CHAPTER 19

Using a Redux Data Store

A *data store* moves the application's data outside of the React component hierarchy. Using a data store means that the data doesn't have to be lifted up to the top-level component and doesn't have to thread props to ensure access to that data where it is needed. The result is a more natural application structure, which leaves the React components to focus on what they are good at, which is rendering content for the user.

But data stores can be complex, and introducing them to an application can be a counterintuitive process. In this chapter, I introduce Redux, which is the most popular choice of data store for React projects, and show you to create a data store and integrate into an application. In Chapter 20, I explain how Redux works in more depth and explain some of its advanced features. Table 19-1 puts using a Redux data store in context.

Table 19-1. *Putting Redux Data Stores in Context*

Question	Answer
What is it?	A data store moves an application's data outside of the component hierarchy, which means that data doesn't have to be lifted up and then made available to descendants through prop threading.
Why is it useful?	Data stores can simplify the components in a project, producing an application that is easier to develop and test.
How is it used?	Data is moved into a dedicated part of the application that can be accessed directly by the components that require it. In the case of Redux, components are connected to the data store through props, which takes advantage of the nature of React, although the mapping process itself can be awkward and require close attention.
Are there any pitfalls or limitations?	Data stores can be complex and often work in counterintuitive ways. Some data store packages, including Redux, enforce specific methods of dealing with data that some developers find restrictive.
Are there any alternatives?	Not all applications need a data store. For smaller amounts of data, using component state features may be acceptable and the React context API, described in Chapter 14, can be used for basic data management features.

© Adam Freeman 2019
A. Freeman, *Pro React 16*, https://doi.org/10.1007/978-1-4842-4451-7_19

Table 19-2 summarizes the chapter.

Table 19-2. *Chapter Summary*

Problem	Solution	Listing
Create a data store	Define the initial data, action types, creators, and reducers	3–8, 13–21
Add a data store to a React application	Use the `Provider` component from the React-Redux package	9
Consume a data store in a React component	Use the `connect` function to map the component's props to the data store's data and action creators	10, 12
Dispatch multiple data store actions	Use the `dispatch` function directly when mapping data store action creators to component function props.	22

Preparing for This Chapter

In this chapter, I continue using the `productapp` project created in Chapter 18. To prepare for this chapter, open a new command prompt and run the commands shown in Listing 19-1 in the `productapp` folder.

■ **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 19-1. Installing Packages

```
npm install redux@4.0.1
npm install react-redux@6.0.0
```

For quick reference, Table 19-3 describes the packages added to the project by the commands in Listing 19-1.

Table 19-3. *The Packages Added to the Project*

Name	Description
`redux`	This package contains the main Redux data store features.
`react-redux`	This package contains the integration features for using Redux with React.

Once the packages have been installed, run the command shown in Listing 19-2 in the `productapp` folder to start the React development tools.

Listing 19-2. Starting the Development Tools

```
npm start
```

Once the application has been compiled, the development HTTP server will start and display the content shown in Figure 19-1.

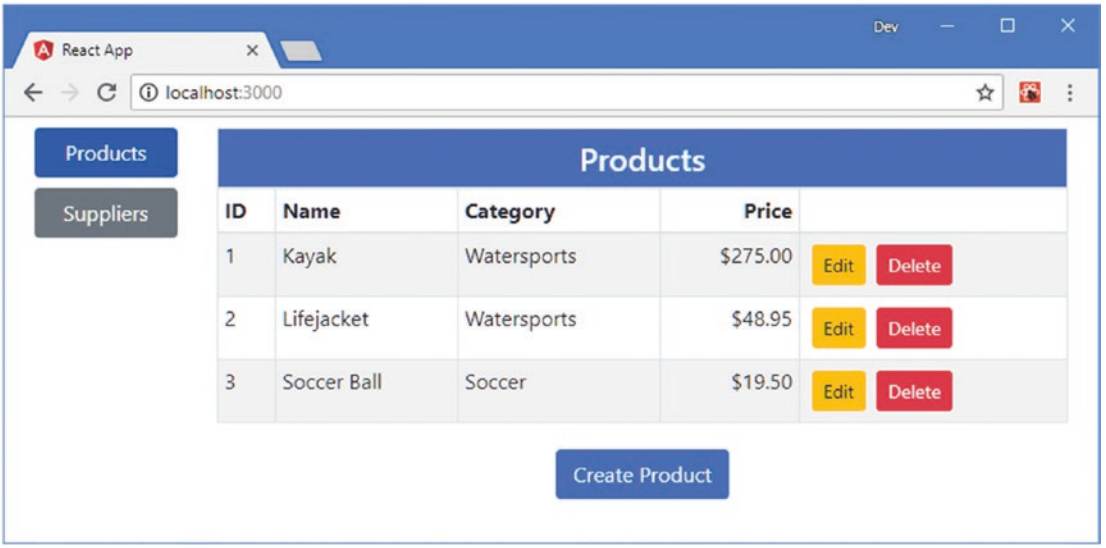

Figure 19-1. *Running the example application*

Creating a Data Store

Much like React, Redux imposes a specific flow for data and changes. And, like React, understanding how the different parts of Redux fit together can be difficult at first. There are two characteristics of Redux that cause confusion.

First, changes in Redux are not applied directly to the data in the store, even though that data is expressed as regular JavaScript objects. Instead, Redux relies on functions that accept a payload and update the data in the store, similar to the way that React components enforce the use of the `setState` method to update state data.

The second point of confusion is the terminology. There are a number of different parts in a Redux data store, and their names don't intuitively describe their purpose. For quick references as you get started with Redux, Table 19-4 describes the terms that you will encounter and that are explained in more detail the sections that follow, where I create a data store and integrate it into the example application.

Table 19-4. *Important Redux Terms*

Name	Description
action	An action describes an operation that will change the data in the store. Redux doesn't allow data to be modified directly and requires actions to specify changes.
action type	Actions are plain JavaScript objects that have a type parameter, which specifies the action type. This ensures that actions can be identified and processed correctly.
action creator	An action creator is a function that creates an action. Action creators are presented to React components as function props so that invoking the action creator function applies a change to the data store.
reducer	A reducer is a function that receives an action and processes the change it represents in the data store. An action specifies which operation should be applied to the data store, but it is the reducer that contains the JavaScript code that makes it happen.
selector	A selector provides a component with access to the data it requires from the data store. Selectors are presented to React components as data props.

CHOOSING AN ALTERNATIVE DATA STORE PACKAGE

Redux is only one of the data store packages available for use with React, although it is the most well-known and the one that is chosen by most projects. If you don't like the way that Redux works, then MobX (`https://github.com/mobxjs/mobx`) may be a good alternative. MobX works well with React and allows direct state changes. The main drawback is that it relies on decorators, which some developers find awkward and which are not yet part of the JavaScript specification (although they are widely used, including by Angular).

In Chapters 24 and 25, I introduce GraphQL and explain its use in retrieving data for applications. If you become a committed React user, then you may want to consider Relay (`https://facebook.github.io/relay`) for data management. Relay works only with GraphQL, which means that it isn't suitable for all projects, but it has some interesting features and integrates well with React.

Defining the Data Types

The example application contains similar sets of features applied to two types of data. In this situation, it is easy to end up with duplication in the code that manages the data store, performing essentially the same operation but on different collections of objects, with the effect that the data store is harder to write, harder to understand, and prone to errors introduced by copying the code for one type of data and incorrectly adapting it.

This is such a common problem that I am going to demonstrate a data store that consolidates as much common code as possible. The first step is to define constant values that will let me consistently identify the different types of data throughout the data store. I created the `src/store` folder and added to it a file called `dataTypes.js` with the statements shown in Listing 19-3.

Listing 19-3. The Contents of the dataTypes.js File in the src/store Folder

```
export const PRODUCTS = "products";
export const SUPPLIERS = "suppliers";
```

Defining the Initial Data

In later chapters, I show you how to get data from web services, but for the moment I am going to continue using statically defined data. To define the initial contents of the data store, I created a file called `initialData.js` in the `store` folder and added the statements shown in Listing 19-4.

■ **Note** As I add more features to the example application, I am going to create different sections of the data store to keep features separate. I am going to refer to the product and supplier data presented to the user as *model data* to differentiate it from the internal data used to coordinate between components, which I will refer to as *state data*.

Listing 19-4. The Contents of the initialData.js File in the src/store Folder

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

export const initialData = {
    [PRODUCTS]: [
        { id: 1, name: "Trail Shoes", category: "Running", price: 100 },
        { id: 2, name: "Thermal Hat", category: "Running", price: 12 },
        { id: 3, name: "Heated Gloves", category: "Running", price: 82.50 }],
    [SUPPLIERS]: [
        { id: 1, name: "Zoom Shoes", city: "London", products: [1] },
        { id: 2, name: "Cosy Gear", city: "New York", products: [2, 3] }],
}
```

The initial state of the data store is defined as a regular JavaScript object; one of the characteristics of working with Redux is that it relies on pure JavaScript for many of its features. To make it clear when the data store is being used, I have used different details for the objects in the `PRODUCTS` and `SUPPLIERS` arrays.

Defining the Model Data Action Types

The next step is to describe the operations that can be performed on the data in the store, which are called *actions*. There can be a lot of actions in a complex application, and it can be helpful to define constant values to identify them. I added a file called `modelActionTypes.js` to the `store` folder and added the content shown in Listing 19-5.

Listing 19-5. The Contents of the modelActionTypes.js File in the src/store Folder

```
export const STORE  = "STORE";
export const UPDATE = "UPDATE";
export const DELETE = "DELETE";
```

To provide the functionality for the example application, I need three events: `STORE` to add objects to the data store, `UPDATE` to modify existing objects, and `DELETE` to remove objects.

The value assigned to the action types isn't important just as long as it is unique, and the simplest approach is to assign each action type a string value of its name.

Defining the Model Action Creators

Actions are objects that are sent from the application to the data store to request a change. An action has an action type and a data payload, where the action type specifies the operation and the payload provides the data that the operation requires. Actions are ordinary JavaScript objects that can define any combination of properties required to describe an operation. The convention is to define a `type` property to indicate the event type, and I will supplement this with `dataType` and `payload` properties to specify the data that the action should be applied to and the data required for the action.

Actions are created by *action creators*, which is the name given to functions that accept data from the application and return an action that describes a change to the data store. To define the action creators, I added a file called `modelActionCreators.js` to the `store` folder and added the code shown in Listing 19-6.

Listing 19-6. The Contents of the modelActionCreators.js File in the src/store Folder

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

let idCounter = 100;

export const saveProduct = (product) => {
    return createSaveEvent(PRODUCTS, product);
}

export const saveSupplier = (supplier) => {
    return createSaveEvent(SUPPLIERS, supplier);
}

const createSaveEvent = (dataType, payload)  => {
    if (!payload.id) {
        return {
            type: STORE,
            dataType: dataType,
            payload: { ...payload, id: idCounter++ }
        }
    } else {
        return {
            type: UPDATE,
            dataType: dataType,
            payload: payload
        }
    }
}

export const deleteProduct = (product) => ({
    type: DELETE,
    dataType: PRODUCTS,
    payload: product.id
})
```

```
export const deleteSupplier = (supplier) => ({
    type: DELETE,
    dataType: SUPPLIERS,
    payload: supplier.id
})
```

There are four action creators in the listing. The `saveProduct` and `saveSupplier` functions receive an object parameter and pass it to `createSaveEvent`, which inspects the value of the `id` property to determine whether a `STORE` or `UPDATE` action is required. The `deleteProduct` and `deleteSupplier` action creators are simpler and create a `DELETE` action whose payload is the `id` property value of the object to be deleted.

Defining the Reducer

Actions are applied to the data store by a JavaScript function called a *reducer*. Put another way, an action describes the type of change that is needed, and the reducer contains the logic to make it happen. I added a file called `modelReducer.js` to the `store` folder and added the code shown in Listing 19-7.

Listing 19-7. The Contents of the modelReducer.js File in the src/store Folder

```
import { STORE, UPDATE, DELETE } from "./modelActionTypes";
import { initialData } from "./initialData";

export default function(storeData, action) {
    switch (action.type) {
        case STORE:
            return {
                ...storeData,
                [action.dataType]:
                    storeData[action.dataType].concat([action.payload])
            }
        case UPDATE:
            return {
                ...storeData,
                [action.dataType]: storeData[action.dataType].map(p =>
                    p.id === action.payload.id ? action.payload : p)
            }
        case DELETE:
            return {
                ...storeData,
                [action.dataType]: storeData[action.dataType]
                    .filter(p => p.id !== action.payload)
            }
        default:
            return storeData || initialData;
    }
}
```

The reducer receives the current data from the data store and an action as its parameters. It inspects the action and uses it to create a new data object, which will replace the existing data in the data store.

There are two important rules to follow. First, the reducer must create a new object and not return the object received as a parameter because Redux will ignore any changes that have been made. Second, because the object that the reducer creates replaces the data in the store, it is important to copy the properties of the existing object, not just the one modified by the action. The simplest way to copy the properties is to use the spread operator, like this:

```
...
case STORE:
    return {
        ...store,
        [action.dataType]: store[action.dataType].concat([action.payload])
}
...
```

This ensures that all the properties are copied to the result object. The property for the data that is changed is then replaced with the data modified by the action.

Another important aspect of the reducer is that it will be invoked when the data store is created to get the initial data. This is handled by the `default` clause of the `switch` statement, as shown here:

```
...
default:
    return storeData || initialData;
...
```

Redux will report an error if the function returns `undefined`, and it is important to ensure that you return a useful result. In the listing, I return the `initialData` object that was defined in Listing 19-4.

AVOIDING CODE DUPLICATION IN THE REDUCER

Most data sets require a core set of common operations. This can be seen in the example application, where the product and supplier data both need store, update, and delete operations. This can result in code duplication when you define the data store, with similar action types, action creators, and reducer code. The approach I have taken in this section is to include a property in the actions that specifies which type of data an operation should be applied to, and then I relied on the JavaScript property accessor feature to select the appropriate data store property in the reducer, like this:

```
...
case STORE:
    return {
        ...store,
        [action.dataType]: store[action.dataType].concat([action.payload])
}
...
```

When the new data store object is created, JavaScript will evaluate the `action.dataType` property and use its value to define a new property on the object and access the property on the old data store, using the values I defined in Listing 19-5, so that a `dataType` value of `PRODUCTS` selects the products data and a value of `SUPPLIERS` selects the supplier data. You don't have to use this type of technique in your own projects, but it helps keep the code concise and manageable.

Creating the Data Store

Redux provides the `createStore` function, which creates the data store and prepares it for use. I added a file called `index.js` to the `store` folder and added the code shown in Listing 19-8.

■ **Tip** You don't have to use the `index.js` file name, but doing so allows the data store to be imported using only the name of the folder, as shown in Listing 19-9.

Listing 19-8. The Contents of the index.js File in the src/store Folder

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

export default createStore(modelReducer);

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

The default export from the `index.js` file is the result of calling `createStore`, which accepts the reducer function as its argument. I also exported the action creators so that all of the data store's functionality can be accessed through a single `import` statement elsewhere in the application, which makes using the data store a little simpler.

Using the Data Store in the React Application

The actions, reducers, and selectors I created in the previous section are not yet integrated into the application, and there are no links between the components in the application and the data in the data store. In the sections that follow, I show you how to use the data store to replace the state data and methods that currently manage the application data.

Applying the Data Store to the Top-Level Component

The React-Redux package includes a React container component that provides access to a data store. This component, called `Provider`, is applied at the top of the component hierarchy so that the data store is available throughout the application. In Listing 19-9, I imported the data store from the `index.js` file created in Listing 19-8 and used the `Provider` component to apply the data store to the components in the application.

Listing 19-9. Applying the Data Store in the App.js File in the src Folder

```
import React, { Component } from "react";
import ProductsAndSuppliers from "./ProductsAndSuppliers";
import { Provider } from "react-redux";
import dataStore from "./store";

export default class App extends Component {

    render() {
```

```
        return (
            <Provider store={ dataStore }>
               <ProductsAndSuppliers/>
            </Provider>
        )
    }
}
```

The `Provider` component has a `store` prop that is used to specify the data store, which I assigned the name `dataStore` in the `import` statement.

Connecting the Product Data

The next step is to connect the data store to the components that require the data it contains and the action creators that operate on it. I am going to take the most direct approach, which is to use the features provided by the React-Redux package to connect the `ProductDisplay` component to the data store, as shown Listing 19-10.

Listing 19-10. Connecting to the Data Store in the ProductDisplay.js File in the src Folder

```
import React, { Component } from "react";
import { ProductTable } from "./ProductTable"
import { ProductEditor } from "./ProductEditor";
import { connect } from "react-redux";
import { saveProduct, deleteProduct } from "./store"

const mapStateToProps = (storeData) => ({
    products: storeData.products
})

const mapDispatchToProps = {
    saveCallback: saveProduct,
    deleteCallback: deleteProduct
}

const connectFunction = connect(mapStateToProps, mapDispatchToProps);

export const ProductDisplay = connectFunction(
        class extends Component {

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

        startEditing = (product) => {
            this.setState({ showEditor: true, selectedProduct: product })
        }
```

```
        createProduct = () => {
            this.setState({ showEditor: true, selectedProduct: {} })
        }

        cancelEditing = () => {
            this.setState({ showEditor: false, selectedProduct: null })
        }

        saveProduct = (product) => {
            this.props.saveCallback(product);
            this.setState({ showEditor: false, selectedProduct: null })
        }

        render() {
            if (this.state.showEditor) {
                return <ProductEditor
                    key={ this.state.selectedProduct.id || -1 }
                    product={ this.state.selectedProduct }
                    saveCallback={ this.saveProduct }
                    cancelCallback={ this.cancelEditing } />
            } else {
                return <div className="m-2">
                    <ProductTable products={ this.props.products }
                        editCallback={ this.startEditing }
                        deleteCallback={ this.props.deleteCallback } />
                    <div className="text-center">
                        <button className="btn btn-primary m-1"
                            onClick={ this.createProduct }>
                            Create Product
                        </button>
                    </div>
                </div>
            }
        }
    })
```

The first step is to define a function that receives the data store and selects the props that will connect the component and the store, like this:

```
...
const mapStateToProps = (storeData) => ({
    products: storeData.products
})
...
```

This function is conventionally named `mapStateToProps`, and it returns an object that maps prop names for the connected component to data in the store. These mappings are known as *selectors* because they select the data that will be mapped to the component's prop. In this case, selector maps the store's `products` array to a prop named `products`.

The next step is to create the object that will map the function props that the component requires to data store action creators, like this:

```
...
const mapDispatchToProps = {
    saveCallback: saveProduct,
    deleteCallback: deleteProduct
}
...
```

The React-Redux package supports different ways of connecting action creators to function props, but this is the simplest, which is to create an object that maps prop names to action creator functions. When the component is connected to the data store, the action creator functions defined in this object will be wired up so that the reducer is automatically invoked. In this case, I mapped the `saveProduct` and `deleteProduct` action creators to function props named `saveCallback` and `deleteCallback`.

Once the mappings for data and function props have been defined, they are passed to the `connect` function, provided by the React-Redux package.

```
...
const connectFunction = connect(mapStateToProps, mapDispatchToProps);
...
```

The `connect` function creates a higher-order component (HOC) that passes on props connected to the data store merged with the props that are provided by the parent component.

■ **Tip** Higher-order components are described in Chapter 14.

The final step is to pass a component to the function returned by `connect`, like this:

```
...
export const ProductDisplay = connectFunction(class extends Component {
...
```

The result is a component whose props are connected to the data store. When you save the changes in Listing 19-10, the application will display the data defined in Listing 19-4, as shown in Figure 19-2.

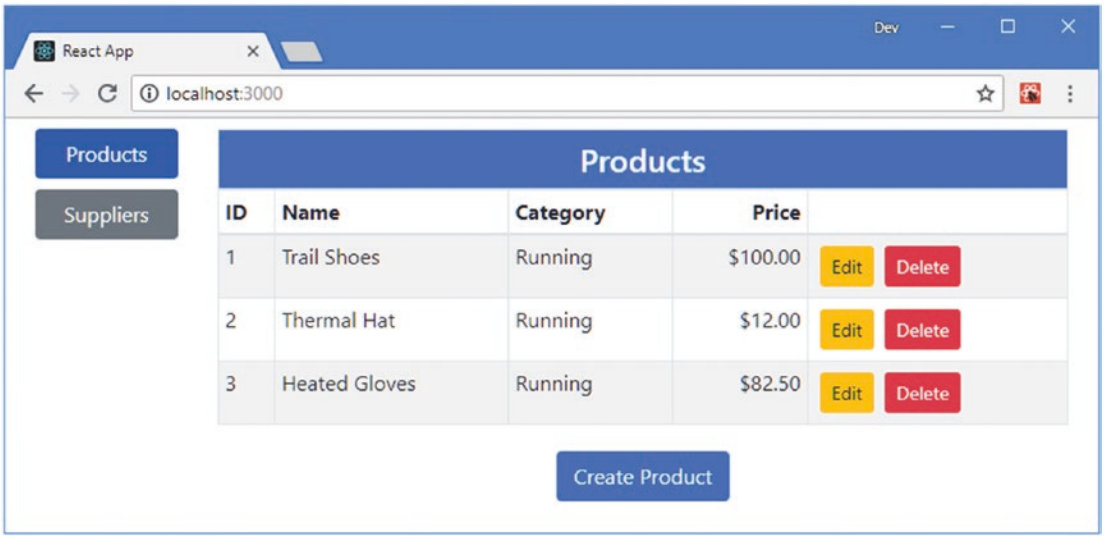

Figure 19-2. *Using a data store for product data*

Because the props provided by the data store replace those from the parent component, the `ProductDisplay` component operates entirely on the data store data, including creating, editing, and deleting objects.

Connecting the Supplier Data

The same process can be applied to connect the supplier data, as shown in Listing 19-11, where I have used the `connect` method to provide the `SupplierDisplay` component with access to the data store.

Listing 19-11. Connecting to the Data Store in the SupplierDisplay.js File in the src Folder

```
import React, { Component } from "react";
import { SupplierEditor } from "./SupplierEditor";
import { SupplierTable } from "./SupplierTable";
import { connect } from "react-redux";
import { saveSupplier, deleteSupplier} from "./store";

const mapStateToProps = (storeData) => ({
    suppliers: storeData.suppliers
})

const mapDispatchToProps = {
    saveCallback: saveSupplier,
    deleteCallback: deleteSupplier
}

const connectFunction = connect(mapStateToProps, mapDispatchToProps);

export const SupplierDisplay = connectFunction(
    class extends Component {
```

```
        constructor(props) {
            super(props);
            this.state = {
                showEditor: false,
                selected: null
            }
        }

        startEditing = (supplier) => {
            this.setState({ showEditor: true, selected: supplier })
        }

        createSupplier = () => {
            this.setState({ showEditor: true, selected: {} })
        }

        cancelEditing = () => {
            this.setState({ showEditor: false, selected: null })
        }

        saveSupplier= (supplier) => {
            this.props.saveCallback(supplier);
            this.setState({ showEditor: false, selected: null })
        }

        render() {
            if (this.state.showEditor) {
                return <SupplierEditor
                    key={ this.state.selected.id || -1 }
                    supplier={ this.state.selected }
                    saveCallback={ this.saveSupplier }
                    cancelCallback={ this.cancelEditing } />
            } else {
                return <div className="m-2">
                        <SupplierTable suppliers={ this.props.suppliers }
                            editCallback={ this.startEditing }
                            deleteCallback={ this.props.deleteCallback }
                        />
                        <div className="text-center">
                            <button className="btn btn-primary m-1"
                                onClick={ this.createSupplier }>
                                    Create Supplier
                            </button>
                        </div>
                </div>
            }
        }
    })
```

The result is that the SupplierDisplay component receives props that connect it to the data store, as illustrated in Figure 19-3.

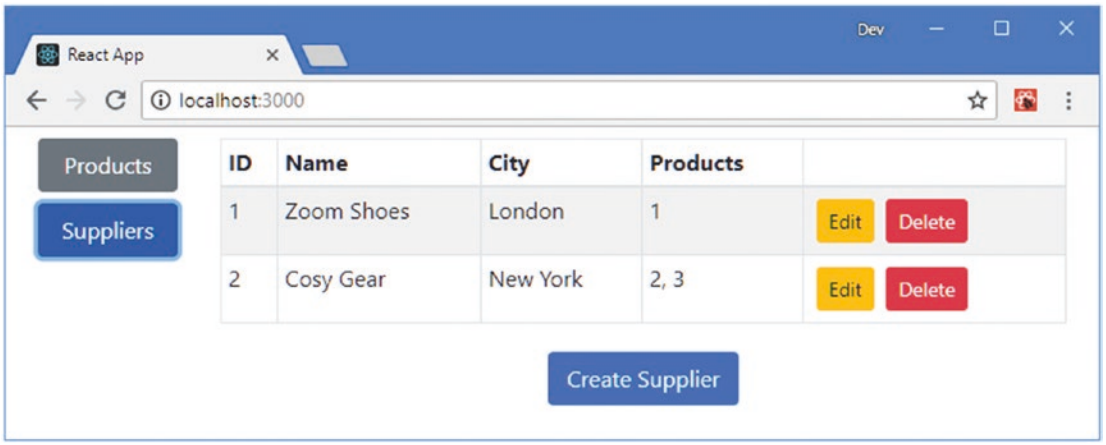

Figure 19-3. *Using a data store for supplier data*

With the data store in place, the ProductsAndSuppliers component is redundant, since its role was to provide the product and supplier data and the methods to store and delete it. In Listing 19-12, I have updated the App component to display the Selector, ProductDisplay, and SupplierDisplay components directly.

Listing 19-12. Displaying Content Directly in the App.js File in the src Folder

```
import React, { Component } from "react";
//import ProductsAndSuppliers from "./ProductsAndSuppliers";
import { Provider } from "react-redux";
import dataStore from "./store";
import { Selector } from "./Selector";
import { ProductDisplay } from "./ProductDisplay";
import { SupplierDisplay } from "./SupplierDisplay";

export default class App extends Component {

    render() {
        return (
            <Provider store={ dataStore }>
                <Selector>
                    <ProductDisplay name="Products" />
                    <SupplierDisplay name="Suppliers" />
                </Selector>
            </Provider>
        )
    }
}
```

Notice I don't have to provide props for the ProductDisplay and SupplierDisplay components to give them access to data and methods; these will be set up by the connect method that connects the components to the data store.

Expanding the Data Store

Data stores are not just for the data that is displayed to the user—they can also be used to store the state data that is used to coordinate and manage components. Expanding the data store to include state data will allow me to connect the model data in the store directly to the components that use it, which is not possible currently because ProductDisplay and SupplierDisplay maintain state data that is used to select the content presented to the user.

In the sections that follow, I move the state data and code that manages it into the data store so that I can further simplify the application.

Adding State Data to the Store

I want to keep the state data separate from the model data, so I am going to add some structure to the store. I like to represent the structure in the initial data that I used to populate the data store, although this is entirely to help me understand the shape of the data that I am working with and is not a requirement enforced by Redux.

To structure the store data, I moved the existing data to a property named modelData and added a new stateData section, as shown in Listing 19-13.

Listing 19-13. Expanding the Data in the initialData.js File in the src/store Folder

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

export const initialData = {
    modelData: {
        [PRODUCTS]: [
            { id: 1, name: "Trail Shoes", category: "Running", price: 100 },
            { id: 2, name: "Thermal Hat", category: "Running", price: 12 },
            { id: 3, name: "Heated Gloves", category: "Running", price: 82.50 }],
        [SUPPLIERS]: [
            { id: 1, name: "Zoom Shoes", city: "London", products: [1] },
            { id: 2, name: "Cosy Gear", city: "New York", products: [2, 3] }],
    },
    stateData: {
        editing: false,
        selectedId: -1,
        selectedType: PRODUCTS
    }
}
```

My goal is to move the state data and logic in the ProductDisplay and SupplierDisplay components into the data store. These components track the user's selection for editing and whether the table or the editor component should be rendered. To provide this information in the store, I defined editing, selected and selectedType properties in the stateData section.

Defining the Action Types and Creators for State Data

Next, I need to define the actions for the state data in the store. When I set up the data store, I defined the action types and the creators in different files, but that's not a requirement, and both can be defined together. To separate the state data actions from the rest of the store, I added a file called `stateActions.js` to the `src/store` folder and used it to define the action types and creators shown in Listing 19-14.

Listing 19-14. The Contents of the stateActions.js File in the src/store Folder

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

export const STATE_START_EDITING = "state_start_editing";
export const STATE_END_EDITING = "state_end_editing";
export const STATE_START_CREATING = "state_start_creating";

export const startEditingProduct = (product) => ({
    type: STATE_START_EDITING,
    dataType: PRODUCTS,
    payload: product
})

export const startEditingSupplier = (supplier) => ({
    type: STATE_START_EDITING,
    dataType: SUPPLIERS,
    payload: supplier
})

export const endEditing = () => ({
    type: STATE_END_EDITING
})

export const startCreatingProduct = () => ({
    type: STATE_START_CREATING, dataType: PRODUCTS
})

export const startCreatingSupplier = () => ({
    type: STATE_START_CREATING, dataType: SUPPLIERS
})
```

The action creators correspond to the methods defined by the `ProductDisplay` and `SupplierDisplay` components and allow a user to start editing an object, cancel editing, and start creating a new object.

Defining the State Data Reducer

To update the data store in response to an action, I need to define a reducer. Rather than add code to the existing reducer, I am going to define a separate function to deal with the state data. I added a file called `stateReducer.js` in the `src/store` folder and added the code shown in Listing 19-15.

Listing 19-15. The Contents of the stateReducer.js File in the src/store Folder

```
import { STATE_START_EDITING, STATE_END_EDITING, STATE_START_CREATING }
    from "./stateActions";
import { initialData } from "./initialData";

export default function(storeData, action) {
    switch(action.type) {
        case STATE_START_EDITING:
        case STATE_START_CREATING:
            return {
                ...storeData,
                editing: true,
                selectedId: action.type === STATE_START_EDITING
                    ? action.payload.id : -1,
                selectedType: action.dataType
            }
        case STATE_END_EDITING:
            return {
                ...storeData,
                editing: false
            }
        default:
            return storeData || initialData.stateData;
    }
}
```

The reducer for the state data keeps track of what the user is editing or creating, which echoes the approach taken by the existing components in the example application, although I am going to use a single set of properties to coordinate the editors for both types of model data in the application.

Incorporating the State Data Features into the Store

Redux provides the `combineReducers` function, which allows multiple reducers to be combined for use in a data store, with each reducer responsible for one section of the data store data. In Listing 19-16, I used the `combineReducers` function to combine the reducers for the model and state data.

Listing 19-16. Configuring the Data Store in the index.js File in the src/store Folder

```
import { createStore, combineReducers } from "redux";
import modelReducer from "./modelReducer";
import stateReducer from "./stateReducer";

export default createStore(combineReducers(
    {
        modelData: modelReducer,
        stateData: stateReducer
    }));

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

The argument for the createReducers function is an object whose property names correspond to sections of the data store and the reducers that will manage them. In the listing, I have made the original reducer responsible for the modelData section of the data store and have made the reducer defined in Listing 19-15 responsible for the stateData section. The combined reducers are passed to the createStore function to create the data store.

■ **Note** Each reducer operates on a separate part of the data store, but when an action is processed, each reducer is passed the action until one of them returns a new data store object, indicating that the action has been processed.

Adding structure to the data in the store requires a corresponding change to the initial state returned by the reducer function for the model data, as shown in Listing 19-17.

Listing 19-17. Changing the Initial State in the modelReducer.js File in the src/store Folder

```
import { STORE, UPDATE, DELETE } from "./modelActionTypes";
import { initialData } from "./initialData";

export default function(storeData, action) {
    switch (action.type) {
        case STORE:
            return {
                ...storeData,
                [action.dataType]:
                    storeData[action.dataType].concat([action.payload])
            }
        case UPDATE:
            return {
                ...storeData,
                [action.dataType]: storeData[action.dataType].map(p =>
                    p.id === action.payload.id ? action.payload : p)
            }
        case DELETE:
            return {
                ...storeData,
                [action.dataType]: storeData[action.dataType]
                    .filter(p => p.id !== action.payload)
            }
        default:
            return storeData || initialData.modelData;
    }
}
```

When the combineReducers function is used, each reducer is provided with only its section of the data in the store and is unaware of the rest of the data and the other reducers. This means I only need to change the source of the initial data and don't have to worry about navigating through the new data structure when applying an action.

Connecting the React Components to the Stored State Data

Now that the state data has been put into the data store, I can connect it to components. Rather than configure each component separately, I am going to define separate connector components that will take care of mapping data store features to component props. I created a file called `EditorConnector.js` in the `src/store` folder with the code shown in Listing 19-18.

UNDERSTANDING THE PRESENTER/CONNECTOR PATTERN

A common approach when using a data store is to use two different types of component. Presenter components are responsible for rendering content to the user and responding to user input. They receive data and function props but are not directly connected to the data store. Connector components—confusingly, also known as *container components*—exist to connect to the data store to provide presenter components with props. This is the general approach I have taken in this part of the chapter, although, as with much in the React/Redux world, implementation details can vary, and there is disagreement over how best to approach this kind of separation.

Listing 19-18. The Contents of the EditorConnector.js File in the src/store Folder

```
import { connect } from "react-redux";
import { endEditing } from "./stateActions";
import { saveProduct, saveSupplier } from "./modelActionCreators";
import { PRODUCTS, SUPPLIERS  } from "./dataTypes";

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

    const mapStateToProps = (storeData) => ({
        editing: storeData.stateData.editing
            && storeData.stateData.selectedType === dataType,
        product: (storeData.modelData[PRODUCTS]
            .find(p => p.id === storeData.stateData.selectedId)) || {},
        supplier:(storeData.modelData[SUPPLIERS]
            .find(s => s.id === storeData.stateData.selectedId)) || {}
    })

    const mapDispatchToProps = {
        cancelCallback: endEditing,
        saveCallback: dataType === PRODUCTS ? saveProduct: saveSupplier
    }

    return connect(mapStateToProps, mapDispatchToProps)(presentationComponent);
}
```

The `EditorConnector` is a higher-order component that provides a presentation component with the props required by both the `ProductEditor` and `SupplierEditor` components, which means that these components can be connected to the data store using the same code, rather than requiring separate uses of the `connect` function. To support both types of editor, the HOC function accepts a data type that is used to select the data and action creators that will be mapped to props.

■ **Tip** Notice that the segmentation of the data store created by the `combineReducers` function doesn't have any effect on data selection, which means I can select data from the entire store.

To provide the same service for the components that display the table components, I added a file called `TableConnector.js` to the `src/store` folder and used it to define the HOC shown in Listing 19-19.

Listing 19-19. The Contents of 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";

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

    const mapStateToProps = (storeData) => ({
        products: storeData.modelData[PRODUCTS],
        suppliers: storeData.modelData[SUPPLIERS]
    })

    const mapDispatchToProps = {
        editCallback: dataType === PRODUCTS
            ? startEditingProduct : startEditingSupplier,
        deleteCallback: dataType === PRODUCTS ? deleteProduct : deleteSupplier
    }

    return connect(mapStateToProps, mapDispatchToProps)(presentationComponent);
}
```

Applying the Connector Components

With the connector components in place, I can remove the state data and methods from the `ProductDisplay` and `SupplierDisplay` components. Listing 19-20 shows the simplification of the `ProductDisplay` component.

Listing 19-20. Using Connector Components in the ProductDisplay.js File in the src Folder

```
import React, { Component } from "react";
import { ProductTable } from "./ProductTable"
import { ProductEditor } from "./ProductEditor";
import { connect } from "react-redux";
//import { saveProduct, deleteProduct } from "./store"
import { EditorConnector } from "./store/EditorConnector";
import { PRODUCTS } from "./store/dataTypes";
import { TableConnector } from "./store/TableConnector";
import { startCreatingProduct } from "./store/stateActions";
```

```
const ConnectedEditor = EditorConnector(PRODUCTS, ProductEditor);
const ConnectedTable = TableConnector(PRODUCTS, ProductTable);

const mapStateToProps = (storeData) => ({
    editing: storeData.stateData.editing,
    selected: storeData.modelData.products
        .find(item =>  item.id === storeData.stateData.selectedId) || {}
})

const mapDispatchToProps = {
    createProduct: startCreatingProduct,
}

const connectFunction = connect(mapStateToProps, mapDispatchToProps);

export const ProductDisplay = connectFunction(
    class extends Component {

        // constructor(props) {
        //     super(props);
        //     this.state = {
        //         showEditor: false,
        //         selectedProduct: null
        //     }
        // }

        // startEditing = (product) => {
        //     this.setState({ showEditor: true, selectedProduct: product })
        // }

        // createProduct = () => {
        //     this.setState({ showEditor: true, selectedProduct: {} })
        // }

        // cancelEditing = () => {
        //     this.setState({ showEditor: false, selectedProduct: null })
        // }

        // saveProduct = (product) => {
        //     this.props.saveCallback(product);
        //     this.setState({ showEditor: false, selectedProduct: null })
        // }

        render() {
            if (this.props.editing) {
                return <ConnectedEditor key={ this.props.selected.id || -1 } />
                // return <ProductEditor
                //     key={ this.state.selectedProduct.id || -1 }
                //     product={ this.state.selectedProduct }
                //     saveCallback={ this.saveProduct }
                //     cancelCallback={ this.cancelEditing } />
```

```
            } else {
                return <div className="m-2">
                    <ConnectedTable />
                    {/* <ProductTable products={ this.props.products }
                        editCallback={ this.startEditing }
                        deleteCallback={ this.props.deleteCallback } />  */}
                    <div className="text-center">
                        <button className="btn btn-primary m-1"
                            onClick={ this.props.createProduct }>
                            Create Product
                        </button>
                    </div>
                </div>
            }
        }
    })
```

The number of commented-out statements shows the amount of the `ProductDisplay` component that was dedicated to providing data and function props to its children, all of which is now handled through the data store and the connector components. There is no longer any need for local state data, so the constructor and all of the methods except `render` can be removed. The component does still require a connection to the data store, however, because it needs to know which child component to display and needs to generate key values for the editor components.

Listing 19-21 shows the simplified `SupplierDisplay` component, with the redundant statements removed rather than just commented out.

Listing 19-21. Using Connector Components in the SupplierDisplay.js File in the src Folder

```
import React, { Component } from "react";
import { SupplierEditor } from "./SupplierEditor";
import { SupplierTable } from "./SupplierTable";
import { connect } from "react-redux";
import { startCreatingSupplier } from "./store/stateActions";
import { SUPPLIERS } from "./store/dataTypes";
import { EditorConnector } from "./store/EditorConnector";
import { TableConnector } from "./store/TableConnector";

const ConnectedEditor = EditorConnector(SUPPLIERS, SupplierEditor);
const ConnectedTable = TableConnector(SUPPLIERS, SupplierTable);

const mapStateToProps = (storeData) => ({
    editing: storeData.stateData.editing,
    selected: storeData.modelData.suppliers
        .find(item => item.id === storeData.stateData.selectedId) || {}
})

const mapDispatchToProps = {
    createSupplier: startCreatingSupplier
}
```

```
const connectFunction = connect(mapStateToProps, mapDispatchToProps);

export const SupplierDisplay = connectFunction(
    class extends Component {

        render() {
            if (this.props.editing) {
                return <ConnectedEditor key={ this.props.selected.id || -1 } />
            } else {
                return <div className="m-2">
                    <ConnectedTable />
                    <div className="text-center">
                        <button className="btn btn-primary m-1"
                            onClick={ this.props.createSupplier }>
                                Create Supplier
                        </button>
                    </div>
                </div>
            }
        }
    })
```

OVER-SIMPLIFYING COMPONENTS

As I have pushed the use of the data store further into the component hierarchy, the differences between the components for product and supplier data have been reduced, and these components are converging. At this point, I could replace the `ProductDisplay` and `SupplierDisplay` components with a single component that deals with both data types and keep working toward driving the entire application from the data store. In practice, however, there comes a point where convergence no longer simplifies the application and starts simply moving complexity around. As you gain experience working with the data store, you will find the point where you are comfortable with the degree you rely on the data store and the amount of duplication in the components. Like much of React and Redux development, this is as much personal preference as it is good practice, and it is worth experimenting until you find an approach that suits you.

Dispatching Multiple Actions

There is a problem with the way that the example application uses the data store. If you create a new object or edit an existing one, clicking the Save button updates the data store but doesn't change the component displayed to the user, as shown in Figure 19-4, and you must click the Cancel button to update the data value that changes the selected component.

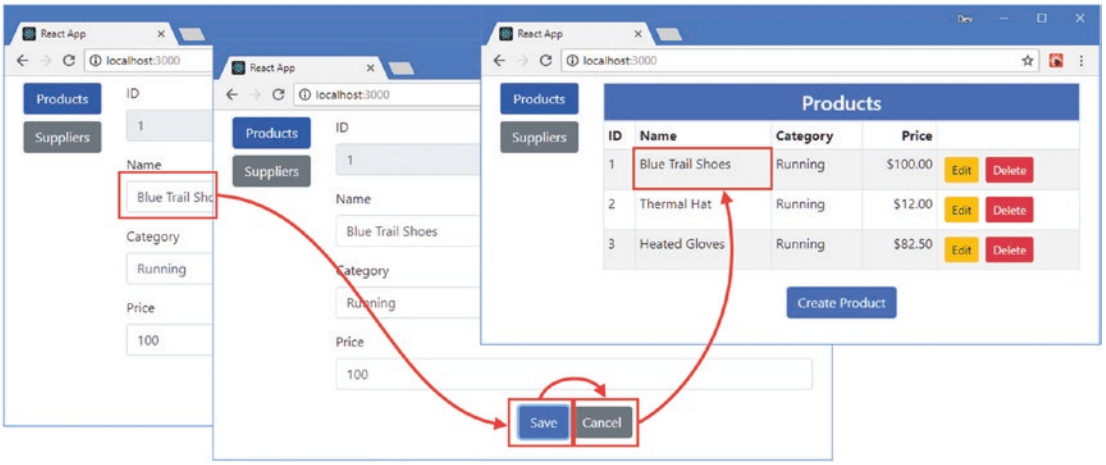

Figure 19-4. *Making a change using the example application*

The problem is that the `connect` function that maps action creators to props allows for only one action creator to be selected by default, but I need two action creators to solve this problem: the `saveProduct` or `saveSupplier` creators to update the model data and the `endEditing` creator to signal that editing is complete and the table should be presented to the user.

I can't define a new creator to perform both tasks because each action is handled by a single reducer and each reducer is responsible for an isolated part of the data in the store, which means that an action can lead to a change in the model data or the state data but not both.

Fortunately, the `connect` function provides an alternative way to map props to action creators that provides more flexibility. When the `mapDispatchToProps` argument to the connect method is an object, the `connect` function wraps each action creator function in a `dispatch` method, which is responsible for sending the action returned by the action creator to the reducer. This means that an object that maps a creator like this:

```
...
const mapDispatchToProps = {
    createSupplier: startCreatingSupplier
}
...
```

```
is transformed into an object like this:...
const mapDispatchToProps = {
    createSupplier: payload => dispatch(startCreatingSupplier(payload))
}
...
```

The action creator is invoked to get the action, which is then passed to the `dispatch` function so it can be processed by a reducer. Instead of defining an object and allowing the `connect` function to wrap each creator, you can define a function that accepts `dispatch` as its argument and produces props that explicitly handle action creation and dispatch, as shown in Listing 19-22.

Listing 19-22. Dispatching Actions in the EditorConnector.js File in the src Folder

```
import { connect } from "react-redux";
import { endEditing } from "./stateActions";
import { saveProduct, saveSupplier } from "./modelActionCreators";
import { PRODUCTS, SUPPLIERS  } from "./dataTypes";

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

    const mapStateToProps = (storeData) => ({
        editing: storeData.stateData.editing
            && storeData.stateData.selectedType === dataType,
        product: (storeData.modelData[PRODUCTS]
            .find(p => p.id === storeData.stateData.selectedId)) || {},
        supplier:(storeData.modelData[SUPPLIERS]
            .find(s => s.id === storeData.stateData.selectedId)) || {}
    })

    const mapDispatchToProps = dispatch => ({
        cancelCallback: () => dispatch(endEditing()),
        saveCallback: (data) => {
            dispatch((dataType === PRODUCTS ? saveProduct: saveSupplier)(data));
            dispatch(endEditing());
        }
    });

    return connect(mapStateToProps, mapDispatchToProps)(presentationComponent);
}
```

A function that dispatches an action is required as the value for each mapped prop and the implementation can simply invoke the action creator or, in the case of the `saveCallback` prop, create and dispatch multiple actions. The result is that he Save buttons rendered by the editor components invoke a function prop that dispatches actions that update the model data and the state data, as shown in Figure 19-5.

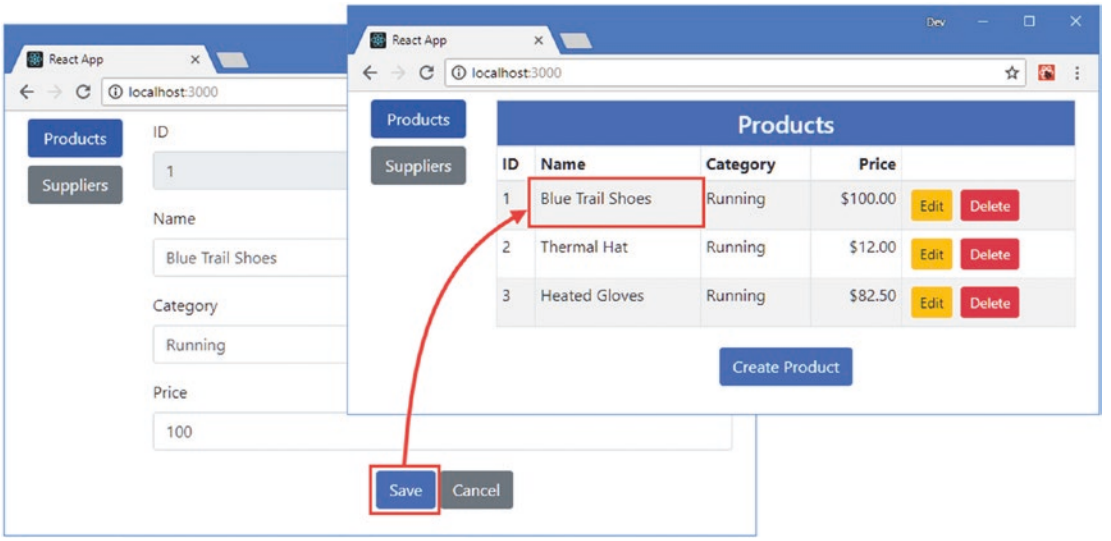

Figure 19-5. *Dispatching multiple actions*

Understanding the Need for References

You may have noticed that I keep track of the object that the user has selected using a combination of `id` property value and data type, like this:

```
...
stateData: {
    editing: false,
    selectedId: -1,
    selectedType: PRODUCTS
}
...
```

The table components pass a complete object to action creators that start the editing process, and you may wonder why I have chosen to only keep an ID reference to the selected object and not store the object itself, especially since this approach requires some additional work to obtain the object for the editor components.

```
...
const mapStateToProps = (storeData) => ({
    editing: storeData.stateData.editing
        && storeData.stateData.selectedType === dataType,
    product: (storeData.modelData[PRODUCTS]
        .find(p => p.id === storeData.stateData.selectedId)) || {},
    supplier:(storeData.modelData[SUPPLIERS]
        .find(s => s.id === storeData.stateData.selectedId)) || {}
})
...
```

The indirection is required because the data store represents the authoritative data source in the application, which may be altered by the selectors that connect the data to components. As a demonstration, I changed the selector for the supplier data in the `TableConnector` connector component, as shown in Listing 19-23.

Listing 19-23. Changing a Selector in the TableConnector.js File in the src Folder

```
import { connect } from "react-redux";
import { startEditingProduct, startEditingSupplier } from "./stateActions";
import { deleteProduct, deleteSupplier } from "./modelActionCreators";
import { PRODUCTS, SUPPLIERS } from "./dataTypes";

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

    const mapStateToProps = (storeData) => ({
        products: storeData.modelData[PRODUCTS],
        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 = {
        editCallback: dataType === PRODUCTS
            ? startEditingProduct : startEditingSupplier,
        deleteCallback: dataType === PRODUCTS ? deleteProduct : deleteSupplier
    }

    return connect(mapStateToProps, mapDispatchToProps)(presentationComponent);
}
```

The new selector matches up the supplier and product data to replace each supplier object's `products` property with one that contains the name, rather than the `id` value, of the corresponding product, as shown in Figure 19-6.

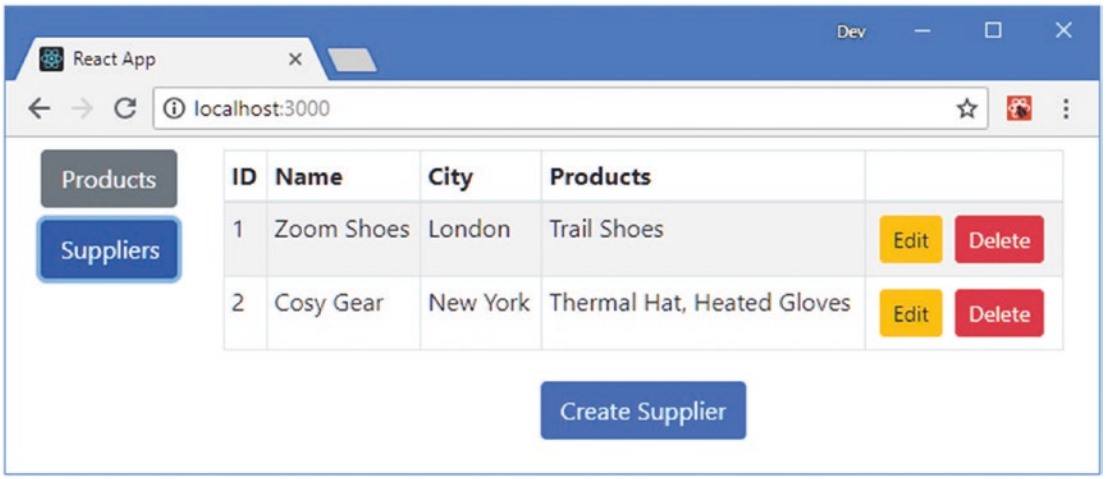

Figure 19-6. *Altering data in a selector*

Transforming data in a selector ensures consistency whenever the same view of the data is required, but it does mean that the connected component is no longer working with the original data from the data store. As a consequence, relying on the data received by one component to drive the behavior of another component can lead to problems, and it is for this reason that I used ID values to keep track of the objects that are selected by the user for editing.

Summary

In this chapter, I created a Redux data store and connected it to the components in the example application. I showed you how to define actions, action creators, reducers, and selectors, and I demonstrated how data store features can be presented to components as props. In the next chapter, I describe the advanced features Redux provides through its API.