CHAPTER 23

Consuming a RESTful Web Service

In this chapter, I address the example application's lack of permanent data storage by creating a web service and using it to manage the application's data. The application will send HTTP requests to the web service to retrieve data and to submit changes. I start this chapter by showing you how to consume a web service directly in a component and then demonstrate how a web service can be used with a data store. In Chapter 24, I explain how to use GraphQL, which is an alternative approach to dealing with web services. Table 23-1 puts this chapter in context.

Table 23-1. *Putting Consuming Web Services in Context*

Question	Answer
What is it?	Web services act as the data repository for an application, allowing data to be read, stored, modified, and deleted using HTTP requests.
Why is it useful?	Web services fit neatly into the features available in browsers and avoid having to deal with local storage issues.
How is it used?	Web services are not all implemented the same way, but the general approach is to send HTTP requests where the request method identifies the operation to be performed and the request URL identifies the data to be operated on.
Are there any pitfalls or limitations?	The inconsistent nature of web service implementations means that each web service can require a slightly different set of requests. Care must be taken when consuming the web service in a component to ensure that requests are not sent each time there is an update.
Are there any alternatives?	Modern web browsers support local storage options, which can be a good alternative for some projects. The main drawback, however, is that each client has its own data, which misses out on some of the advantages of a single central repository.

Table 23-2 summarizes the chapter.

Table 23-2. *Chapter Summary*

Problem	Solution	Listing
Get data from a web service	Create a data source that makes HTTP requests and feed the data back into the application using a callback that invokes the `setState` method.	1–11
Perform additional data operations	Extend the data source to send different combinations of HTTP methods and URLs to indicate the required operation. Trigger the requests by responding to component events	12–15
Handle request errors	Use a `try/catch` block to catch the error and pass it to a component so that a warning can be displayed to the user.	16–19
Consume a web service with a data store	Use middleware to intercept the data store actions and send the required requests to the web service. Once a request has completed, forward the action to the data store so that it can be updated.	20–24

Preparing for This Chapter

In this chapter, I continue using the `productapp` project from Chapter 22 that was modified in the chapters since. Some preparation is required to install additional packages to the project and create the web service that the application will rely on.

■ **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`.

Adding Packages to the Project

Run the commands shown in Listing 23-1 in the `productapp` folder to add the required packages to the project.

Listing 23-1. Installing Additional Packages to the Project

```
npm install json-server@0.14.0 --save-dev
npm install npm-run-all@4.1.3 --save-dev
npm install axios@0.18.0
```

For quick reference, the packages by the commands in Listing 23-1 are described in Table 23-3.

Table 23-3. *The Packages Added to the Project*

Name	Description
`json-server`	This package provides a web service that the application will query for data. This command is installed with the `save-dev` command because it is required for development and is not part of the application.
`npm-run-all`	This package allows multiple commands to be run in parallel so that the web service and the development server can be started at the same time. This command is installed with the `save-dev` command because it is required for development and is not part of the application.
`axios`	This package will be used by the application to make HTTP requests to the web service.

Preparing the Web Service

To provide the `json-server` package with data to work with, add a file called `restData.js` to the `productapp` folder and add the code shown in Listing 23-2.

Listing 23-2. The Contents of the restData.js File in the productapp Folder

```
module.exports = function () {
    var data = {
        products: [
            { id: 1, name: "Kayak", category: "Watersports", price: 275 },
            { id: 2, name: "Lifejacket", category: "Watersports", price: 48.95 },
            { id: 3, name: "Soccer Ball", category: "Soccer", price: 19.50 },
            { id: 4, name: "Corner Flags", category: "Soccer", price: 34.95 },
            { id: 5, name: "Stadium", category: "Soccer", price: 79500 },
            { id: 6, name: "Thinking Cap", category: "Chess", price: 16 },
            { id: 7, name: "Unsteady Chair", category: "Chess", price: 29.95 },
            { id: 8, name: "Human Chess Board", category: "Chess", price: 75 },
            { id: 9, name: "Bling Bling King", category: "Chess", price: 1200 }
        ],
        suppliers: [
            { id: 1, name: "Surf Dudes", city: "San Jose", products: [1, 2] },
            { id: 2, name: "Goal Oriented", city: "Seattle", products: [3, 4, 5] },
            { id: 3, name: "Bored Games", city: "New York", products: [6, 7, 8, 9] },
        ]
    }
    return data
}
```

The `json-server` package can work with JSON or JavaScript files. If a JSON file is used, its contents will be modified to reflect changes requests made by clients. Instead, I have chosen the JavaScript option, which allows data to be generated programmatically and means that restarting the process will return to the original data. This isn't something that you would do in a real project, but it is useful for the example because it makes it easy to return to a known state, while still allowing the application access to persistent data.

To configure the `json-server` package so that it responds to requests for URLs that start with `/api`, create a file called `api.routes.json` in the `productapp` folder with the contents shown in Listing 23-3.

Listing 23-3. The Contents of the api.routes.json File in the productapp Folder

```
{ "/api/*": "/$1" }
```

To configure the development tools so that the web service is started at the same time as the development web server, make the changes shown in Listing 23-4 to the `package.json` file in the `productapp` folder.

Listing 23-4. Configuring Tools in the package.json File in the productapp Folder

```
...
"scripts": {
    "start": "npm-run-all --parallel reactstart json",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject",
    "reactstart": "react-scripts start",
    "json": "json-server --p 3500 -r api.routes.json restData.js"
},
...
```

The changes to the `scripts` section of the `package.json` file use the `npm-run-all` package so that the HTTP development server and `json-server` are started by `npm start`.

Adding a Component and a Route

I am going to demonstrate how to consume a web service in isolation and then show you how to use the data in a data store. The existing components in the application are already connected to the data store, and so to show how unconnected components can be used, I created a file called `IsolatedTable.js` in the `src` folder and used it to create the component shown in Listing 23-5.

Listing 23-5. The Contents of the IsolatedTable.js File in the src Folder

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

export class IsolatedTable extends Component {

    render() {
        return <table className="table table-sm table-striped table-bordered">
            <thead>
                <tr><th colSpan="5"
                        className="bg-info text-white text-center h4 p-2">
                    (Isolated) Products
                </th></tr>
                <tr>
                    <th>ID</th><th>Name</th><th>Category</th>
                    <th className="text-right">Price</th>
                    <th></th>
                </tr>
            </thead>
            <tbody>
                <tr><td colSpan="5" className="text-center p-2">No Data</td></tr>
            </tbody>
        </table>
    }
}
```

The component renders an empty table as a placeholder for the moment. To incorporate the component into the application, I updated the routing configuration in the Selector component to add a new Route and a corresponding navigation link, as shown in Listing 23-6.

Listing 23-6. Adding a Route in the Selector.js File in the src Folder

```
import React, { Component } from "react";
import { BrowserRouter as Router, Route, Switch, Redirect }
    from "react-router-dom";
import { ToggleLink } from "./routing/ToggleLink";
import { RoutedDisplay } from "./routing/RoutedDisplay";
import { IsolatedTable } from "./IsolatedTable";

export class Selector extends Component {

    render() {

        const routes = React.Children.map(this.props.children, child => ({
            component: child,
            name: child.props.name,
            url: `/${child.props.name.toLowerCase()}`,
            datatype: child.props.datatype
        }));

        return <Router getUserConfirmation={ this.customGetUserConfirmation }>
            <div className="container-fluid">
                <div className="row">
                    <div className="col-2">
                        <ToggleLink to="/isolated">Isolated Data</ToggleLink>
                        { routes.map(r => <ToggleLink key={ r.url } to={ r.url }>
                                            { r.name }
                                        </ToggleLink>)}
                    </div>
                    <div className="col">
                        <Switch>
                            <Route path="/isolated" component={ IsolatedTable } />
                            { routes.map(r =>
                               <Route key={ r.url }
                                   path={ `/:datatype(${r.datatype})/:mode?/:id?`}
                                   component={ RoutedDisplay(r.datatype)} />
                            )}
                            <Redirect to={ routes[0].url } />
                        </Switch>
                    </div>
                </div>
            </div>
        </Router>
    }
}
```

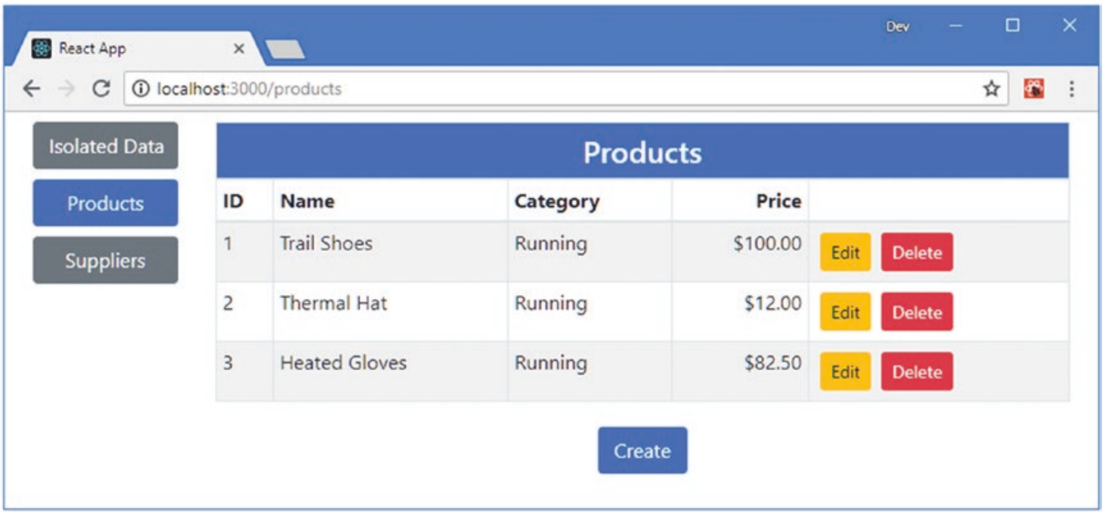
Running the Web Service and the Example Application

Using the command prompt, run the command shown in Listing 23-7 in the `productapp` folder to start the development tools and the web service.

Listing 23-7. Starting the Development Tools

```
npm start
```

Once the initial preparation for the project is complete, a new browser window will open and display the URL `http://localhost:3000`, as shown in Figure 23-1.

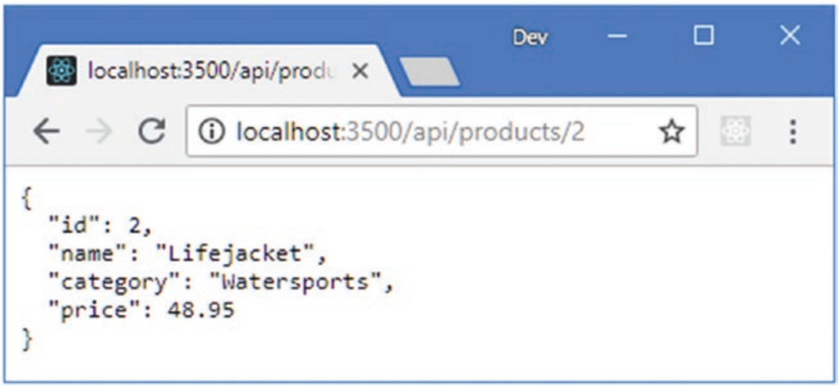


ID	Name	Category	Price	
1	Trail Shoes	Running	$100.00	Edit Delete
2	Thermal Hat	Running	$12.00	Edit Delete
3	Heated Gloves	Running	$82.50	Edit Delete

Figure 23-1. *Running the example application*

Open a new browser window and navigate to `http://localhost:3500/api/products/2`. The server will respond with the following data, which is also shown in Figure 23-2:

```
...
{ "id": 2, "name": "Lifejacket", "category": "Watersports", "price": 48.95 }
...
```



```
{
  "id": 2,
  "name": "Lifejacket",
  "category": "Watersports",
  "price": 48.95
}
```

Figure 23-2. *Testing the web service*

The configuration I have chosen for this chapter means that there are two HTTP servers running. The React development server is listening for requests on port 3000 and provides the HTML document that bootstraps the application, along with the JavaScript and CSS files required to present the application to the user. The RESTful web service is listening for requests on port 3500 and responds with data. This data is expressed in the JSON format, which means it is easily processed by a JavaScript application but should not be presented directly to most users.

Understanding RESTful Web Services

The most common approach for delivering and storing application data is applying the *Representational State Transfer* pattern, known as REST, to create a data web service. There is no detailed specification for REST, which leads to a lot of different approaches that fall under the RESTful banner. There are, however, some unifying ideas that are useful in web application development.

The core premise of a RESTful web service is to embrace the characteristics of HTTP so that request methods—also known as *verbs*—specify an operation for the server to perform, and the request URL specifies one or more data objects to which the operation will be applied.

As an example, here is a URL that might refer to a specific product in the example application:

```
http://localhost:3500/api/products/2
```

The first segment of the URL—`api`—conventionally indicates that the request is for data. The next segment—`products`—is used to indicate the collection of objects that will be operated on and allows a single server to provide multiple services, each of which with its own data. The final segment—`2`—selects an individual object within the `products` collection. In the example, it is the value of the `id` property that uniquely identifies an object and that would be used in the URL, in this case, specifying the `Lifejacket` object.

The HTTP verb or method used to make the request tells the RESTful server what operation should be performed on the specified object. When you tested the RESTful server in the previous section, the browser sent an HTTP GET request, which the server interprets as an instruction to retrieve the specified object and send it to the client.

Table 23-4 shows the most common combination of HTTP methods and URLs and explains what each of them does when sent to a RESTful server.

***Table 23-4.** Common HTTP Verbs and Their Effect in a RESTful Web Service*

Verb	URL	Description
`GET`	`/api/products`	This combination retrieves all the objects in the `products` collection.
`GET`	`/api/products/2`	This combination retrieves the object whose `id` is `2` from the `products` collection.
`POST`	`/api/products`	This combination is used to add a new object to the `products` collection. The request body contains a JSON representation of the new object.
`PUT`	`/api/products/2`	This combination is used to replace the object in the `products` collection whose `id` is 2. The request body contains a JSON representation of the replacement object.
`PATCH`	`/api/products/2`	This combination is used to update a subset of the properties of the object in the `products` collection whose `id` is 2. The request body contains a JSON representation of the properties to update and the new values.
`DELETE`	`/api/products/2`	This combination is used to delete the product whose `id` is 2 from the `products` collection.

There are considerable differences in the way that web services are implemented, caused by differences in the frameworks used to create them and the preferences of the development team. It is important to confirm how a web service uses verbs and what is required in the URL and request body to perform operations.

Common variations include web services that won't accept any request bodies that contain `id` values (to ensure they are generated uniquely by the server's data store) and web services that don't support all of the verbs (it is common to ignore `PATCH` requests and only accept updates using the `PUT` verb).

■ **Tip** You may have noticed that the editor components don't allow the user to provide a value for the `id` property. This is because the web service that I create in this chapter generates `id` values automatically to ensure uniqueness.

CHOOSING AN HTTP REQUEST LIBRARY

Throughout this chapter, I use the Axios library to send HTTP requests to the web service because it is easy to use, deals with common data types automatically, and doesn't require convoluted code to deal with features like CORS (see the "Making Cross-Origin Requests" sidebar). Axios is widely used in web application development, although it is not specific to React.

Axios isn't the only way to send HTTP requests to web services. The most basic option to use the `XMLlHttpRequest` object that provided the original API for making requests using JavaScript (and which is capable of handling a range of data types, despite the XML in the name). The `XMLHttpRequest` object is awkward to use but has wide browser support, and you can get further details at https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest. (Axios uses XMLHttpRequest to make HTTP requests but simplifies how they are created and processed.)

The Fetch API is a recent API provided by modern browsers that is intended to replace XMLHttpRequest and is described at https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API. The Fetch API is supported by recent releases of the mainstream browsers but not by older browsers, which can be a problem for some applications.

If you are using GraphQL, then you should consider using the Apollo client, as described in Chapter 25.

Consuming a Web Service

I go through the steps required to consume a web service in the sections that follow, beginning with requesting the initial data that the application will display to the user and then adding support for storing and updating objects.

Creating the Data Source Component

It is a good idea to keep the code that uses Axios to consume the web service separate from the component that uses it so that it can be more easily tested and used elsewhere in the application. I created the src/webservice folder and added to it a file called RestDataSource.js with the code shown in Listing 23-8.

Listing 23-8. The Contents of the RestDataSource.js File in the src/webservice Folder

```
import Axios from "axios";

export class RestDataSource {

    constructor(base_url) {
        this.BASE_URL = base_url;
    }

    GetData(callback) {
        this.SendRequest("get", this.BASE_URL, callback);
    }

    SendRequest(method, url, callback) {
        Axios.request({
            method: method,
            url: url
        }).then(response => callback(response.data));
    }
}
```

The RestDataSource class defines a constructor that receives the base URL for the web service and defines a GetData method that calls the SendRequest.

I imported the HTTP functionality from the axios package and assigned it the name Axios. The SendRequest method uses Axios to send an HTTP request through the request method, where the details of the request are specified using a configuration object that has method and url properties.

Axios provides methods for sending different types of HTTP request—the get, post, and put methods, for example, but using the approach in the listing makes it easier to apply features that affect all request types, as you will see when I add error handling later in the chapter.

HTTP requests made using JavaScript are asynchronous. The request method returns a Promise object that represents the eventual outcome of the request (see Chapter 4 for details of how Promise objects are used). In Listing 23-8, I use the then method to supply Axios with a callback function to use when the request is complete. The callback function is passed an object that describes the response using the properties described in Table 23-5.

Table 23-5. *The Axios Response Properties*

Name	Description
status	This property returns the status code for the response, such as 200 or 404.
statusText	This property returns the explanatory text that accompanies the status code, such as OK or Not Found.
headers	This property returns an object whose properties represent the response headers.
data	This property returns the payload from the response.
config	This property returns an object that contains the configuration options used to make the request.
request	This property returns the underlying XMLHttpRequest object that was used to make the request, which can be useful if you require direct access to the API provided by the browser.

Axios automatically converts the JSON data format into a JavaScript object and presents it through the response data property. As explained in Chapter 4, code that uses promises can be simplified using the async and await keywords, as shown in Listing 23-9.

Listing 23-9. Using async and await in the RestDataSource.js File in the src/webservice Folder

```
import Axios from "axios";

export class RestDataSource {

    constructor(base_url) {
        this.BASE_URL = base_url;
    }

    GetData(callback) {
        this.SendRequest("get", this.BASE_URL, callback);
    }

    async SendRequest(method, url, callback) {
        let response = await Axios.request({
            method: method,
            url: url
        });
        callback(response.data);
    }
}
```

I can further simplify the code by combining the statements in the GetData method, as shown in Listing 23-10.

Listing 23-10. Combining Statements in the RestDataSource.js File in the src/webservice Folder

```
import Axios from "axios";

export class RestDataSource {

    constructor(base_url) {
        this.BASE_URL = base_url;
    }

    GetData(callback) {
        this.SendRequest("get", this.BASE_URL, callback);
    }

    async SendRequest(method, url, callback) {
        callback((await Axios.request({
            method: method,
            url: url
        })).data);
    }
}
```

This approach is more concise, but it is important to make sure you put the parentheses in the right places, such that the await keyword is applied to the object returned by the SendRequest method and the data property is read from the object that it produces. Without care, you can easily create a situation where the HTTP request is sent but the response is ignored if you don't follow this pattern.

Getting Data in the Component

The next step is to get the data into the component so that it can be displayed to the user. In Listing 23-11, I have updated the IsolatedTable component so that it creates a data source and uses it to request data from the web service.

■ **Note** The term *isolated* in the name of the component indicates that the component doesn't share data with any other components and deals directly with the web service. In the "Consuming a Web Service with a Data Store" section, I show you an alternative approach where components share data via the data store.

Listing 23-11. Getting Data in the IsolatedTable.js File in the src Folder

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

export class IsolatedTable extends Component {

    constructor(props) {
        super(props);
        this.state = {
            products: []
        }
        this.dataSource = new RestDataSource("http://localhost:3500/api/products")
    }

    render() {
        return <table className="table table-sm table-striped table-bordered">
            <thead>
                <tr><th colSpan="5"
                        className="bg-info text-white text-center h4 p-2">
                    (Isolated) Products
                </th></tr>
                <tr>
                    <th>ID</th><th>Name</th><th>Category</th>
                    <th className="text-right">Price</th>
                    <th></th>
                </tr>
            </thead>
            <tbody>
                {
                    this.state.products.map(p => <tr key={ p.id }>
                        <td>{ p.id }</td><td>{ p.name }</td><td>{p.category}</td>
                        <td className="text-right">
                            ${ Number(p.price).toFixed(2)}
                        </td><td/>
                    </tr>)
                }
            </tbody>
        </table>
    }

    componentDidMount() {
        this.dataSource.GetData(data => this.setState({products: data}));
    }
}
```

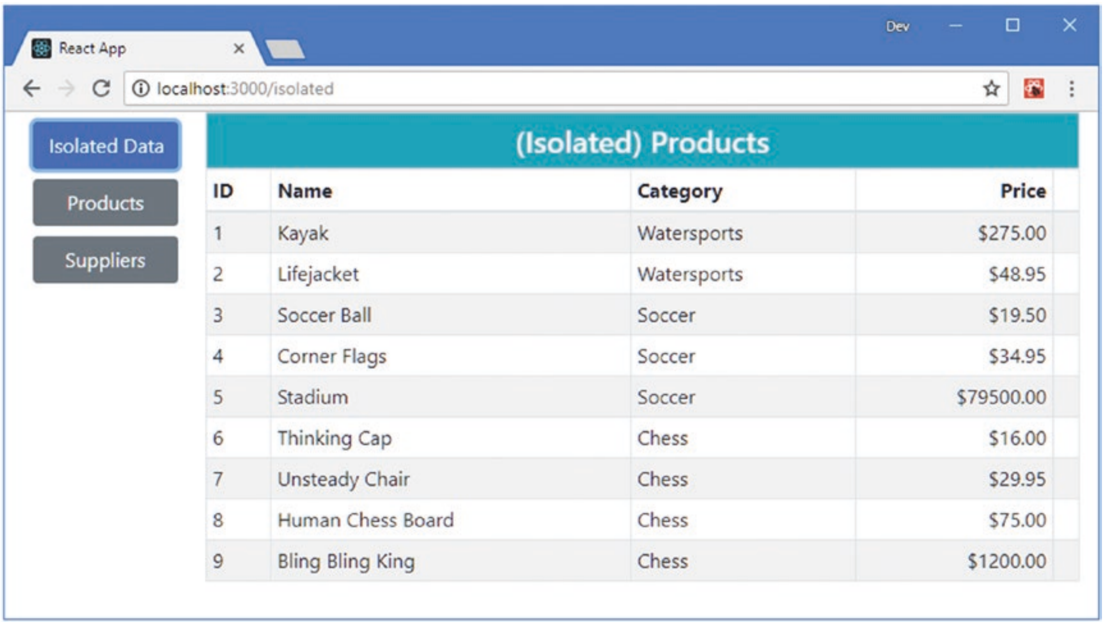
The data is requested in the `componentDidMount` method, which ensures that the HTTP request won't be sent until after the component has rendered its content. The callback functions provided to the `GetData` method update the component's state data, which will trigger an update and ensure that the data is presented to the user.

AVOIDING EXTRANEOUS DATA REQUESTS

Do not request data in the `render` method. As I explained in Chapter 13, a component's `render` method can be called often, and starting tasks in the `render` method can generate large numbers of unnecessary HTTP requests and increase the number of updates that React has to perform as it processes the response data.

Even when using the `componentDidMount` method, care should be taken when making requests from components that may be unmounted and remounted, which is the case for the `IsolatedTable` component in the example, which will be mounted by the routing system for the `/isolated` URL and unmounted when the user navigates to another location. Each time the component is mounted, it will request fresh data from the web service, which may not be what the application requires. To avoid unnecessary data requests, the data can be lifted up to a component that won't be unmounted, stored in a context (as described in Chapter 14), or incorporated into a data store, as described in the "Consuming a Web Service with a Data Store" section.

The result is that the data is obtained from the web service and displayed to the user when the Isolated Data button is clicked, as shown in Figure 23-3.



(Isolated) Products

ID	Name	Category	Price
1	Kayak	Watersports	$275.00
2	Lifejacket	Watersports	$48.95
3	Soccer Ball	Soccer	$19.50
4	Corner Flags	Soccer	$34.95
5	Stadium	Soccer	$79500.00
6	Thinking Cap	Chess	$16.00
7	Unsteady Chair	Chess	$29.95
8	Human Chess Board	Chess	$75.00
9	Bling Bling King	Chess	$1200.00

Figure 23-3. *Getting data from the web service*

Saving, Updating, and Deleting Data

To implement the operations required for saving, updating, and deleting data, I added the methods shown in Listing 23-12 to the data source class, using Axios to send requests to the web service with different HTTP methods.

Listing 23-12. Adding Methods in the RestDataSource.js File in the src/webservice Folder

```
import Axios from "axios";

export class RestDataSource {

    constructor(base_url) {
        this.BASE_URL = base_url;
    }

    GetData(callback) {
        this.SendRequest("get", this.BASE_URL, callback);
    }

    async GetOne(id, callback) {
        this.SendRequest("get", `${this.BASE_URL}/${id}`, callback);
    }

    async Store(data, callback) {
        this.SendRequest("post", this.BASE_URL, callback, data)
    }

    async Update(data, callback) {
        this.SendRequest("put", `${this.BASE_URL}/${data.id}`, callback, data);
    }

    async Delete(data, callback) {
        this.SendRequest("delete", `${this.BASE_URL}/${data.id}`, callback, data);
    }

    async SendRequest(method, url, callback, data) {
        callback((await Axios.request({
            method: method,
            url: url,
            data: data
        })).data);
    }
}
```

The request configuration object passed to the `Axios.request` method uses a `data` property to specify the payload for the request, which allows the application to provide JavaScript objects and leave Axios to serialize them automatically.

When you implement data source methods, you will find that some adjustment is required to accommodate the range of ways that web services can be implemented. For example, the example web service will automatically assign a unique `id` property value to objects that are received in POST requests and include the complete object in the response. The `Store` method in Listing 23-12 uses the `data` property to get the complete object from the HTTP response and uses it to invoke the callback, which ensures that the application receives the object as it has been stored by the web service. Not all web services operate this way—some may require the application to include a unique identifier or will return only the identifier in the response instead of sending the complete object.

When modifying an object, a PUT request is sent with the URL identifying the object to be modified, like this:

```
...
this.SendRequest("put", `${this.BASE_URL}/${data.id}`, callback, data);
...
```

The web service returns the complete updated object, which is used to invoke the callback function. Once again, not all web services will return the complete object, but it is a common approach because it ensures that any additional transformations that are applied by the web service are reflected in the client.

Adding Application Support for Creating, Editing, and Deleting Data

To provide support for creating and editing data, I added a file called `IsolatedEditor.js` to the `src` folder and used it to define the component shown in Listing 23-13.

Listing 23-13. The Contents of the IsolatedEditor.js File in the src Folder

```
import React, { Component } from "react";
import { RestDataSource } from "./webservice/RestDataSource";
import { ProductEditor } from "./ProductEditor";

export class IsolatedEditor extends Component {

    constructor(props) {
        super(props);
        this.state = {
            dataItem: {}
        };
        this.dataSource = this.props.dataSource
            || new RestDataSource("http://localhost:3500/api/products");
    }

    save = (data) => {
        const callback = () => this.props.history.push("/isolated");
        if (data.id === "") {
            this.dataSource.Store(data, callback);
        } else {
            this.dataSource.Update(data, callback);
        }
    }

    cancel = () => this.props.history.push("/isolated");

    render() {
        return <ProductEditor key={ this.state.dataItem.id }
            product={ this.state.dataItem } saveCallback={ this.save }
            cancelCallback={ this.cancel } />
    }
    componentDidMount() {
        if (this.props.match.params.mode === "edit") {
            this.dataSource.GetOne(this.props.match.params.id,
                data => this.setState({ dataItem: data}));
        }
    }
}
```

React makes it easy to use existing components in new ways, and the `IsolatedEditor` component uses the existing `ProductEditor` and its props to provide it with data and callbacks from the web service data source. Details of the current route are used to request details of a single object using the `GetOne` method when the user has selected an object for editing, and changes are sent back to the web service using the `Store` or `Update` methods. In Listing 23-14, I have added support to the `IsolatedTable` component for creating and editing objects by navigating to new URLs. I have also added a Delete button whose event handler invokes the data source's `Delete` method, which sends a DELETE request to the web service.

Listing 23-14. Adding Data Operations in the IsolatedTable.js File in the src Folder

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

export class IsolatedTable extends Component {

    constructor(props) {
        super(props);
        this.state = {
            products: []
        }
        this.dataSource = new RestDataSource("http://localhost:3500/api/products")
    }

    deleteProduct(product) {
        this.dataSource.Delete(product,
            () => this.setState({products: this.state.products.filter(p =>
                p.id !== product.id)}));
    }

    render() {
        return <table className="table table-sm table-striped table-bordered">
            <thead>
                <tr><th colSpan="5"
                        className="bg-info text-white text-center h4 p-2">
                    (Isolated) Products
                </th></tr>
                <tr>
                    <th>ID</th><th>Name</th><th>Category</th>
                    <th className="text-right">Price</th>
                    <th></th>
                </tr>
            </thead>
            <tbody>
                {
                    this.state.products.map(p => <tr key={ p.id }>
                        <td>{ p.id }</td><td>{ p.name }</td><td>{p.category}</td>
                        <td className="text-right">
                            ${ Number(p.price).toFixed(2)}
                        </td>
                        <td>
                            <Link className="btn btn-sm btn-warning mx-2"
                                    to={`/isolated/edit/${p.id}`}>
                                Edit
                            </Link>
                            <button className="btn btn-sm btn-danger mx-2"
                                onClick={ () => this.deleteProduct(p)}>
                                    Delete
                            </button>
                        </td>
                    </tr>)
                }
            </tbody>
            <tfoot>
                <tr className="text-center">
                    <td colSpan="5">
                        <Link to="/isolated/create"
                            className="btn btn-info">Create</Link>
                    </td>
                </tr>
            </tfoot>
        </table>
    }

    componentDidMount() {
        this.dataSource.GetData(data => this.setState({products: data}));
    }
}
```

The final step is to update the routing configuration in the Selector component so that the /isolated/edit and /isolated/create URLs select the IsolatedEditor component. I have also set the route for the /isolated URL to match exactly to ensure that the Route for the IsolatedTable component doesn't match the other URLs, as shown in Listing 23-15.

Listing 23-15. Adding a Route in the Selector.js File in the src Folder

```
import React, { Component } from "react";
import { BrowserRouter as Router, Route, Switch, Redirect }
    from "react-router-dom";
import { ToggleLink } from "./routing/ToggleLink";
import { RoutedDisplay } from "./routing/RoutedDisplay";
import { IsolatedTable } from "./IsolatedTable";
import { IsolatedEditor } from "./IsolatedEditor";
```

```
export class Selector extends Component {

    render() {

        const routes = React.Children.map(this.props.children, child => ({
            component: child,
            name: child.props.name,
            url: `/${child.props.name.toLowerCase()}`,
            datatype: child.props.datatype
        }));

        return <Router getUserConfirmation={ this.customGetUserConfirmation }>
            <div className="container-fluid">
                <div className="row">
                    <div className="col-2">
                        <ToggleLink to="/isolated">Isolated Data</ToggleLink>
                        { routes.map(r => <ToggleLink key={ r.url } to={ r.url }>
                                            { r.name }
                                        </ToggleLink>)}
                    </div>
                    <div className="col">
                        <Switch>
                            <Route path="/isolated" component={ IsolatedTable }
                                exact={ true } />
                            <Route path="/isolated/:mode/:id?"
                                component={ IsolatedEditor } />
                            { routes.map(r =>
                               <Route key={ r.url }
                                   path={ `/:datatype(${r.datatype})/:mode?/:id?`}
                                   component={ RoutedDisplay(r.datatype)} />
                            )}
                            <Redirect to={ routes[0].url } />
                        </Switch>
                    </div>
                </div>
            </div>
        </Router>
    }
}
```

The IsolatedTable component displays Create, Edit, and Delete buttons, as shown in Figure 23-4. The Create and Edit buttons present the editor component to the user, which then updates the web service with the changes that the user makes by sending POST or PUT requests. The Delete buttons remove the object with which they are associated by sending a DELETE request to the web service.

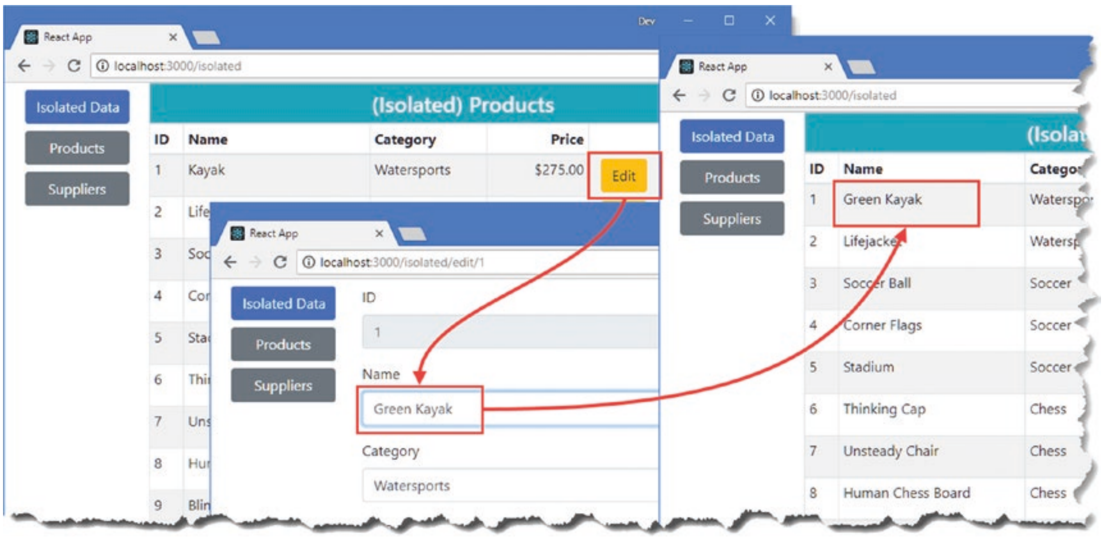


Figure 23-4. *Consuming a web service*

■ **Note** The changes made by the application are stored in the web service, which means you can reload the browser and the changes will still be visible. The configuration of the `json-server` package at the start of the chapter means that restarting the development tools will reset the data presented to by the web service. See the SportsStore application in Chapter 8 for an example of using `json-server` for truly persistent data that does not reset when the tools are restarted.

Dealing with Errors

The application assumes that all the HTTP requests will succeed, which is an unrealistically optimistic approach. There are lots of reasons why an HTTP request may fail, such as connectivity issues or server failure. Error boundaries, which I described in Chapter 14, can't deal with problems that arise in asynchronous operations such as HTTP requests, so a different approach is required. In Listing 23-16, I have changed the data source so that it receives a function that it will invoke when there is a problem and use the `try/catch` keywords to invoke the function when a request fails.

Listing 23-16. Handling Errors in the RestDataSource.js File in the src/webservice Folder

```
import Axios from "axios";

export class RestDataSource {

    constructor(base_url, errorCallback) {
        this.BASE_URL = base_url;
        this.handleError = errorCallback;
    }
    GetData(callback) {
        this.SendRequest("get", this.BASE_URL, callback);
    }

    async GetOne(id, callback) {
        this.SendRequest("get", `${this.BASE_URL}/${id}`, callback);
    }

    async Store(data, callback) {
        this.SendRequest("post", this.BASE_URL, callback, data)
    }

    async Update(data, callback) {
        this.SendRequest("put", `${this.BASE_URL}/${data.id}`, callback, data);
    }

    async Delete(data, callback) {
        this.SendRequest("delete", `${this.BASE_URL}/${data.id}`, callback, data);
    }

    async SendRequest(method, url, callback, data) {
        try {
            callback((await Axios.request({
                method: method,
                url: url,
                data: data
            })).data);
        } catch(err) {
            this.handleError("Operation Failed: Network Error");
        }
    }
}
```

The advantage of consolidating all the requests through the SendRequest method is that I can use a single try/catch block to handle errors for all request types. The catch block handles errors that arise from requests and invokes the callback function that is received as a constructor argument.

PRESENTING ERROR MESSAGES TO THE USER

The Axios package presents detailed errors when something goes wrong and includes the status code from the response and any descriptive text the web service supplies. For most applications, however, it doesn't make sense to present this information to the user, who won't understand what has happened or know how to fix it. Instead, I recommend presenting a general error message to the user and logging details of the problem at the server so that common issues can be identified.

To receive errors and display them to the user, I added a file called RequestError.js to the src/webservice folder and used it to define the component shown in Listing 23-17.

Listing 23-17. The Contents of the RequestError.js File in the src/webservice Folder

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

export class RequestError extends Component {

    render() {
        return <div>
            <h5 className="bg-danger text-center text-white m-2 p-3">
                { this.props.match.params.message }
            </h5>
            <div className="text-center">
                <Link to="/" className="btn btn-secondary">OK</Link>
            </div>
        </div>
    }
}
```

This component displays a message obtained from a URL parameter. Listing 23-18 adds a new Route to the Selector component that will display this component for the /error URL.

Listing 23-18. Adding a Route in the Selector.js File in the src Folder

```
import React, { Component } from "react";
import { BrowserRouter as Router, Route, Switch, Redirect }
    from "react-router-dom";
import { ToggleLink } from "./routing/ToggleLink";
import { RoutedDisplay } from "./routing/RoutedDisplay";
import { IsolatedTable } from "./IsolatedTable";
import { IsolatedEditor } from "./IsolatedEditor";
import { RequestError } from "./webservice/RequestError";

export class Selector extends Component {

    render() {

        const routes = React.Children.map(this.props.children, child => ({
            component: child,
            name: child.props.name,
            url: `/${child.props.name.toLowerCase()}`,
            datatype: child.props.datatype
        }));

        return <Router getUserConfirmation={ this.customGetUserConfirmation }>
            <div className="container-fluid">
                <div className="row">
                    <div className="col-2">
                        <ToggleLink to="/isolated">Isolated Data</ToggleLink>
                        { routes.map(r => <ToggleLink key={ r.url } to={ r.url }>
                                               { r.name }
                                           </ToggleLink>)}
                    </div>
                    <div className="col">
                        <Switch>
                            <Route path="/isolated" component={ IsolatedTable }
                                exact={ true } />
                            <Route path="/isolated/:mode/:id?"
                                component={ IsolatedEditor } />
                            <Route path="/error/:message"
                                component={ RequestError } />
                            { routes.map(r =>
                               <Route key={ r.url }
                                   path={ `/:datatype(${r.datatype})/:mode?/:id?`}
                                   component={ RoutedDisplay(r.datatype)} />
                            )}
                            <Redirect to={ routes[0].url } />
                        </Switch>
                    </div>
                </div>
            </div>
        </Router>
    }
}
```

Listing 23-19 provides the data source with a callback that navigates to the `/error` URL when a problem arises and adds a button that creates an error by requesting a URL that will always produce a 404 – Not Found error.

Listing 23-19. Handling Errors in the IsolatedTable.js File in the src Folder

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

export class IsolatedTable extends Component {

    constructor(props) {
        super(props);
        this.state = {
            products: []
        }
        this.dataSource = new RestDataSource("http://localhost:3500/api/products",
            (err) => this.props.history.push(`/error/${err}`));
    }

    deleteProduct(product) {
        this.dataSource.Delete(product,
           () => this.setState({products: this.state.products.filter(p =>
                p.id !== product.id)}));
    }
    render() {
        return <table className="table table-sm table-striped table-bordered">
            <thead>
                <tr><th colSpan="5"
                        className="bg-info text-white text-center h4 p-2">
                    (Isolated) Products
                </th></tr>
                <tr>
                    <th>ID</th><th>Name</th><th>Category</th>
                    <th className="text-right">Price</th>
                    <th></th>
                </tr>
            </thead>
            <tbody>
                {
                    this.state.products.map(p => <tr key={ p.id }>
                        <td>{ p.id }</td><td>{ p.name }</td><td>{p.category}</td>
                        <td className="text-right">
                            ${ Number(p.price).toFixed(2)}
                        </td>
                        <td>
                            <Link className="btn btn-sm btn-warning mx-2"
                                    to={`/isolated/edit/${p.id}`}>
                                Edit
                            </Link>
                            <button className="btn btn-sm btn-danger mx-2"
                                onClick={ () => this.deleteProduct(p)}>
                                    Delete
                            </button>
                        </td>
                    </tr>)
                }
            </tbody>
            <tfoot>
                <tr className="text-center">
                    <td colSpan="5">
                        <Link to="/isolated/create"
                            className="btn btn-info">Create</Link>
                        <button className="btn btn-danger mx-2"
                            onClick={ () => this.dataSource.GetOne("err")}>
                            Error
                        </button>
                    </td>
                </tr>
            </tfoot>
        </table>
    }

    componentDidMount() {
        this.dataSource.GetData(data => this.setState({products: data}));
    }
}
```

Clicking the Error button rendered by `IsolatedTable` will send a request that receives an error response from the web service, which triggers navigation to the URL that displays the error message, as shown in Figure 23-5.

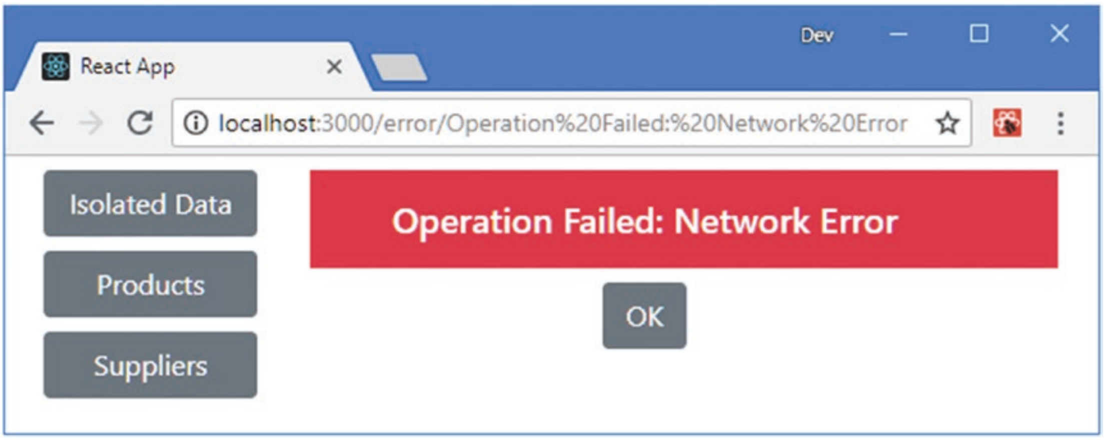

React App
Dev
localhost:3000/error/Operation%20Failed:%20Network%20Error
Isolated Data
Products
Suppliers
Operation Failed: Network Error
OK

Figure 23-5. *Displaying an error message*

MAKING CROSS-ORIGIN REQUESTS

By default, browsers enforce a security policy that only allows JavaScript code to make asynchronous HTTP requests within the same origin as the document that contains them. This policy is intended to reduce the risk of cross-site scripting (CSS) attacks, where the browser is tricked into executing malicious code, which is described at `http://en.wikipedia.org/wiki/Cross-site_scripting`. For web application developers, the same-origin policy can be a problem when using web services because they are often outside of the origin that contains the application's JavaScript code. Two URLs are considered to be in the same origin if they have the same protocol, host, and port, and they have different origins if this is not the case. The URL that I use for the RESTful web service in this chapter has a different origin to the URL used by the main application because they use different TCP ports.

The Cross-Origin Resource Sharing (CORS) protocol is used to send requests to different origins. With CORS, the browser includes headers in the asynchronous HTTP request that provide the server with the origin of the JavaScript code. The response from the server includes headers that tell the browser whether it is willing to accept the request. The details of CORS are outside the scope of this book, but there is an introduction to the topic at `https://en.wikipedia.org/wiki/Cross-origin_resource_sharing`, and the CORS specification is available at `www.w3.org/TR/cors`.

CORS is something that happens automatically in this chapter. The `json-server` package that provides the RESTful web service supports CORS and will accept requests from any origin, while the Axios package that I use to make HTTP requests automatically applies CORS. When you select software for your own projects, you must either select a platform that will allow all requests to be handled through a single origin or configure CORS so that the server will accept the application's requests for data.

Consuming a Web Service with a Data Store

The components I defined in the previous section are isolated from one another and are coordinated only through the URL routing system. The advantage of this approach is simplicity, but it can lead to repeatedly requesting the same data from the web service as the user navigates around the application and each component sends its HTTP requests when it is mounted. If the application uses a data store, then the data can be shared between components.

Creating the New Middleware

The store already has actions that receive objects and update the data it contains, so the approach I am going to take is to create new Redux middleware that will intercept the existing actions and send the corresponding HTTP requests to the web service. I added a file called `RestMiddleware.js` to the `src/webservice` folder, with the contents shown in Listing 23-20.

Listing 23-20. The Contents of the RestMiddleware.js File in the src/webservice Folder

```
import { STORE, UPDATE, DELETE} from "../store/modelActionTypes";
import { RestDataSource } from "./RestDataSource";
import { PRODUCTS, SUPPLIERS } from "../store/dataTypes";

export const GET_DATA = "rest_get_data";

export const getData = (dataType) => {
    return {
        type: GET_DATA,
        dataType: dataType
    }
}

export const createRestMiddleware = (productsURL, suppliersURL) => {

    const dataSources = {
        [PRODUCTS]: new RestDataSource(productsURL, () => {}),
        [SUPPLIERS]: new RestDataSource(suppliersURL, () => {})
    }

    return ({dispatch, getState}) => next => action => {
        switch (action.type) {
            case GET_DATA:
                if (getState().modelData[action.dataType].length === 0) {
                    dataSources[action.dataType].GetData((data) =>
                        data.forEach(item => next({ type: STORE,
                            dataType: action.dataType, payload: item})));
                }
                break;
            case STORE:
                action.payload.id = null;
                dataSources[action.dataType].Store(action.payload, data =>
                    next({ ...action, payload: data }))
                break;
            case UPDATE:
                dataSources[action.dataType].Update(action.payload, data =>
                     next({ ...action, payload: data }))
                break;
            case DELETE:
                dataSources[action.dataType].Delete({id: action.payload },
                    () => next(action));
                break;
            default:
                next(action);
        }
    }
}
```

One new action is required, which is to request the data from the web service. This hasn't been required previously because the data store has been automatically initialized with data. The action type is GET_DATA, and Listing 23-20 defines a getData action creator.

The createRestMiddleware function accepts data sources for the product and supplier data and returns middleware that deals with the new GET_DATA action and the existing STORE, UPDATE, and DELETE actions by sending a request to the web service and then dispatching additional actions when the result is received, using the existing features of the data store.

Adding the Middleware to the Data Store

In Listing 23-21, I have added the new middleware to the data store. As noted in Chapter 20, middleware components are applied in the order in which they are added to the store.

Listing 23-21. Applying Middleware in the index.js File in the src/store Folder

```
import { createStore, combineReducers, applyMiddleware, compose } from "redux";
import modelReducer from "./modelReducer";
import stateReducer from "./stateReducer";
import { customReducerEnhancer } from "./customReducerEnhancer";
import { multiActions } from "./multiActionMiddleware";
import { asyncEnhancer } from "./asyncEnhancer";
import { createRestMiddleware } from "../webservice/RestMiddleware";

const enhancedReducer = customReducerEnhancer(
    combineReducers(
        {
            modelData: modelReducer,
            stateData: stateReducer
        })
);

const restMiddleware = createRestMiddleware(
    "http://localhost:3500/api/products",
    "http://localhost:3500/api/suppliers");
```

```
export default createStore(enhancedReducer,
    compose(applyMiddleware(multiActions),
        applyMiddleware(restMiddleware),
        asyncEnhancer(2000)));

export { saveProduct, saveSupplier, deleteProduct, deleteSupplier }
    from "./modelActionCreators";
```

The order is important when considering how the data store is used by the existing components in the application. The `multiActions` middleware created in Chapter 20 allows arrays of actions to be dispatched, and this must come first; otherwise, the new middleware won't properly process actions.

Completing the Application Changes

To automatically request the data on demand, I added a file called `DataGetter.js` to the `src` folder and used it to define the higher-order component shown in Listing 23-22.

Listing 23-22. The Contents of the DataGetter.js File in the src Folder

```
import React, { Component } from "react";
import { PRODUCTS, SUPPLIERS } from "./store/dataTypes";

export const DataGetter = (dataType, WrappedComponent) => {

    return class extends Component {
        render() {
            return <WrappedComponent { ...this.props } />
        }

        componentDidMount() {
            this.props.getData(PRODUCTS);
            if (dataType === SUPPLIERS) {
                this.props.getData(SUPPLIERS);
            }
        }
    }
}
```

The component requests the data after it mounts and knows that the supplier data must be complemented by the product data in order to display the data correctly to the user so that product names can be shown. In Listing 23-23, I have added support for the new HOC in the `TableConnector` component, which ensures that the data required by the application is requested when the application starts.

Listing 23-23. Dispatching Actions in the TableConnector.js File in the src/store Folder

```
import { connect } from "react-redux";
//import { startEditingProduct, startEditingSupplier } from "./stateActions";
import { deleteProduct, deleteSupplier } from "./modelActionCreators";
import { PRODUCTS, SUPPLIERS } from "./dataTypes";
import { withRouter } from "react-router-dom";
import { getData } from "../webservice/RestMiddleware";
import { DataGetter } from "../DataGetter";

export const TableConnector = (dataType, presentationComponent) => {

    const mapStateToProps = (storeData, ownProps) => {
        if (dataType === PRODUCTS) {
            return { products: storeData.modelData[PRODUCTS] };
        } else {
            return {
                suppliers: storeData.modelData[SUPPLIERS].map(supp => ({
                    ...supp,
                    products: supp.products.map(id =>
                        storeData.modelData[PRODUCTS]
                            .find(p => p.id === Number(id)) || id)
                            .map(val => val.name || val)
                    }))
            }
        }
    }

    const mapDispatchToProps = (dispatch, ownProps) => {
        return {
            getData: (type) => dispatch(getData(type)),
            deleteCallback: dataType === PRODUCTS
                ? (...args) => dispatch(deleteProduct(...args))
                : (...args) => dispatch(deleteSupplier(...args))
        }
    }

    const mergeProps = (dataProps, functionProps, ownProps) => {
        let routedDispatchers = {
            editCallback: (target) => {
                ownProps.history.push(`/${dataType}/edit/${target.id}`);
            },
            deleteCallback: functionProps.deleteCallback,
            getData: functionProps.getData

        }
        return Object.assign({}, dataProps, routedDispatchers, ownProps);
    }

    return withRouter(connect(mapStateToProps,
        mapDispatchToProps, mergeProps)(DataGetter(dataType,
            presentationComponent)));
}
```

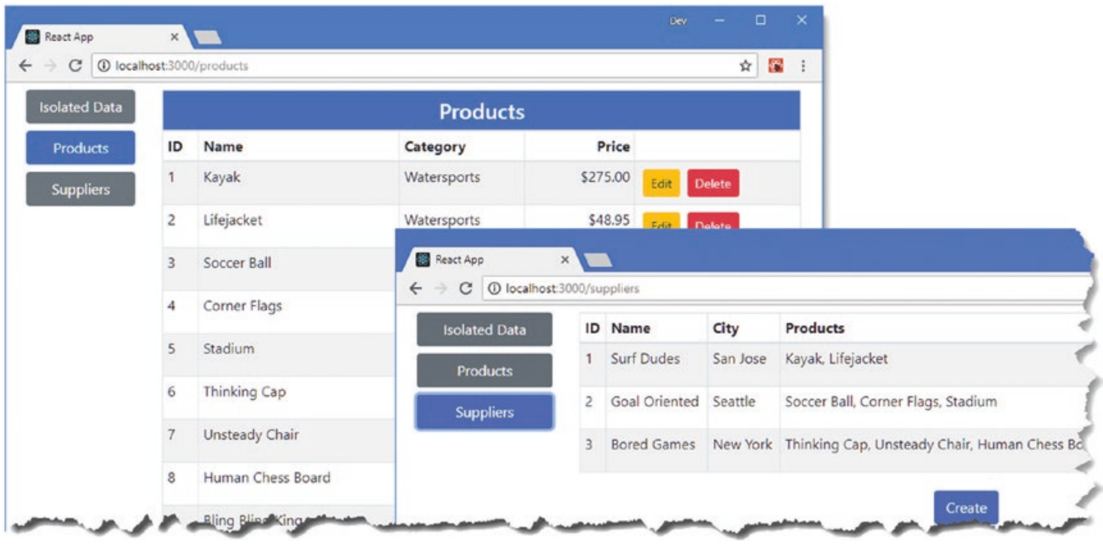
The final change is to remove the static content that was used to seed the data store, as shown in Listing 23-24.

Listing 23-24. Removing the Static Data in the initialData.js File in the src/store Folder

```
import { PRODUCTS, SUPPLIERS } from "./dataTypes";

export const initialData = {
    modelData: {
        [PRODUCTS]: [],
        [SUPPLIERS]: []
    },
    stateData: {
        editing: false,
        selectedId: -1,
        selectedType: PRODUCTS
    }
}
```

The result is that the initial product and supplier data are obtained from the web service and that any changes will trigger updates to the web service, as shown in Figure 23-6.



Products

ID	Name	Category	Price	
1	Kayak	Watersports	$275.00	Edit Delete
2	Lifejacket	Watersports	$48.95	Edit Delete
3	Soccer Ball			
4	Corner Flags			
5	Stadium			
6	Thinking Cap			
7	Unsteady Chair			
8	Human Chess Board			
	Bling Bling King			

ID	Name	City	Products
1	Surf Dudes	San Jose	Kayak, Lifejacket
2	Goal Oriented	Seattle	Soccer Ball, Corner Flags, Stadium
3	Bored Games	New York	Thinking Cap, Unsteady Chair, Human Chess Bo

Figure 23-6. *Using the web service with the data store*

Summary

In this chapter, I introduced a web service and used it to obtain the data displayed by the user, store new data, make changes, and delete data. I used the Axios library in this chapter, but there are many other options available, and consuming a web service in a React application is a relatively simple process. In the next chapter, I introduce GraphQL, which is a more flexible alternative to REST for web services.