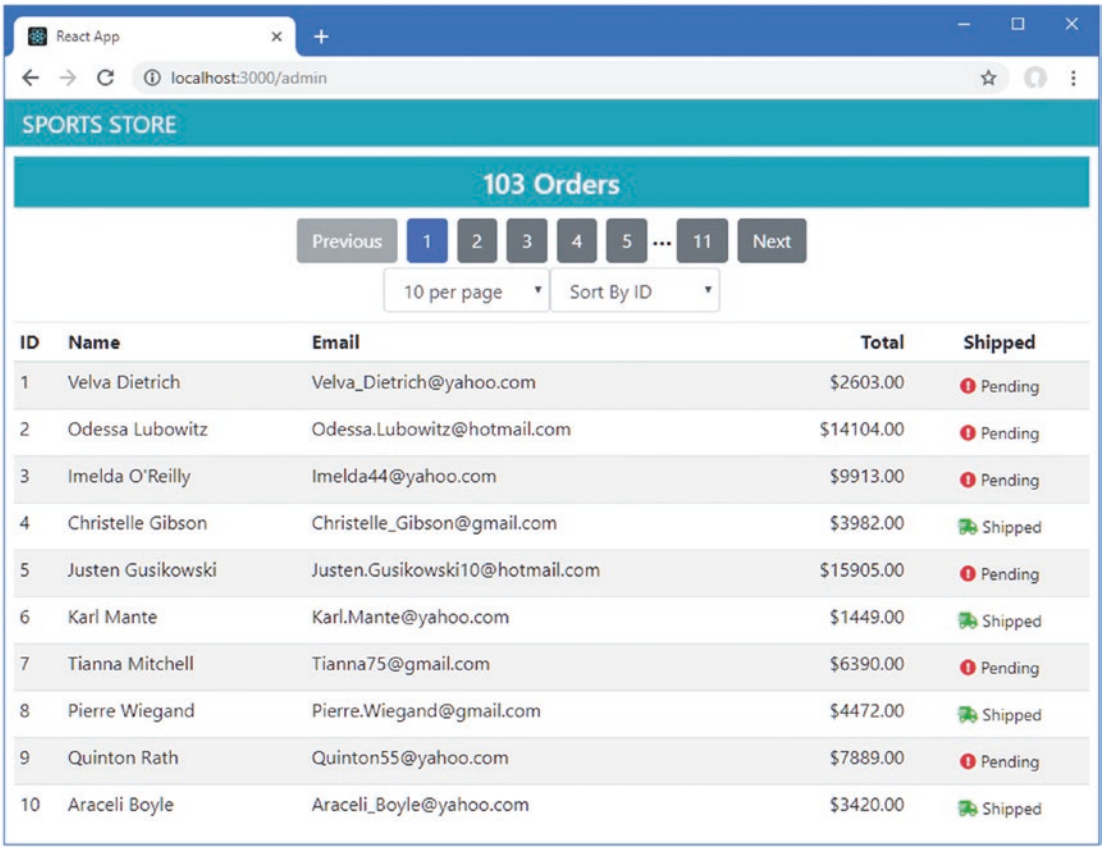

Figure 7-4. *Making a GraphQL query from a component*

Configuring the Mutation

The same basic approach for queries can be applied to integrate mutations into a React application. To allow the administrator to mark orders as shipped, I added a file called `clientMutations.js` to the `src/admin` folder with the content shown in Listing 7-17.

Listing 7-17. The Contents of the clientMutations.js File in the src/admin Folder

```
import gql from "graphql-tag";

export const shipOrder = gql`
    mutation($id: ID!, $shipped: Boolean!) {
        shipOrder(id: $id, shipped: $shipped) {
            id, shipped
        }
    }`
```

The GraphQL targets the `shipOrder` mutation, which updates the `shipped` property of an order specified by the value of its `id` property. In Listing 7-18 I have used the `graphql` function to provide access to the mutation and its results.

Listing 7-18. Applying a Mutation in the OrdersConnector.js File in the src/admin Folder

```
import { graphql, compose } from "react-apollo";
import { ordersSummaryQuery } from "./clientQueries";
import { OrdersTable } from "./OrdersTable";
import { shipOrder } from "./clientMutations";

const vars = {
    onlyShipped: false, page: 1, pageSize: 10, sort: "id"
}

export const OrdersConnector = compose(
    graphql(ordersSummaryQuery,
        {
            options: (props) => ({ variables: vars }),
            props: ({data: { loading, orders, refetch }}) => ({
                totalSize: loading ? 0 : orders.totalSize,
                orders: loading ? []: orders.orders,
                currentPage: vars.page,
                pageCount: loading ? 0 : Math.ceil(orders.totalSize / vars.pageSize),
                navigateToPage: (page) => { vars.page = Number(page); refetch(vars)},
                pageSize: vars.pageSize,
                setPageSize: (size) =>
                    { vars.pageSize = Number(size); refetch(vars)},
                sortKey: vars.sort,
                setSortProperty: (key) => { vars.sort = key; refetch(vars)},
            })
        }
    ),
    graphql(shipOrder, {
        props: ({ mutate }) => ({
            toggleShipped: (id, shipped) => mutate({ variables: { id, shipped }})
        })
    })
)(OrdersTable);
```

The React-Apollo package provides the `compose` function that simplifies combining queries and mutations. The existing query is combined with another call to the `graphql` function, which is passed the mutation from Listing 7-17. When using a mutation, the `props` property in the configuration object receives a function named `mutate`, which I use to create a prop called `toggleShipped`, corresponding to the prop used by the `OrdersRow` component to change the status of an order. To see the result, click the Shipped/Pending indicator for an order in the table, and its status will be changed, as shown in Figure 7-5.

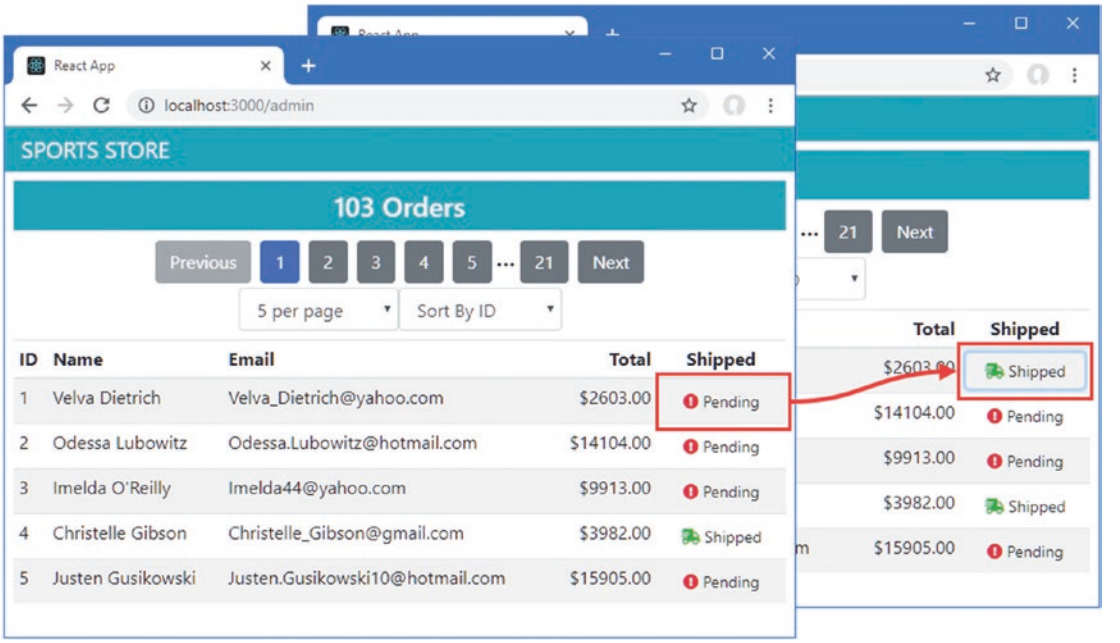

Figure 7-5. *Using a mutation*

The Apollo client automatically updates its cache of data when there is a change, which means that the change to the value of the shipped property is automatically reflected in the data displayed by the `OrdersTable` component.

Creating the Product Administration Features

To provide administration of the products presented to the user, I added a file called `ProductsTable.js` to the `src/admin` folder and used it to define the component shown in Listing 7-19.

Listing 7-19. The Contents of the ProductsTable.js File in the src/admin Folder

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

export class ProductsTable extends Component {

    render = () =>
        <div>
            <h4 className="bg-info text-white text-center p-2">
                { this.props.totalSize } Products
            </h4>
```

```
        <PaginationControls keys={["ID", "Name", "Category"]}
            { ...this.props } />

        <table className="table table-sm table-striped">
            <thead>
                <tr><th>ID</th>
                    <th>Name</th><th>Category</th>
                    <th className="text-right">Price</th>
                    <th className="text-center"></th>
                </tr>
            </thead>
            <tbody>
                { this.props.products.map(prod =>
                    <ProductsRow key={ prod.id} product={ prod }
                        deleteProduct={ this.props.deleteProduct } />
                )}
            </tbody>
        </table>
        <div className="text-center">
            <Link to="/admin/products/create" className="btn btn-primary">
                Create Product
            </Link>
        </div>
    </div>
}
```

The `ProductsTable` component receives an array of objects through its `products` prop and uses the `ProductsRow` component to generate a table row for each of them. There is also a `Link` styled as a button that will be used to navigate to the component that will allow new products to be created.

To create the `ProductsRow` component that is responsible for a single table row, I added a file called `ProductsRow.js` to the `src/admin` folder and added the code shown in Listing 7-20.

Listing 7-20. The Contents of the ProductsRow.js File in the src/admin Folder

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

export class ProductsRow extends Component {

    render = () =>
        <tr>
            <td>{ this.props.product.id }</td>
            <td>{this.props.product.name}</td>
            <td>{ this.props.product.category }</td>
            <td className="text-right">
                ${ this.props.product.price.toFixed(2) }
            </td>
            <td className="text-center">
                <button className="btn btn-sm btn-danger mx-1"
                    onClick={ () =>
                        this.props.deleteProduct(this.props.product.id) }>
                            Delete
```

```
                </button>
                <Link to={`/admin/products/${this.props.product.id}`}
                    className="btn btn-sm btn-warning">
                        Edit
                </Link>
            </td>
        </tr>
}
```

Table cells are rendered for the `id`, `name`, `category`, and `price` properties. There is a `button` that invokes a function prop named `deleteProduct` that will remove a product from the database, and there is a `Link` that will navigate to the component used to edit product details.

Connecting the Product Table Component

To connect the product table component to the GraphQL data, I added the queries shown in Listing 7-21 to the `clientQueries.js` file, which also include the query I will require for editing a product. These queries correspond to the server-side GraphQL defined at the start of the chapter.

Listing 7-21. Adding Queries in the clientQueries.js File in the src/admin Folder

```
import gql from "graphql-tag";

export const ordersSummaryQuery = gql`
    query($onlyShipped: Boolean, $page:Int, $pageSize:Int, $sort:String) {
        orders(onlyUnshipped: $onlyShipped) {
            totalSize,
            orders(page: $page, pageSize: $pageSize, sort: $sort) {
                id, name, email, shipped
                products {
                    quantity, product { price }
                }
            }
        }
    }`

export const productsList = gql`
    query($page: Int, $pageSize: Int, $sort: String) {
        products {
            totalSize,
            products(page: $page, pageSize: $pageSize, sort: $sort) {
                id, name, category, price
            }
        }
    }`

export const product = gql`
    query($id: ID!) {
        product(id: $id) {
            id, name, description, category, price
        }
    }`
```

The query assigned to the constant named `productsList` will retrieve the `id`, `name`, `category`, and `price` properties for a page of products. The query assigned to the constant named `product` will retrieve the `id`, `name`, `description`, `category`, and `price` properties of a single `product` object. To add support for deleting, creating, and editing objects, I added the mutations shown in Listing 7-22 to the `clientMutations.js` file.

Listing 7-22. Adding Mutations in the clientMutations.js File in the src/admin Folder

```
import gql from "graphql-tag";

export const shipOrder = gql`
    mutation($id: ID!, $shipped: Boolean!) {
        shipOrder(id: $id, shipped: $shipped) {
            id, shipped
        }
    }`

export const storeProduct = gql`
    mutation($product: productStore) {
        storeProduct(product: $product) {
            id, name, category, description, price
        }
    }`

export const updateProduct = gql`
    mutation($product: productUpdate) {
        updateProduct(product: $product) {
            id, name, category, description, price
        }
    }`

export const deleteProduct = gql`
    mutation($id: ID!) {
        deleteProduct(id: $id) {
            id
        }
    }`
```

The new mutations correspond to the server-side GraphQL defined at the start of the chapter and allow the client to store a new product, edit an existing product, and delete a product.

Having defined the queries and mutations, I added a file called `ProductsConnector.js` to the `src/admin` folder and defined the higher-order component shown in Listing 7-23.

Listing 7-23. The Contents of the ProductsConnector.js File in the src/admin Folder

```
import { graphql, compose } from "react-apollo";
import { ProductsTable } from "./ProductsTable";
import { productsList } from "./clientQueries";
import { deleteProduct } from "./clientMutations";

const vars = {
    page: 1, pageSize: 10, sort: "id"
}
```

```
export const ConnectedProducts = compose(
    graphql(productsList,
        {
            options: (props) => ({ variables: vars }),
            props: ({data: { loading, products, refetch }}) => ({
                totalSize: loading ? 0 : products.totalSize,
                products: loading ? []: products.products,
                currentPage: vars.page,
                pageCount: loading ? 0
                    : Math.ceil(products.totalSize / vars.pageSize),
                navigateToPage: (page) => { vars.page = Number(page); refetch(vars)},
                pageSize: vars.pageSize,
                setPageSize: (size) =>
                    { vars.pageSize = Number(size); refetch(vars)},
                sortKey: vars.sort,
                setSortProperty: (key) => { vars.sort = key; refetch(vars)},
            })
        }
    ),
    graphql(deleteProduct,
        {
            options: {
                update: (cache, { data: { deleteProduct: { id }}}) => {
                    const queryDetails = { query: productsList, variables: vars };
                    const data = cache.readQuery(queryDetails)
                    data.products.products =
                        data.products.products.filter(p => p.id !== id);
                    data.products.totalSize = data.products.totalSize - 1;
                    cache.writeQuery({...queryDetails, data });
                }
            },
            props: ({ mutate }) => ({
                deleteProduct: (id) => mutate({ variables: { id }})
            })
    })
)(ProductsTable);
```

The code in Listing 7-23 is similar to the corresponding code for the orders administration features. One key difference is that mutations that remove objects do not automatically update the local cached data. For this type of mutation, an `update` function must be defined that modifies the cached data directly, like this:

```
...
update: (cache, { data: { deleteProduct: { id }}}) => {
    const queryDetails = { query: productsList, variables: vars };
    const data = cache.readQuery(queryDetails)
    data.products.products = data.products.products.filter(p => p.id !== id);
    data.products.totalSize = data.products.totalSize - 1;
    cache.writeQuery({...queryDetails, data });
}
...
```

This function reads the cached data, removes an object, reduces the `totalSize` to reflect the deletion, and then writes the data back to the cache, which will have the effect of updating the product list without needing to query the server.

■ **Tip** The downside of this approach is that it doesn't repaginate the data to reflect the deletion, which means that the page displays fewer items until the user navigates to another page. In the next section, I demonstrate how to address this by clearing the cached data, which leads to an additional GraphQL query but ensures that the application is consistent.

Creating the Editor Components

To allow the user to create a new product, I added a file called `ProductEditor.js` to the `src/admin` folder and defined the component shown in Listing 7-24.

Listing 7-24. The Contents of the ProductEditor.js File in the src/admin Folder

```
import React, { Component } from "react";
import { Query } from "react-apollo";
import { ProductCreator } from "./ProductCreator";
import { product } from "./clientQueries";

export class ProductEditor extends Component {

    render = () =>
        <Query query={ product } variables={ {id: this.props.match.params.id} } >
            { ({ loading, data }) => {
                if (!loading) {
                    return <ProductCreator {...this.props } product={data.product}
                        mode="edit" />
                }
                return null;
            }}
        </Query>
}
```

The `Query` component is provided as an alternative to the `graphql` function and allows GraphQL queries to be performed declaratively, with the results and other client features presented through a *render prop function*, which is described in Chapter 14. The `ProductEditor` component defined in Listing 7-24 will obtain the `id` of the product that the administrator wants to edit and obtains it using the `Query` component, which is configured using its `query` and `variables` props. The render prop function receives an object with `loading` and `data` properties, which have the same purpose as for the `graphql` function I used earlier. The `ProductEditor` component renders no content while the loading property is `true` and then displays a `ProductCreator` component, passing the data received from the query through the prop named `product`.

The `ProductCreator` component will do double duty in the SportsStore application. When used on its own, it will present the administrator with an empty form that will be sent to the `storeProduct` mutation. When it is used by the `ProductEditor` component, it will show details of an existing product and send the form data to the `updateProduct` mutation. To define the component, I added a file called `ProductCreator.js` to the `src/admin` folder with the code shown in Listing 7-25.

Listing 7-25. The Contents of the ProductCreator.js File in the src/admin Folder

```
import React, { Component } from "react";
import { ValidatedForm } from "../forms/ValidatedForm";
import { Mutation } from "react-apollo";
import { storeProduct, updateProduct } from "./clientMutations";

export class ProductCreator extends Component {

    constructor(props) {
        super(props);
        this.defaultAttrs = { type: "text", required: true };
        this.formModel = [
            { label: "Name" }, { label: "Description" },
            { label: "Category" },
            { label: "Price", attrs: { type: "number"}}
        ];
        this.mutation = storeProduct;
        if (this.props.mode === "edit" ) {
            this.mutation = updateProduct;
            this.formModel = [ { label: "Id", attrs: { disabled: true }},
                     ...this.formModel]
                .map(item => ({ ...item, attrs: { ...item.attrs,
                    defaultValue: this.props.product[item.label.toLowerCase()]} }));
        }
    }

    navigate = () => this.props.history.push("/admin/products");

    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 m-2">
                    <Mutation mutation={ this.mutation }>
                        { (saveMutation, {client }) => {
                            return <ValidatedForm formModel={ this.formModel }
                                defaultAttrs={ this.defaultAttrs }
                                submitCallback={ data => {
                                    saveMutation({variables: { product:
                                        { ...data, price: Number(data.price) }}});
                                    if (this.props.mode !== "edit" ) {
                                        client.resetStore();
                                    }
                                    this.navigate();
                                }}
```

```
                                cancelCallback={ this.navigate }
                                submitText="Save" cancelText="Cancel" />
                        }}
                    </Mutation>
                </div>
            </div>
        </div>
    }
}
```

The `ProductCreator` component relies on the `ValidatedForm` that I created in Chapter 6 to handle checkout from the shopping part of the application. The form is configured with the fields required to edit a product, which will include the values obtained from the GraphQL query when they are provided through the `product` prop.

The counterpart to the `Query` component is `Mutation`, which allows a mutation to be used within the `render` function. The render prop function receives a function that is invoked to send the mutation to the server and that accepts an object that provides the variables for the mutation, like this:

```
...
<Mutation mutation={ this.mutation }>
    { (saveMutation, {client }) => {
        return <ValidatedForm formModel={ this.formModel }
            defaultAttrs={ this.defaultAttrs }
            submitCallback={ data => {
                saveMutation({variables: { product:
                    { ...data, price: Number(data.price) }}});
                    if (this.props.mode !== "edit" ) {
                        client.resetStore();
                    }
                    this.navigate();
                }}
                cancelCallback={ this.navigate }
                submitText="Save" cancelText="Cancel" />
        }
    }
</Mutation>
...
```

I have highlighted the section of code that sets up the function prop that is passed to the `ValidatedForm` component and that sends the mutation when it is invoked. When an object is updated, the Apollo client automatically updates its cached data to reflect the change, just as when I marked orders as shipped earlier in the chapter. New objects are not automatically processed, which means that the application has to take responsibility for managing the cache. The approach I took for deleting an object was to update the existing cache, but that is a much more complex process for a new item because it means trying to work out whether it should be displayed on the current page and, if so, where in the sort order it would appear. As a simpler alternative, I have received a `client` parameter from the render prop function, which allows me to clear the cached data through its `resetStore` method. When the `navigate` function sends the browser back to the product list, a fresh GraphQL will be sent to the server, which ensures that the data is consistently paged and sorted, albeit at the cost of an additional query.

Updating the Routing Configuration

The final step is to update the routing configuration to add navigation buttons that allow the order and product administration features to be selected, as shown in Listing 7-26.

Listing 7-26. Updating the Routing Configuration 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";

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

export class Admin extends Component {

    render() {
        return <ApolloProvider client={ graphQlClient }>
            <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>
                </div>
                <div className="col-9 p-2">
                    <Switch>
                        <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>
    }
}
```

Save the changes, and you will see the layout shown in Figure 7-6. Clicking the Products button will display a paged table of products, which can be deleted and edited using the buttons in each table row.

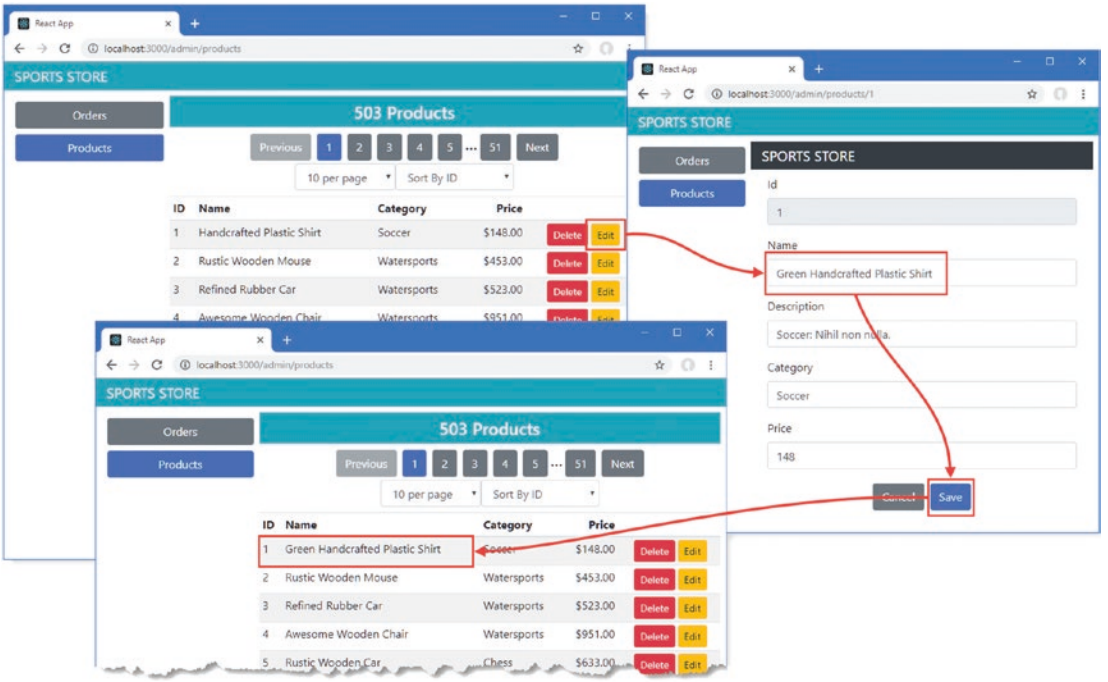

Figure 7-6. *The product administration features*

Clicking the Create Product button will display an editor that allows new products to be defined, as shown in Figure 7-7.

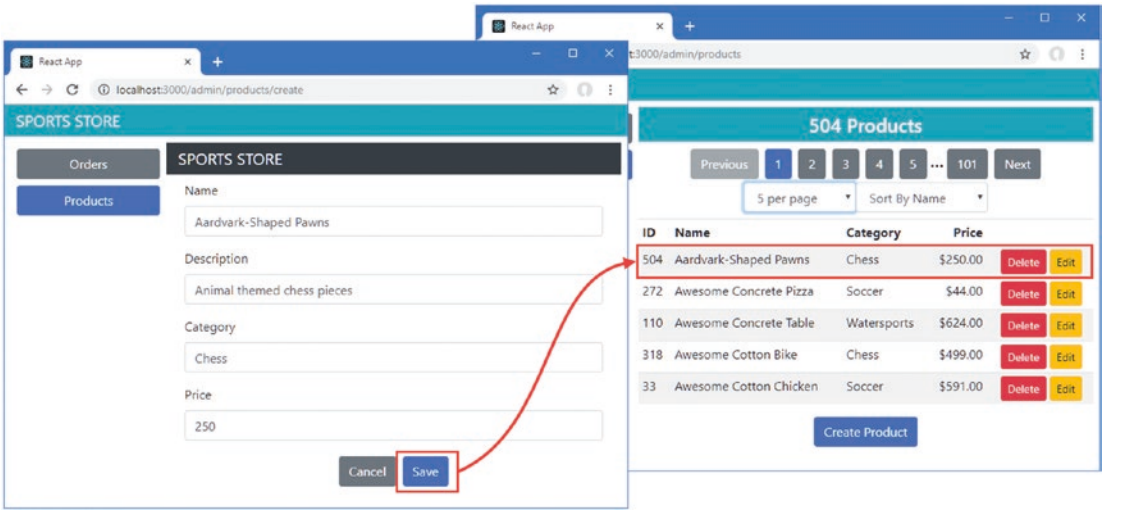

Figure 7-7. *Creating a new product*

Summary

In this chapter, I added the administration features to the SportsStore application. I started by creating a GraphQL service with the queries and mutations required to manage the order and products data. I used the GraphQL service to expand the application features, relying on the GraphQL client to manage the data in the application so that I didn't need to create and manage a data store. In the next chapter, I add authentication for the administration features and prepare the application for deployment.