CHAPTER 6

SportsStore: REST and Checkout

In this chapter, I continue adding features to the SportsStore application I created in Chapter 5. I add support for retrieving data from a web service, presenting larger amounts of data in pages and checking out and placing orders.

Preparing for This Chapter

No preparation is required for this chapter, which uses the SportsStore project created in Chapter 5.
To start the React development tools and the RESTful web service, open a command prompt, navigate to the `sportsstore` folder, and run the command shown in Listing 6-1.

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 6-1. Starting the Development Tools and Web Service

```
npm start
```

The initial build process will take a few seconds, after which a new browser window or tab will open and display the SportsStore application, as shown in Figure 6-1.

A. Freeman, *Pro React 16*, https://doi.org/10.1007/978-1-4842-4451-7_6

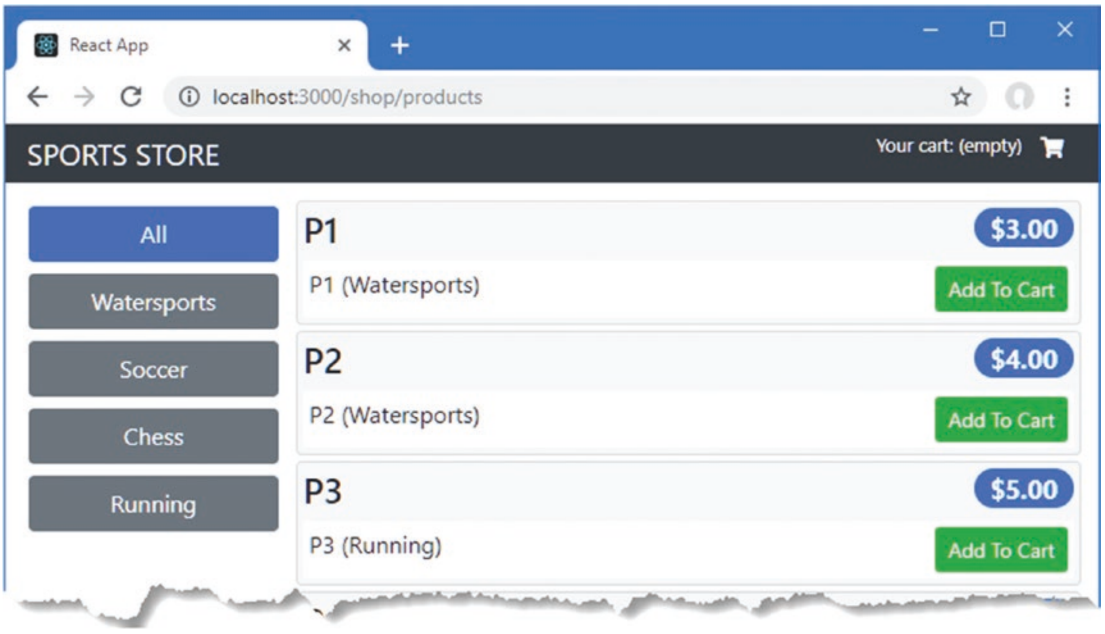


Figure 6-1. *Running the SportsStore application*

Consuming the RESTful Web Service

The basic structure of the SportsStore application is taking shape, and I have enough functionality in place to remove the placeholder data and start using the RESTful web service. In Chapter 7, I use GraphQL, which is a more flexible (and complex) alternative to REST web services, but regular web services are common, and I am going to use a REST web service to provide the SportsStore application with its product data and to submit orders at the end of the checkout process.

I describe REST in more detail in Chapter 23, but for this chapter, I need just one basic HTTP request to get started. Open a new browser tab and request `http://localhost:3500/api/products`. The browser will send an HTTP GET request to the web service that was created in Chapter 5 and started by the command in Listing 6-1. The GET method combined with the URL tells the web service that a list of the products is required and produces the following result:

```
...
[{"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":19.5},
 {"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}]
...
```

The web service responds to requests using the JSON data format, which is easy to deal with in a React application since it is similar to the JavaScript object literal form described in Chapter 4. In the sections that follow, I'll create a foundation for working with the web service and use it to replace the static data that is currently displayed by the SportsStore application.

Creating a Configuration File

Projects often require different URLs in production and development. To avoid hard-coding the URLs into individual JavaScript files, I added a file called `Urls.js` to the `src/data` folder and used it to define the configuration data shown in Listing 6-2.

Listing 6-2. The Contents of 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`
}
```

When I prepare the SportsStore application for deployment in Chapter 8, I will be able to configure the URLs required to access the web service in one place. I have used the data types already defined for the data store for consistency, which helps keeps references to the different types of data consistent and reduces the risk of a typo.

Creating a Data Source

I added a file called `RestDataSource.js` to the `src/data` folder and added the code shown in Listing 6-3. I want to consolidate the code that is responsible for sending HTTP requests to the web service and processing the results, allowing me to keep it contained in one place in the project.

Listing 6-3. The Contents of the RestDataSource.js File in the src/data Folder

```
import Axios from "axios";
import { RestUrls } from "./Urls";

export class RestDataSource {

    GetData = (dataType) =>
        this.SendRequest("get", RestUrls[dataType]);

    SendRequest = (method, url) => Axios.request({ method, url });
}
```

The `RestDataSource` class uses the Axios package to make HTTP requests to the web service. Axios is described in Chapter 23 and is a popular package for handling HTTP because it provides a consistent API and automatically processes responses to transform JSON into JavaScript objects. In Listing 6-3, the `GetData` method uses Axios to send an HTTP request to the web service to get all of the available objects for a specified data type. The result from the `GetData` method is a `Promise` that is resolved when the response is received from the web service.

Extending the Data Store

HTTP requests sent by JavaScript code are performed asynchronously. This doesn't fit well with the default behavior of the Redux data store, which responds to changes only when an action is processed by a reducer.

Redux data stores can be extended to support asynchronous operations using a middleware function, which inspects the actions that are sent to the data store and alters them before they are processed. In Chapter 20, I create data store middleware that intercepts actions and delays them while it performs asynchronous requests to get data.

For the SportsStore application, I am going to take a different approach and add support for actions whose payload is a `Promise`, which I described briefly in Chapter 4. The middleware will wait until the `Promise` is resolved and then pass on the action using the outcome of the `Promise` as the payload. I added a file called `AsyncMiddleware.js` to the `src/data` folder and added the code shown in Listing 6-4.

Listing 6-4. The Contents of the AsyncMiddleware.js File in the src/data Folder

```
const isPromise = (payload) =>
    (typeof(payload) === "object" || typeof(payload) === "function")
        && typeof(payload.then) === "function";

export const asyncActions = () => (next) => (action) => {
    if (isPromise(action.payload)) {
        action.payload.then(result => next({...action, payload: result}));
    } else {
        next(action)
    }
}
```

The code in Listing 6-4 contains a function that checks to see whether an action's payload is a `Promise`, which it does by looking for function or objects that have a `then` function. The `asyncAction` function will be used as the data store middleware, and it calls `then` on the `Promise` to wait for it to be resolved, at which point it uses the result to replace the payload and passes it on, using the `next` function, which continues the normal path through the data store. Actions whose payloads are not a `Promise` are passed on immediately. In Listing 6-5, I have added the middleware to the data store.

Listing 6-5. Adding Middleware in the DataStore.js File in the src/data Folder

```
import { createStore, applyMiddleware } from "redux";
import { ShopReducer } from "./ShopReducer";
import { CartReducer } from "./CartReducer";
import { CommonReducer } from "./CommonReducer";
import { asyncActions } from "./AsyncMiddleware";

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

The `applyMiddleware` is used to wrap the middleware so that it receives the actions, and the result is passed as an argument to the `createStore` function that creates the data store. The effect is that the `asyncActions` function defined in Listing 6-4 will be able to inspect all of the actions sent to the data store and seamlessly deal with those with a `Promise` payload.

Updating the Action Creator

In Listing 6-6, I removed the placeholder data from the store action creator and replaced it with a `Promise` that sends a request using the data source.

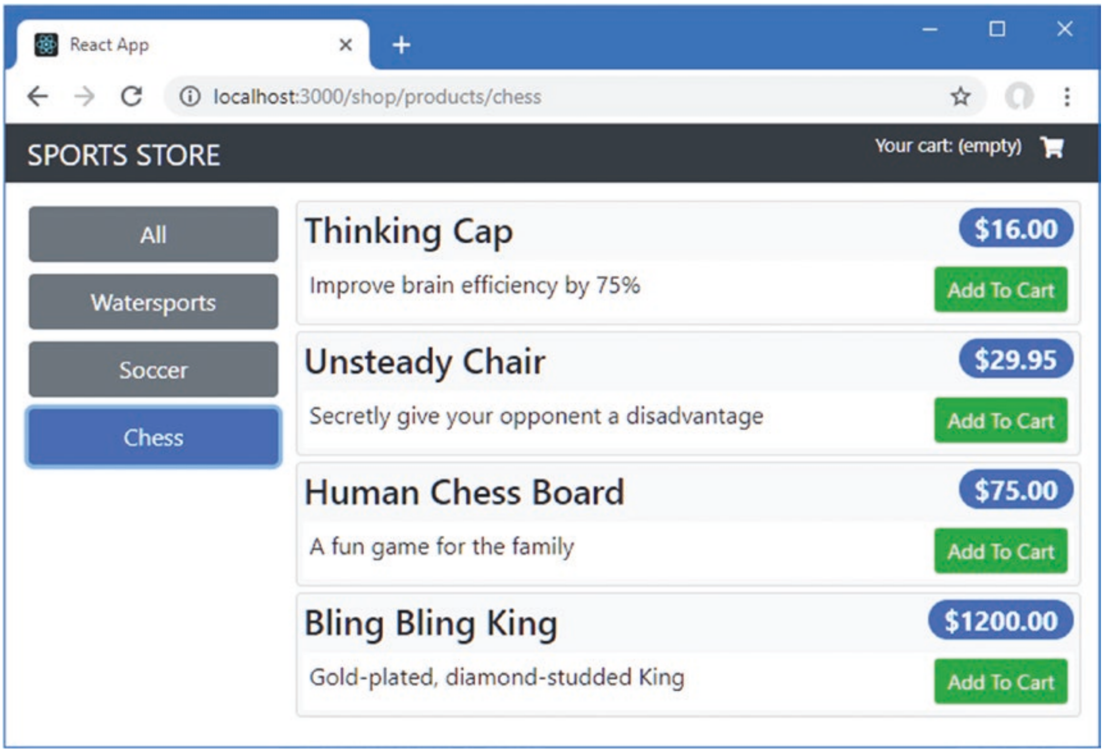
Listing 6-6. Using a Promise in the ActionCreators.js File in the src/data Folder

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

const dataSource = new RestDataSource();

export const loadData = (dataType) => ({
    type: ActionTypes.DATA_LOAD,
    payload: dataSource.GetData(dataType)
        .then(response => ({ dataType, data: response.data}))
});
```

When the action object created by the `loadData` function is received by the data store, the middleware defined in Listing 6-5 will wait for the response to be received from the web service and then pass on the action for normal processing, with the result that the SportsStore application displays data obtained remotely, as shown in Figure 6-2.



Thinking Cap $16.00
Improve brain efficiency by 75%

Unsteady Chair $29.95
Secretly give your opponent a disadvantage

Human Chess Board $75.00
A fun game for the family

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

Figure 6-2. *Using data from a web service*

Paginating Data

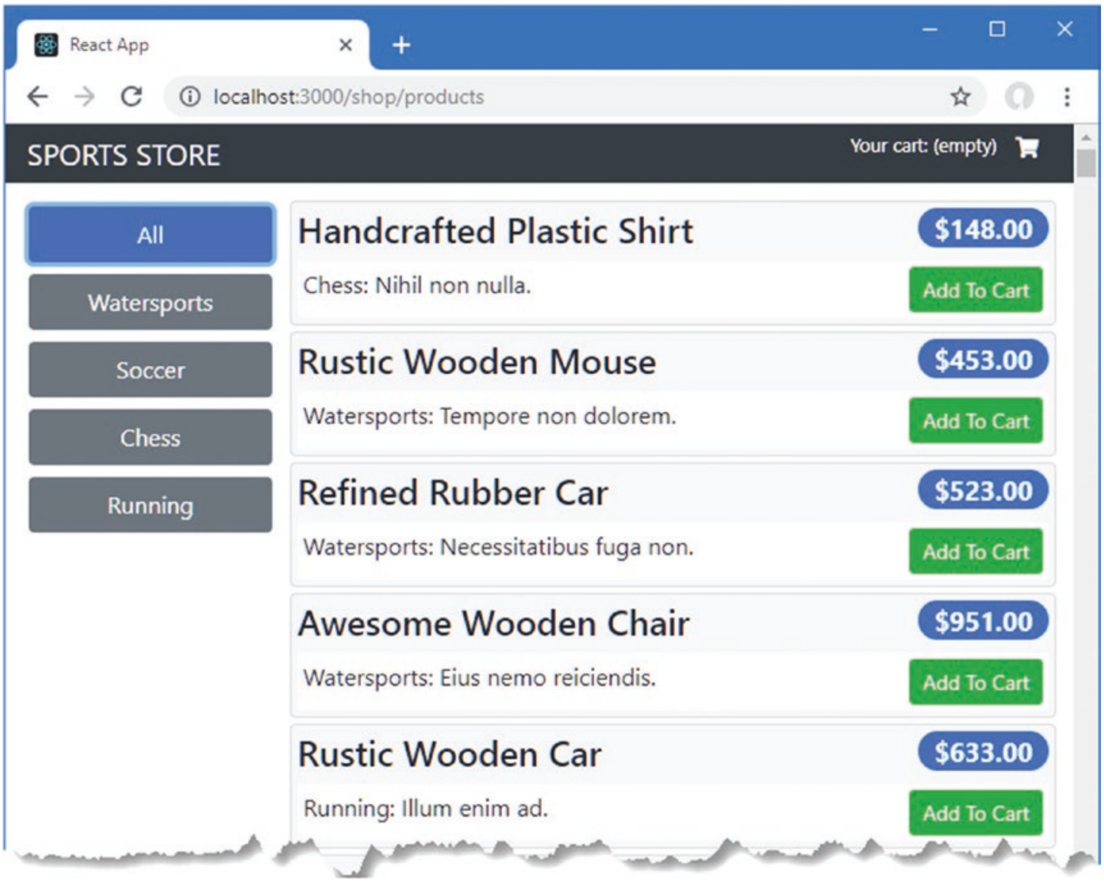
The SportsStore application is now receiving data from the web service, but most applications have to deal with larger amounts of data, which must be presented to the user in pages. In Listing 6-7, I have used the Faker.js package to generate a larger number of products to replace the data presented by the web service.

Listing 6-7. Increasing the Amount of Data in the data.js File in the sportsstore Folder

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

```
module.exports = function () {
    return {
        categories: categories,
        products: data,
        orders: []
    }
}
```

The Faker.js package is an excellent tool for easily generating data for development and testing, providing contextual data through an API described at `https://github.com/Marak/Faker.js`. When you save the `data.js` file, the change will be detected by the server code created in Chapter 5 and loaded into the web service. Reload the SportsStore application in the browser window, and you will see all of the new products shown in a single list, as shown in Figure 6-3. The user can still filter the products using the category buttons, but there is still too much data presented in one go.



Handcrafted Plastic Shirt $148.00
Chess: Nihil non nulla.

Rustic Wooden Mouse $453.00
Watersports: Tempore non dolorem.

Refined Rubber Car $523.00
Watersports: Necessitatibus fuga non.

Awesome Wooden Chair $951.00
Watersports: Eius nemo reiciendis.

Rustic Wooden Car $633.00
Running: Illum enim ad.

Figure 6-3. *Generating more data for testing pagination*

■ **Tip** The code in Listing 6-7 creates 503 product objects. It is a good idea to use numbers of objects that are not divisible by the size of the pages you intend to support so that you can be sure that your code deals with a few stragglers on the last page.

Understanding the Web Service Pagination Support

Pagination requires support from the server so that it provides the client with the means to request a subset of the available data and information about how much data is available. There is no standard approach to providing pagination, and you should consult the documentation for the server or service you are using.

The `json-server` package that provides the RESTful web service for the SportsStore application supports pagination through query strings. Open a new browser window and request the URL shown in Listing 6-8 to see how pagination works.

Listing 6-8. Requesting a Page of Data

```
http://localhost:3500/api/products?category_like=watersports&_page=2&_limit=3&_sort=name
```

The query string for this URL—the part that follows the `?` character—asks the web service to return a page of products from a specific category, using the fields described in Table 6-1.

Table 6-1. *The Query String Fields Required for Pagination*

Name	Description
`category_like`	This field filters the results to include only those objects whose category property matches the field value, which is `Watersports` in the example URL. If the category field is omitted, then products from all categories will be included in the results.
`_page`	This field selects the page number.
`_limit`	This field selects the page size.
`_sort`	This field specifies the property by which the objects will be sorted before being paginated.

The URL in Listing 6-8 asks the web service to return the second page containing three products from the set that have a `category` value of `Watersports`, sorted by the `name` property, producing the following results:

```
...
[
 {"id":469,"name":"Awesome Fresh Pants","category":"Watersports",
    "description":"Watersports: Quia quam aut.","price":864},
 {"id":19,"name":"Awesome Frozen Car","category":"Watersports",
      "description":"Watersports: A rerum mollitia.","price":314},
  {"id":182,"name":"Awesome Granite Fish", "category":"Watersports",
      description":"Watersports: Hic omnis incidunt.","price":521}
]
...
```

The web service response contains headers that help the client make future requests. Use the browser to request the URL shown in Listing 6-9.

Listing 6-9. Making a Simpler Pagination Request

```
http://localhost:3500/api/products?_page=2&_limit=3
```

The simpler URL makes the result headers easier to understand. Use the browser's F12 developer tools to inspect the response, and you will see that it contains the following headers:

```
...
X-Total-Count: 503
Link: <http://localhost:3500/api/products?_page=1&_limit=3>; rel="first",
      <http://localhost:3500/api/products?_page=1&_limit=3>; rel="prev",
      <http://localhost:3500/api/products?_page=3&_limit=3>; rel="next",
      <http://localhost:3500/api/products?_page=168&_limit=3>; rel="last"
...
```

These are not the only headers in the response, but they have been added specifically to help the client with future pagination requests. The `X-Total-Count` header provides the total number of objects that are matched by the request URL, which is useful for determining the total number of pages. Since there is no `category` field in the URL in Listing 6-9, the server has reported that 503 objects are available.

The `Link` header provides a set of URLs that can be used to query the first and last pages, and the pages before and after the current pages, although clients are not required to use the `Link` header to formulate subsequent requests.

Changing the HTTP Request and Action

In Listing 6-10, I changed the formulation of the URL for the request that obtains the product data to include request parameters, which will be used to request pages and specify a category. The Axios package will use the parameters to add query string to the request URL.

Listing 6-10. Adding URL Parameters in the RestDataSource.js File in the src/data/rest Folder

```
import Axios from "axios";
import { RestUrls } from "./Urls";

export class RestDataSource {

    GetData = async(dataType, params) =>
        this.SendRequest("get", RestUrls[dataType], params);

    SendRequest = (method, url, params) => Axios.request({
                method, url, params
    });
}
```

In Listing 6-11, I have updated the action created by the `loadData` action creator so that it includes parameters and adds additional information from the response to the data store.

Listing 6-11. Changing the Action in the ActionCreators.js File in the src/data Folder

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

const dataSource = new RestDataSource();

export const loadData = (dataType, params) => (
    {
        type: ActionTypes.DATA_LOAD,
        payload: dataSource.GetData(dataType, params).then(response =>
             ({ dataType,
                data: response.data,
                total: Number(response.headers["x-total-count"]),
                params
             })
        )
    })
```

When the `Promise` is resolved by the data store middleware, the action object that is sent to the reducer will contain `payload.total` and `payload.params` properties. The `total` property will contain the value of the `X-Total-Count` header, which I will use to create the pagination navigation controls. The `params` property will contain the parameters used to make the request, which I will use to determine when the user has made a change that requires an HTTP request for more data. In Listing 6-12, I have updated the reducer that processes the `DATA_LOAD` action so that the new action properties are added to the data store.

Listing 6-12. Adding Data Store Properties in the ShopReducer.js File in the src/data Folder

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

export const ShopReducer = (storeData, action) => {
    switch(action.type) {
        case ActionTypes.DATA_LOAD:
            return {
                ...storeData,
                [action.payload.dataType]: action.payload.data,
                [`${action.payload.dataType}_total`]: action.payload.total,
                [`${action.payload.dataType}_params`]: action.payload.params
            };
        default:
            return storeData || {};
    }
}
```

Creating the Data Loading Component

To create a component that takes care of obtaining the product data, I added a file called `DataGetter.js` to the `src/data` folder and used it to define the component shown in Listing 6-13.

Listing 6-13. The Contents of the DataGetter.js File in the src/data Folder

```
import React, { Component } from "react";
import { DataTypes } from "../data/Types";

export class DataGetter extends Component {

    render() {
        return <React.Fragment>{ this.props.children }</React.Fragment>
    }

    componentDidUpdate = () => this.getData();
    componentDidMount = () => this.getData();

    getData = () => {
        const dsData = this.props.products_params || {} ;
        const rtData = {
            _limit: this.props.pageSize || 5,
            _sort: this.props.sortKey || "name",
            _page: this.props.match.params.page || 1,
            category_like: (this.props.match.params.category || "") === "all"
                ? "" : this.props.match.params.category
        }

        if (Object.keys(rtData).find(key => dsData[key] !== rtData[key])) {
            this.props.loadData(DataTypes.PRODUCTS, rtData);
        }
    }
}
```

This component renders the content its parent provides between the start and end tags using the `children` props. This is useful for defining components that provide services to the application but that don't present content to the user. In this case, I need a component that can receive details of the current route and its parameters and also access the data store. The component's `componentDidMount` and `componentDidUpdate` methods, both part of the component lifecycle described in Chapter 13, call the `getData` method, which gets the parameters from the URL and compares them with those in the data store that were added after the last request. If there has been a change, a new action is dispatched that will load the data the user requires.

In addition to the category and page number, which are taken from the URL, the action is created with `_sort` and `_limit` parameters that order the results and set the data size. The values used for sorting and for setting the page size will be obtained from the data store.

Updating the Store Connector Component

To introduce the pagination support into the application, I updated the `ShopConnector` component, which is responsible for connecting the shop features in the application to the data store and the URL router. The changes in Listing 6-14 add the `DataGetter` component and remove the category filter for product data (since the products will already be filtered by the web service).

Listing 6-14. Adding Pagination in the ShopConnector.js File in the src/shop Folder

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

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

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

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

export const ShopConnector = connect(mapStateToProps, mapDispatchToProps)(
    class extends Component {
        render() {
            return <Switch>
                <Redirect from="/shop/products/:category"
                    to="/shop/products/:category/1" exact={ true } />
                <Route path={ "/shop/products/:category/:page" }
                    render={ (routeProps) =>
                        <DataGetter { ...this.props } { ...routeProps }>
                            <Shop { ...this.props } { ...routeProps }  />
                        </DataGetter>
                    } />
                <Route path="/shop/cart" render={ (routeProps) =>
                        <CartDetails { ...this.props } { ...routeProps }  />} />
                <Redirect to="/shop/products/all/1" />
            </Switch>
        }

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

I have updated the routing configuration to support pagination. The first routing change is the addition of a `Redirect`, which matches URLs that have a category but no page and redirects them to the URL for the first page of the selected category. I also changed the existing `Redirect` so that it redirects any unmatched URLs to `/shop/products/all`.

The result is a block of code that looks more complicated than it is. When the `ShopConnector` component is asked to render its content, it uses a `Route` to match the URL and get `category` and `parameters`, like this:

```
...
<Route path={ "/shop/products/:category/:page" }
...
```

Immediately before the `Route` is a `Redirect` that matches URLs that have one segment and redirects the browser to a URL that will select the first page:

```
...
<Redirect from="/shop/products/:category"
          to="/shop/products/:category/1" exact={ true } />
...
```

This redirection ensures that there is always category and page values to work with. The other `Redirect` matches any other URLs and redirects them to the URL for the first page of the products, unfiltered by category.

```
...
<Redirect to="/shop/products/all/1" />
...
```

Updating the All Category Button

The routing components used in Listing 6-14 require a corresponding change to the `All` category button so that it is highlighted when no category has been selected, as shown in Listing 6-15.

Listing 6-15. Updating the All Button in the CategoryNavigation.js File in the src/shop Folder

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

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>
            )}
        </React.Fragment>
    }
}
```

I have added `/all` to the URL matched by the `ToggleLink` component and set the `exact` prop to `false` so that URLs such as `/shop/products/all/1` will be matched. The effect is that the application requests individual pages of data from the web service, which is also responsible for filtering based on category. Each time the user clicks a category button, the `DataGetter` component requests new data, as shown in Figure 6-4.

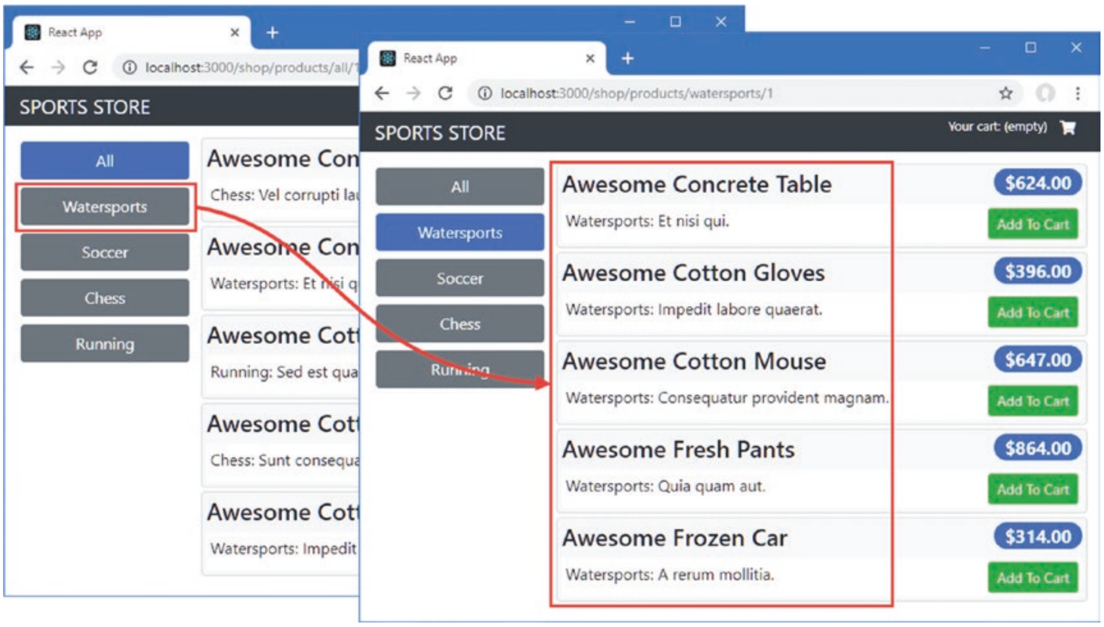


Figure 6-4. *Requesting pages of data from the web service*

Creating the Pagination Controls

The next step is to create a component that will allow the user to navigate to different pages and change the page size. Listing 6-16 defines new data store action types that will be used to change the page size and specify the property that will be used for sorting results.

Listing 6-16. Adding New Action Types in the Types.js File in the src/data Folder

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

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

In Listing 6-17, I added new action creators that create actions using the new types.

Listing 6-17. Defining Creators in the ActionCreators.js File in the src/data Folder

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

const dataSource = new RestDataSource();

export const loadData = (dataType, params) => (
    {
        type: ActionTypes.DATA_LOAD,
        payload: dataSource.GetData(dataType, params).then(response =>
             ({ dataType,
                data: response.data,
                total: Number(response.headers["x-total-count"]),
                params
             })
        )
    })

export const setPageSize = (newSize) =>
    ({ type: ActionTypes.DATA_SET_PAGESIZE, payload: newSize});

export const setSortProperty = (newProp) =>
    ({ type: ActionTypes.DATA_SET_SORT_PROPERTY, payload: newProp});
```

In Listing 6-18, I extended the reducer to support the new actions.

Listing 6-18. Supporting New Actions in the ShopReducer.js File in the src/data Folder

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

export const ShopReducer = (storeData, action) => {
    switch(action.type) {
        case ActionTypes.DATA_LOAD:
            return {
                ...storeData,
                [action.payload.dataType]: action.payload.data,
                [`${action.payload.dataType}_total`]: action.payload.total,
                [`${action.payload.dataType}_params`]: action.payload.params
            };
        case ActionTypes.DATA_SET_PAGESIZE:
            return { ...storeData, pageSize: action.payload }
        case ActionTypes.DATA_SET_SORT_PROPERTY:
            return { ...storeData, sortKey: action.payload }
        default:
            return storeData || {};
    }
}
```

To produce the HTML elements that will allow the user to use the pagination features, I added a file called `PaginationControls.js` to the `src` folder and used it to define the component shown in Listing 6-19.

Listing 6-19. The Contents of the PaginationControls.js File in the src Folder

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

export class PaginationControls extends Component {

    constructor(props) {
        super(props);
        this.pageSizes = this.props.sizes || [5, 10, 25, 100];
        this.sortKeys = this.props.keys || ["Name", "Price"];
    }

    handlePageSizeChange = (ev) => {
        this.props.setPageSize(ev.target.value);
    }

    handleSortPropertyChange = (ev) => {
        this.props.setSortProperty(ev.target.value);
    }

    render() {
        return <div className="m-2">
                <div className="text-center m-1">
                    <PaginationButtons currentPage={this.props.currentPage}
                        pageCount={this.props.pageCount}
                        navigate={ this.props.navigateToPage }/>
                </div>
                <div className="form-inline justify-content-center">
                    <select className="form-control"
                            onChange={ this.handlePageSizeChange }
                            value={ this.props.pageSize|| this.pageSizes[0] }>
                        { this.pageSizes.map(s =>
                            <option value={s} key={s}>{s} per page</option>
                        )}
                    </select>
                    <select className="form-control"
                            onChange={ this.handleSortPropertyChange }
                            value={ this.props.sortKey || this.sortKeys[0] }>
                        { this.sortKeys.map(k =>
                            <option value={k.toLowerCase()} key={k}>
                                Sort By { k }
                            </option>
                        )}
                    </select>
                </div>
            </div>
    }
}
```

The PaginationControls component uses select elements to allow the user to change the page size and the property used to sort the results. The option elements that provide the individual values that can be selected can be configured using props, which will allow me to reuse this component for the administration features in Chapter 7. If no props are supplied, then default values suitable for paginating products are used.

The onChange prop is applied to the select elements to respond to user changes, which are handled by methods that receive the event triggered by the change and invoke function props that are received from the parent component.

The process of generating the buttons that will allow movement between pages has been delegated to a component named PaginationButtons. To create this component, I added a file called PaginationButtons.js to the src folder and added the code shown in Listing 6-20.

Listing 6-20. The Contents of the PaginationButtons.js File in the src Folder

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

export class PaginationButtons extends Component {

    getPageNumbers = () => {
        if (this.props.pageCount < 4) {
            return [...Array(this.props.pageCount + 1).keys()].slice(1);
        } else if (this.props.currentPage <= 4) {
            return [1, 2, 3, 4, 5];
        } else  if (this.props.currentPage > this.props.pageCount - 4) {
            return [...Array(5).keys()].reverse()
                .map(v => this.props.pageCount - v);
        } else {
            return [this.props.currentPage -1, this.props.currentPage,
                this.props.currentPage + 1];
        }
    }

    render() {
        const current = this.props.currentPage;
        const pageCount = this.props.pageCount;
        const navigate = this.props.navigate;
        return <React.Fragment>
            <button onClick={ () => navigate(current  - 1) }
                disabled={ current === 1 } className="btn btn-secondary mx-1">
                    Previous
            </button>
            { current > 4 &&
                <React.Fragment>
                    <button className="btn btn-secondary mx-1"
                        onClick={ () => navigate(1)}>1</button>
                    <span className="h4">...</span>
                </React.Fragment>
            }
            { this.getPageNumbers().map(num =>
                <button className={ `btn mx-1 ${num === current
                        ? "btn-primary": "btn-secondary"}`}
                    onClick={ () => navigate(num)} key={ num }>
                        { num }
                </button>)}
            { current <= (pageCount - 4) &&
                <React.Fragment>
                    <span className="h4">...</span>
                    <button className="btn btn-secondary mx-1"
                            onClick={ () => navigate(pageCount)}>
                        { pageCount }
                    </button>
                </React.Fragment>
            }
            <button onClick={ () => navigate(current + 1) }
                disabled={ current === pageCount }
                className="btn btn-secondary mx-1">
                    Next
            </button>
        </React.Fragment>
    }
}
```

Creating the pagination buttons is a complex process, and it is easy to get bogged down in the detail. The approach I have taken in Listing 6-20 aims to strike a balance between simplicity and providing the user with enough context to navigate through large amounts of data.

To connect the pagination controls to the product data in the store, I added a file called `ProductPageConnector.js` to the `src/shop` folder and defined the component shown in Listing 6-21.

Listing 6-21. The Contents of the ProductPageConnector.js File in the src/shop Folder

```
import { connect } from "react-redux";
import { withRouter } from "react-router-dom";
import { setPageSize, setSortProperty } from "../data/ActionCreators";

const mapStateToProps = dataStore => dataStore;
const mapDispatchToProps = { setPageSize, setSortProperty };

const mergeProps = (dataStore, actionCreators, router) => ({
    ...dataStore, ...router, ...actionCreators,
    currentPage: Number(router.match.params.page),
    pageCount: Math.ceil((dataStore.products_total
        | dataStore.pageSize || 5)/(dataStore.pageSize || 5)),
    navigateToPage: (page) => router.history
        .push(`/shop/products/${router.match.params.category}/${page}`),
})

export const ProductPageConnector = (PageComponent) =>
    withRouter(connect(mapStateToProps, mapDispatchToProps,
        mergeProps)(PageComponent))
```

As I explained earlier, the complexity in a React application often coalesces where different features are combined, which is the connector components in the SportsStore application. The code in Listing 6-21 creates a higher-order component (known as a *HOC* and described in Chapter 14), which is a function that provides features to another component through its props. The HOC is named `ProductPageConnector`, and it combines data store properties, action creators, and route parameters to provide the pagination control components with access to the features they require. The `connect` function is the same one I used in Chapter 5 to connect a component to the data store, and it has been used in conjunction with the `withRouter` function, which is its counterpart from the React Router package and which provides a component with the route details from the closest `Route`. In Listing 6-22, I have applied the higher-order component to the `PaginationControls` component and added the result to the content presented to the user.

Listing 6-22. Adding Pagination Controls in the Shop.js File in the src/shop Folder

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

const ProductPages = ProductPageConnector(PaginationControls);

export class Shop extends Component {

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

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

The result is a series of buttons allowing the user to move between pages, alongside select elements that change the sort property and the page size, as shown in Figure 6-5.

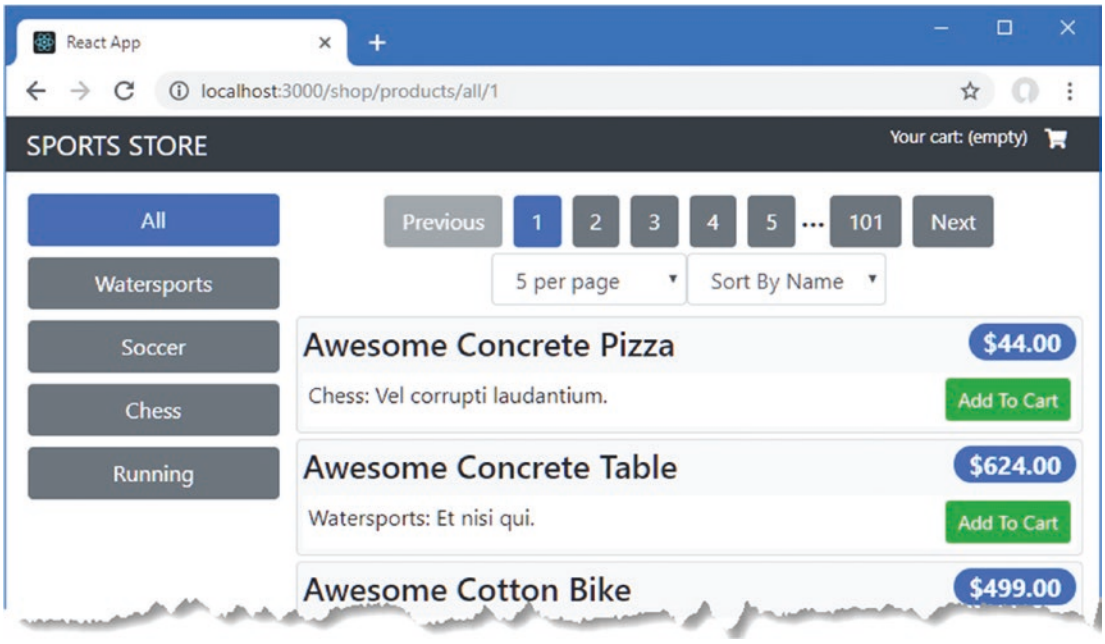


Figure 6-5. *Adding support for paginating products*

Adding the Checkout Process

The core features of the application are in place, allowing the user to filter and navigate through the product data and add items to a basket that are displayed in summary and detailed views. Once the user completes the checkout process, a new order must be sent to the web service, which will complete the shopping, reset the user's cart, and display a summary message. In the sections that follow, I add support for checking out and placing an order.

Extending the REST Data Source and the Data Store

As I explain in Chapter 23, when a RESTful web service receives an HTTP request, it uses a combination of the request method (also known as the *verb*) and the URL to determine what operation should be performed. To send an order to the web service, I am going to send a POST request to the web service's /orders URL. To keep the new features consistent with the existing application, I started by adding a constant that identifies the data type for orders and a new action for storing the order, as shown in Listing 6-23.

Listing 6-23. Adding Types in the Types.js File in the src/data Folder

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

export const ActionTypes = {
    DATA_LOAD: "data_load",
    DATA_STORE: "data_store",
    DATA_SET_SORT_PROPERTY: "data_set_sort",
    DATA_SET_PAGESIZE: "data_set_pagesize",
    CART_ADD: "cart_add",
    CART_UPDATE: "cart_update",
    CART_REMOVE: "cart_delete",
    CART_CLEAR: "cart_clear"
}
```

The new data type allows me to define the URL for placing the order, as shown in Listing 6-24. I also use it in Chapter 7 when I add support for administration features.

Listing 6-24. Adding a New 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`
}
```

In Listing 6-25, I added a method to the REST data source that receives the order object and sends it to the web service.

Listing 6-25. Adding a Method in the RestDataSource.js File in the src/data Folder

```
import Axios from "axios";
import { RestUrls } from "./Urls";

export class RestDataSource {

    constructor(err_handler) {
        this.error_handler = err_handler || (() => {});
    }

    GetData = (dataType, params) =>
        this.SendRequest("get", RestUrls[dataType], params);

    StoreData = (dataType, data) =>
        this.SendRequest("post", RestUrls[dataType], {}, data);

    SendRequest = (method, url, params, data) =>
        Axios.request({ method, url, params, data });
}
```

The Axios package will receive a data object and take care of formatting it so that it can be sent to the web service. In Listing 6-26, I added a new action creator that uses a Promise to send an order to the web service. The web service will return the stored data, which will include a unique identifier.

Listing 6-26. Adding a Creator to the ActionCreators.js File in the src/data Folder

```
import { ActionTypes, DataTypes } from "./Types";
import { RestDataSource } from "./RestDataSource";

const dataSource = new RestDataSource();

export const loadData = (dataType, params) => (
    {
        type: ActionTypes.DATA_LOAD,
        payload: dataSource.GetData(dataType, params).then(response =>
             ({ dataType,
                data: response.data,
                total: Number(response.headers["x-total-count"]),
                params
             })
        )
    })

export const setPageSize = (newSize) => {
    return ({ type: ActionTypes.DATA_SET_PAGESIZE, payload: newSize});
}

export const setSortProperty = (newProp) =>
    ({ type: ActionTypes.DATA_SET_SORT_PROPERTY, payload: newProp});
```

```
export const placeOrder = (order) => ({
        type: ActionTypes.DATA_STORE,
        payload: dataSource.StoreData(DataTypes.ORDERS, order).then(response => ({
            dataType: DataTypes.ORDERS, data: response.data
        }))
    })
```

To process the result and add the order to the data store, I added the reducer shown in Listing 6-27.

Listing 6-27. Storing an Order in the ShopReducer.js File in the src/data Folder

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

export const ShopReducer = (storeData, action) => {
    switch(action.type) {
        case ActionTypes.DATA_LOAD:
            return {
                ...storeData,
                [action.payload.dataType]: action.payload.data,
                [`${action.payload.dataType}_total`]: action.payload.total,
                [`${action.payload.dataType}_params`]: action.payload.params
            };
        case ActionTypes.DATA_SET_PAGESIZE:
            return { ...storeData, pageSize: action.payload }
        case ActionTypes.DATA_SET_SORT_PROPERTY:
            return { ...storeData, sortKey: action.payload }
        case ActionTypes.DATA_STORE:
            if (action.payload.dataType === DataTypes.ORDERS) {
                return { ...storeData, order: action.payload.data }
            }
            break;
        default:
            return storeData || {};
    }
}
```

Creating the Checkout Form

To complete a SportsStore order, the user must complete a form with their personal details, which means that I must present the user with a form. React supports two ways to use form elements: *controlled* and *uncontrolled*. For a controlled element, React manages the element's content and responds to its change events. The `select` elements used for configuring pagination fall into this category. For the checkout form, I am going to use uncontrolled elements, which are not closely managed by React and rely more on the browser's functionality. The key to using uncontrolled for elements is a feature called *refs*, described in Chapter 16, which allow a React component to keep track of the HTML elements that are produced by its `render` method after they have been displayed to the user. For the checkout form, the advantage of using `refs` is that I can validate the form using the HTML5 validation API, which I describe in Chapter 15. The validation API requires direct access to the form elements, which wouldn't be possible without the use of refs.

■ **Note** There are packages available for creating and validating forms in React applications, but they can be awkward to use and apply restrictions on the appearance of the form or the structure of the data that it produces. It is easy to create custom forms and validation using the features described in Chapters 15 and 16, which is the approach I have taken for the SportsStore chapter.

Creating the Validated Form

I am going to create a reusable form with validation that will generate the fields required programmatically. I created the `src/forms` folder and added to it a file called `ValidatedForm.js`, which I used to define the component shown in Listing 6-28.

Listing 6-28. The Contents of the ValidatedForm.js File in the src/forms Folder

```
import React, { Component } from "react";
import { ValidationError } from "./ValidationError";
import { GetMessages } from "./ValidationMessages";

export class ValidatedForm extends Component {

    constructor(props) {
        super(props);
        this.state = {
            validationErrors: {}
        }
        this.formElements = {};
    }

    handleSubmit = () => {
        this.setState(state => {
            const newState = { ...state, validationErrors: {} }
            Object.values(this.formElements).forEach(elem => {
                if (!elem.checkValidity()) {
                    newState.validationErrors[elem.name] = GetMessages(elem);
                }
            })
            return newState;
        }, () => {
            if (Object.keys(this.state.validationErrors).length === 0) {
                const data =  Object.assign(...Object.entries(this.formElements)
                    .map(e => ({[e[0]]: e[1].value})) )
                this.props.submitCallback(data);
            }
        });
    }
    registerRef = (element) => {
        if (element !== null) {
            this.formElements[element.name] = element;
        }
    }

    renderElement = (modelItem) => {
        const name = modelItem.name || modelItem.label.toLowerCase();
        return <div className="form-group" key={ modelItem.label }>
            <label>{ modelItem.label }</label>
            <ValidationError errors={ this.state.validationErrors[name] } />
            <input className="form-control" name={ name } ref={ this.registerRef }
                { ...this.props.defaultAttrs } { ...modelItem.attrs } />
        </div>
    }

    render() {
        return <React.Fragment>
            { this.props.formModel.map(m => this.renderElement(m))}
            <div className="text-center">
                <button className="btn btn-secondary m-1"
                        onClick={ this.props.cancelCallback }>
                    { this.props.cancelText || "Cancel" }
                </button>
                <button className="btn btn-primary m-1"
                        onClick={ this.handleSubmit }>
                    { this.props.submitText || "Submit"}
                </button>
            </div>
        </React.Fragment>
    }
}
```

The `ValidatedForm` component receives a data model and uses it to create a form that is validated using the HTML5 API. Each form element is rendered with a label and a `ValidationError` component that displays validation messages to the user. The form is displayed with buttons that cancel or submit the form using callback functions provided as props. The `submit` callback will not be invoked unless all of the elements meet their validation constraints.

When the submit callback is invoked, it will receive an object whose properties are the `name` attribute values for the form elements and whose values are the data entered into each field by the user.

Defining the Form

To create the component that is used to display error messages, I added a file called `ValidationError.js` to the `src/forms` folder and added the code shown in Listing 6-29.

Listing 6-29. The Contents of the ValidationError.js File in the src/forms Folder

```
import React, { Component } from "react";
export class ValidationError extends Component {

    render() {
        if (this.props.errors) {
            return this.props.errors.map(err =>
                <h6 className="text-danger" key={err}>
                    { err }
                </h6>
            )
        }
        return null;
    }
}
```

The validation API presents validation errors in an awkward way, as explained in Chapter 16. To create messages that can be shown to the user, I added a file called `ValidationMessages.js` in the `src/forms` folder and defined the function shown in Listing 6-30.

Listing 6-30. The Contents of the ValidationMessages.js File in the src/forms Folder

```
export const GetMessages = (elem) => {
    const messages = [];
    if (elem.validity.valueMissing) {
        messages.push("Value required");
    }
    if (elem.validity.typeMismatch) {
        messages.push(`Invalid ${elem.type}`);
    }
    return messages;
}
```

To use the validated form for checking out, I added a file called `Checkout.js` to the `src/shop` folder and defined the component shown in Listing 6-31.

Listing 6-31. The Contents of the Checkout.js File in the src/shop Folder

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

export class Checkout extends Component {

    constructor(props) {
        super(props);
        this.defaultAttrs = { type: "text", required: true };
        this.formModel = [
                { label: "Name"},
                { label: "Email", attrs: { type: "email" }},
                { label: "Address" },
                { label: "City"},
                { label: "Zip/Postal Code", name: "zip"},
                { label: "Country"}]
    }

    handleSubmit = (formData) => {
        const order = { ...formData, products: this.props.cart.map(item =>
            ({  quantity: item.quantity, product_id: item.product.id})) }
        this.props.placeOrder(order);
        this.props.clearCart();
        this.props.history.push("/shop/thanks");
    }

    handleCancel = () => {
        this.props.history.push("/shop/cart");
    }

    render() {
        return <div className="container-fluid">
            <div className="row">
                <div className="col bg-dark text-white">
                    <div className="navbar-brand">SPORTS STORE</div>
                </div>
            </div>
            <div className="row">
                <div className="col m-2">
                    <ValidatedForm formModel={ this.formModel }
                        defaultAttrs={ this.defaultAttrs }
                        submitCallback={ this.handleSubmit }
                        cancelCallback={ this.handleCancel }
                        submitText="Place Order"
                        cancelText="Return to Cart" />
                </div>
            </div>
        </div>
    }
}
```

The Checkout component uses a ValidatedForm to present the user with fields for their name, email, and address. Each form element will be created with the required attribute, and the type attribute of the input element for the email address is set to email. These attributes are used by the HTML5 constraint validation API and will prevent the user from placing an order unless they provide a value for all fields and enter a valid email address into the email field (although it should be noted that only the format of the email address is validated).

The handleSubmit method will be invoked when the user submits valid form data. This method receives the form data and combines it with details of the user's cart before calling the placeOrder and clearCart action creators and then navigating to the /shop/thanks URL.

Creating the Thank You Component

To present the user with confirmation of their order and to complete the checkout process, I added a file called `Thanks.js` to the `src/shop` folder and defined the component shown in Listing 6-32.

Listing 6-32. The Contents of the Thanks.js File in the src/shop Folder

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

export class Thanks extends Component {

    render() {
        return <div>
            <div className="col bg-dark text-white">
                <div className="navbar-brand">SPORTS STORE</div>
            </div>
            <div className="m-2 text-center">
                <h2>Thanks!</h2>
                <p>Thanks for placing your order.</p>
                <p>Your order is #{ this.props.order ? this.props.order.id : 0 }</p>
                <p>We'll ship your goods as soon as possible.</p>
                <Link to="/shop" className="btn btn-primary">
                    Return to Store
                </Link>
            </div>
        </div>
    }
}
```

The Thanks component displays a simple message and includes the value of the `id` property from the `order` object, which it obtains through its `order` prop. This component will be connected to the data store, and the `order` object it contains will have an `id` value that is assigned by the RESTful web service.

Applying the New Components

To add the new components to the application, I altered the routing configuration in the `ShopConnector` component, as shown in Listing 6-33.

Listing 6-33. Adding New Routes in the ShopConnector.js File in the src/shop Folder

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

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

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

export const ShopConnector = connect(mapStateToProps, mapDispatchToProps)(
    class extends Component {
        render() {
            return <Switch>
                <Redirect from="/shop/products/:category"
                    to="/shop/products/:category/1" exact={ true } />
                <Route path={ "/shop/products/:category/:page" }
                    render={ (routeProps) =>
                        <DataGetter { ...this.props } { ...routeProps }>
                            <Shop { ...this.props } { ...routeProps } />
                        </DataGetter>
                    } />
                <Route path="/shop/cart" render={ (routeProps) =>
                        <CartDetails { ...this.props } { ...routeProps } />} />
                <Route path="/shop/checkout" render={ routeProps =>
                    <Checkout { ...this.props } { ...routeProps } /> } />
                <Route path="/shop/thanks" render={ routeProps =>
                    <Thanks { ...this.props } { ...routeProps } /> } />
                <Redirect to="/shop/products/all/1" />
            </Switch>
        }

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

The result allows the user to check out. To test the new features, navigate to `http://localhost:3000`, add one or more products to the cart, and click the Checkout button, which will present the form shown in Figure 6-6. If you click the Place Order button before filling out the form, you will see validation warnings, as shown in the figure.

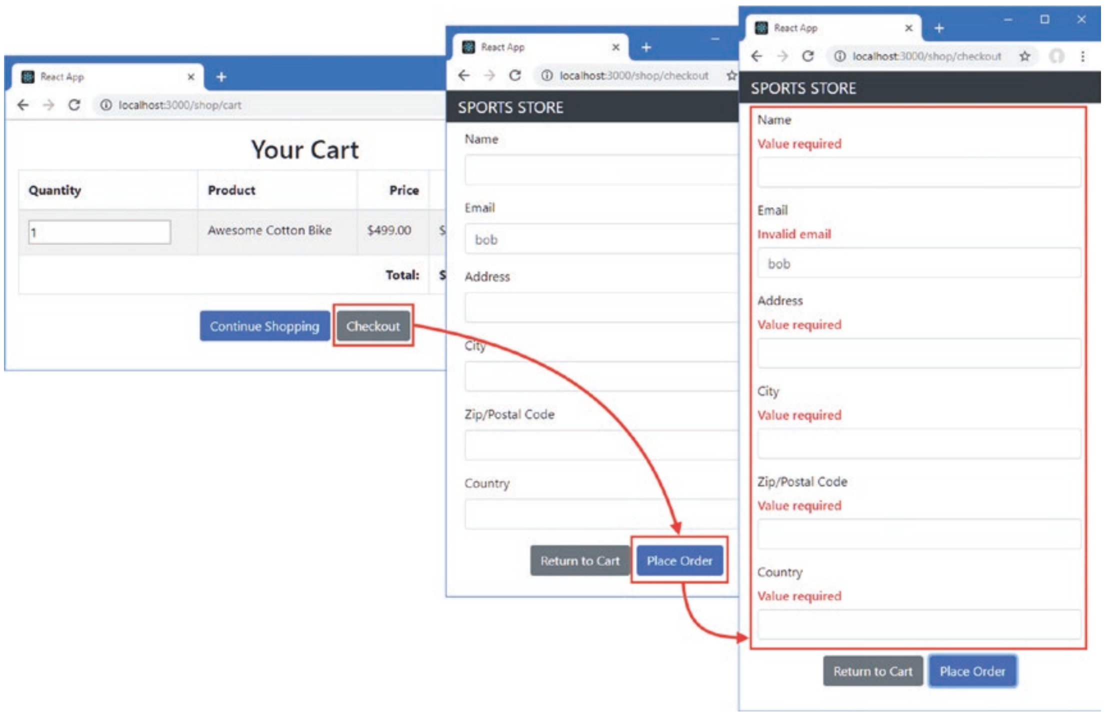
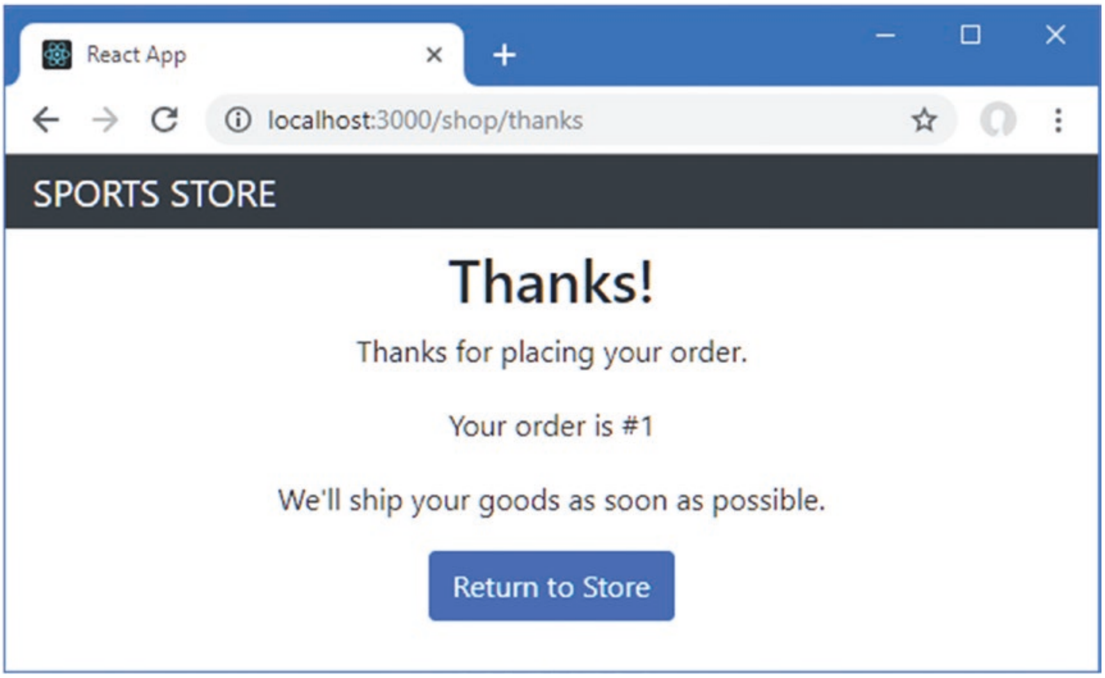


Figure 6-6. *Validation errors when checking out*

■ **Note** Validation is performed only when the user clicks the button. See Chapters 15 and 16 for examples of validating the contents of a form element after each keystroke.

If you have filled all the fields and entered a valid email address, your order will be placed when you click the Place Order button, displaying the summary shown in Figure 6-7.



Thanks!

Thanks for placing your order.

Your order is #1

We'll ship your goods as soon as possible.

Figure 6-7. *Placing an order*

Open a new browser tab and request `http://localhost:3500/api/orders`, and the response will show the JSON representation of the order you place, like this:

```
...
[{
  "name":"Bob Smith","email":"bob@example.com",
  "address":"123 Main Street","city":"New York","zip":"NY 10036",
  "country":"USA","products":[{"quantity":1,"product_id":318}],"id":1
 }]
...
```

Each time you place an order, it will be assigned an `id` by the RESTful web service, which will then be displayed in the order summary.

■ **Tip** The data used by the web service is regenerated each time that the development tools are started with the `npm start` command, which makes it easy to reset the application. In Chapter 8, I switch the SportsStore application to a persistent database as part of the preparations for deployment.

Simplifying the Shop Connector Component

All of the features required by the shopping part of the SportsStore application are complete, but I am going to make one more change in this chapter.

A React application is driven by its props, which provide components with the data and functions they require. When features like URL routing and a data store are used, the point where their capabilities are translated into props can become complex. For the SportsStore application, that is the `ShopConnector` component, which incorporates data store properties, action creators, and URL routing for the shopping part of the application. The advantage of consolidating these features is that the other shopping components are simpler to write, maintain, and test. The disadvantage is that consolidation results in code that is hard to read and where errors are likely to arise.

As I added features to the application, I added a new `Route` that selected a component and provided it with access to props from the data store and the URL router. I could have been more specific about the props each component received, which is the practice I have followed in many of the examples later in the book. For the SportsStore project, however, I gave every component access to all of the props, which is an approach that makes development easier and which allows the routing code to be tidied up once all of the features have been added. In Listing 6-34, I have simplified the connector for the shopping features.

Listing 6-34. Simplifying the Code in the ShopConnector.js File in the src/connectors Folder

```
import React, { Component } from "react";
import { Switch, Route, Redirect }
    from "react-router-dom"
import { connect } from "react-redux";
import * as ShopActions from "../data/ActionCreators";
import { DataTypes } from "../data/Types";
import { Shop } from "../shop/Shop";
import  * as CartActions from "../data/CartActionCreators";
import { CartDetails } from "../shop/CartDetails";
import { DataGetter } from "../data/DataGetter";
import { Checkout } from "../shop/Checkout";
import { Thanks } from "../shop/Thanks";

const mapDispatchToProps = { ...ShopActions, ...CartActions};

export const ShopConnector = connect(ds => ds, mapDispatchToProps)(
    class extends Component {

        selectComponent = (routeProps) => {
            const wrap = (Component, Content) =>
                <Component { ...this.props}  { ...routeProps}>
                    { Content && wrap(Content)}
                </Component>
            switch (routeProps.match.params.section) {
                case "products":
                    return wrap(DataGetter, Shop);
                case "cart":
                    return wrap(CartDetails);
                case "checkout":
                    return wrap(Checkout);
                case "thanks":
                    return wrap(Thanks);
                default:
                    return <Redirect to="/shop/products/all/1" />
            }
        }

        render() {
            return <Switch>
                <Redirect from="/shop/products/:category"
                    to="/shop/products/:category/1" exact={ true } />
                <Route path={ "/shop/:section?/:category?/:page?"}
                    render = { routeProps => this.selectComponent(routeProps) } />
            </Switch>
        }

        componentDidMount = () => this.props.loadData(DataTypes.CATEGORIES);
    }
)
```

In Chapter 9, I explain how JSX is translated into JavaScript, but it is easy to forget that all components can be restructured to rely less on the declarative nature of HTML elements and more on pure JavaScript. In Listing 6-34, I have collapsed the multiple `Route` components into one whose `render` function selects the component that should be displayed to the user and provides it with props from the data store and URL router. I have also changed the `import` statements for the action creators and used the spread operator when mapping them to function props, which I didn't do earlier because I wanted to show how I connected each data store feature to the rest of the application.

Summary

In this chapter, I continued the development of the SportsStore folder, adding support for working with the RESTful web server, scaling up the amount of data that the application can deal with, and adding support for checking out and placing orders. In the next chapter, I add the administration features to the SportsStore application.