CHAPTER 20

Using the Data Store APIs

In Chapter 19, I showed you how to use the Redux and React-Redux packages to create a data store and connect it to the example application. In this chapter, I describe the APIs that both packages provide for advanced use, allowing direct access to the data store and managing the connection between a component and the data features it requires. Table 20-1 puts the data store APIs in context.

Table 20-1. *Putting Data Store APIs in Context*

Question	Answer
What are they?	The Redux and React-Redux packages both define APIs that support advanced use, going beyond the basic techniques described in Chapter 19.
Why are they useful?	These APIs are useful for exploring how data stores work and how components can be connected to them. They can also be used to add features to a data store and to fine-tune an application's use of it.
How are they used?	The Redux API is used directly on the data store object or during its creation. The React-Redux API is used when connecting a component to the data store, either using the `connect` function or using its more flexible `connectAdvanced` alternative.
Are there any pitfalls or limitations?	The APIs described in this chapter require careful thought to ensure that you achieve the desired effect. It is easy to create an application that doesn't properly respond to data store changes or that updates too often.
Are there any alternatives?	You don't have to use the APIs described in this chapter, and most projects will be able to make effective use of a data store using only the basic techniques described in Chapter 19.

A. Freeman, *Pro React 16*, https://doi.org/10.1007/978-1-4842-4451-7_20

Table 20-2 summarizes the chapter.

Table 20-2. *Chapter Summary*

Problem	Solution	Listing
Access the Redux data store API	Use the methods defined by the data store object returned by the `createStore` method	2–4
Observe data store changes	Use the subscribe method	5
Dispatch actions	Use the dispatch method	6
Create a custom connector	Map the props of a component to the data store features	7–8
Add features to the data store	Create a reducer enhancer	9–11
Process actions before they are passed to the reducer	Create a middleware function	12–16
Extend the data store API	Create an enhancer function	17–19
Incorporate a component's props into a data store mapping	Use the optional argument to the connect function	20–24

Preparing for This Chapter

In this chapter, I continue working with the `productapp` project created in Chapter 18 and modified in Chapter 19. No changes are required for this chapter. Open a new command prompt, navigate to the `productapp` folder, and run the command shown in Listing 20-1 to start the development tools.

■ **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 20-1. Starting the Development Tools

```
npm start
```

Once the development tools have started, a new browser window will open and display the content shown in Figure 20-1.

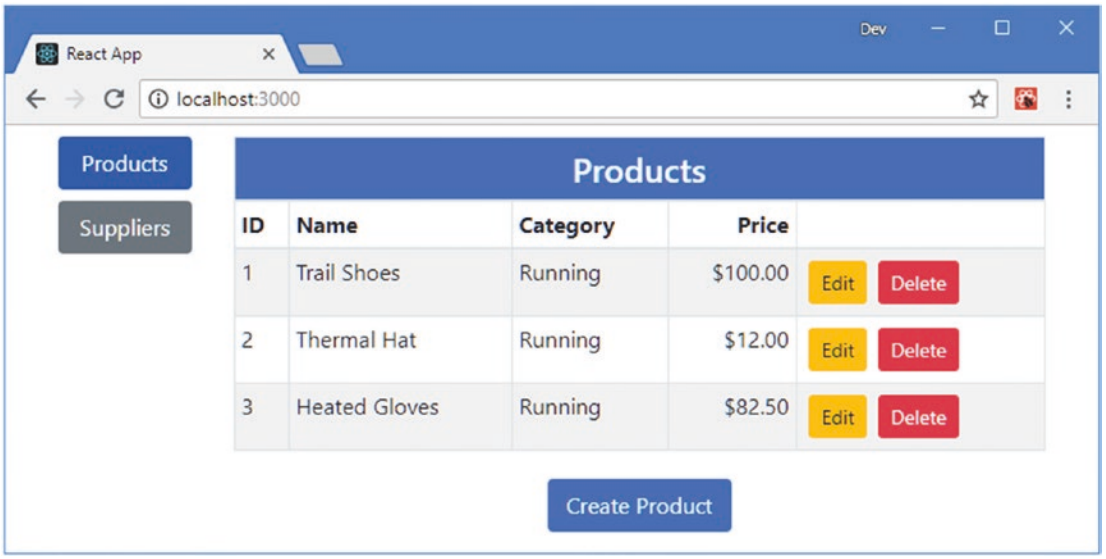

Figure 20-1. *Running the example application*

Using the Redux Data Store API

In most React applications, access to a Redux data store is mediated through the React-Redux package, which maps data store features to props. This is the most convenient way to use Redux, but there is also a full API that provides direct access to data store features, which I describe in the sections that follow, starting with the features that provide access to the data in the store.

In Chapter 19, I used the Redux `createStore` function to create a new data store so that I could pass it as a prop to the `Provider` component from the React-Redux package. The object returned by the `createStore` function can also be used directly through the four methods described in Table 20-3.

Table 20-3. *The Data Store Methods*

Name	Description
`getState()`	This method returns the data from the data store, as described in the "Obtaining the Data Store State" section.
`subscribe(listener)`	This method registers a function that will be invoked each time changes are made to the data store, as described in the "Observing Data Store Changes" section.
`dispatch(action)`	This method accepts an action, typically produced by an action creator, and sends it to the data store so that it can be processed by the reducer, as described in the "Dispatching Actions" section.
`replaceReducer(next)`	This method replaces the reducer used by the data store to process actions. This method is not useful in most project, and middleware provides a more useful mechanism for changing the behavior of the data store.

Obtaining the Data Store State

The getState method returns the data in the data store and allows for the contents of the store to be read. As a demonstration, I added a file called StoreAccess.js to the store folder and used it to define the component shown in Listing 20-2.

Listing 20-2. The Contents of the StoreAccess.js File in the src/store Folder

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

export class StoreAccess extends Component {

    render() {
        return <div className="bg-info">
            <pre className="text-white">
                { JSON.stringify(this.props.store.getState(), null, 2) }
            </pre>
        </div>
    }
}
```

The component receives the data store object as a prop and calls the getState method, which returns the data object for the store. To format the data, I use the JSON.stringify method, which serializes the JavaScript object to JSON and then formats the result so that it can be easily read. In Listing 20-3, I have added a grid layout so that the new component is displayed alongside the rest of the application functionality.

Listing 20-3. Displaying the Data Store Contents in the App.js File in the src Folder

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

export default class App extends Component {

    render() {
        return <div className="container-fluid">
            <div className="row">
                <div className="col-3">
                    <StoreAccess store={ dataStore } />
                </div>
                <div className="col">
                    <Provider store={ dataStore }>
                        <Selector>
                            <ProductDisplay name="Products" />
                            <SupplierDisplay name="Suppliers" />
                        </Selector>
                    </Provider>
```

```
                </div>
            </div>
        </div>
    }
}
```

There can be a lot of data in a store, so I have displayed the JSON text so it will appear in its own column, as shown in Figure 20-2. Don't worry if you can't fit all of the text on-screen because I'll narrow the focus to a subset of the data shortly.

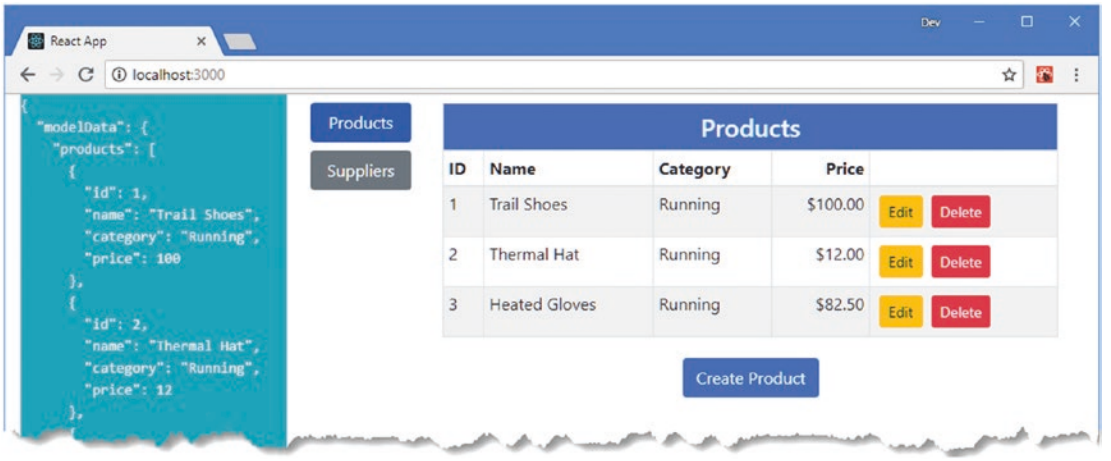

Figure 20-2. *Getting the contents of the data store*

If you examine the data that has been obtained through the `getState` method, you will see that everything is included so that both the contents of the `modelData` and `stateData` properties are available. The segmentation that is applied to reducers doesn't affect the data returned by the `getState` method, which provides access to everything in the data store.

Narrowing the Focus on Specific Data

To make it easier to keep track of the contents of the data store, I am going to focus on a subset of the data returned by the `getState` method, which will allow me to more easily demonstrate other Redux features. In Listing 20-4, I have changed the `StoreAccess` component so that it displays only the first product object and the set of state data variables.

Listing 20-4. Focusing the Data in the StoreAccess.js File in the src/store Folder

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

export class StoreAccess extends Component {

    constructor(props) {
        super(props);
        this.selectors = {
```

```
            product: (storeState) => storeState.modelData.products[0],
            state: (storeState) => storeState.stateData
        }
    }

    render() {
        return <div className="bg-info">
            <pre className="text-white">
                { JSON.stringify(this.selectData(), null, 2) }
            </pre>
        </div>
    }

    selectData() {
        let storeState = this.props.store.getState();
        return Object.entries(this.selectors).map(([k, v]) => [k, v(storeState)])
            .reduce((result, [k, v]) => ({ ...result, [k]: v}), {});
    }
}
```

I have defined a `selectors` object whose property values are functions that select data from the store. The `selectData` method uses the `getState` method to get the data from the data store and invokes each selector function to generate the data that is rendered by the component. (The use of the `entries`, `map`, and `reduce` methods produces an object with the same property names as the `selectors` prop with values that are produced by invoking each selector function.)

The changes to the component select a more manageable section of the data from the store, as shown in Figure 20-3.

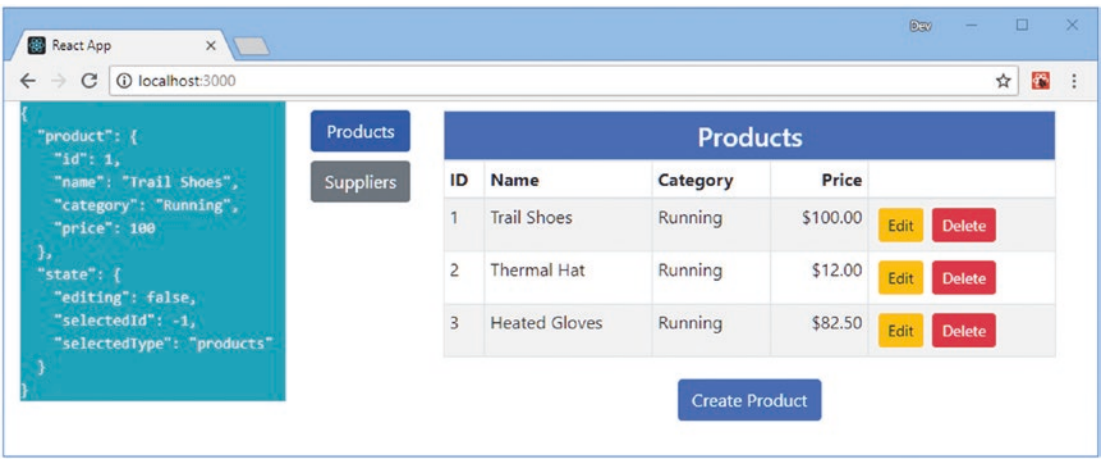

Figure 20-3. *Selecting a subset of the store data*

Observing Data Store Changes

The object returned by the getState method is a snapshot of the data in the store, which isn't automatically updated when the store changes. The usual React change detection features don't work on the store because it is not part of a component's state data. As a consequence, changes made to the data in the store do not trigger a React update.

Redux provides the subscribe method to receive notifications when a change has been made to the data store, which allows the getState method to be called again to get a fresh snapshot of the data. In Listing 20-5, I have used the subscribe method in the StoreAccess component to ensure that the data displayed by the component is kept up-to-date.

Listing 20-5. Subscribing to Change Notifications in the StoreAccess.js File in the src/store Folder

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

export class StoreAccess extends Component {

    constructor(props) {
        super(props);
        this.selectors = {
            product: (storeState) => storeState.modelData.products[0],
            state: (storeState) => storeState.stateData
        }
        this.state = this.selectData();
    }

    render() {
        return <div className="bg-info">
            <pre className="text-white">
                { JSON.stringify(this.state, null, 2) }
            </pre>
        </div>
    }

    selectData() {
        let storeState = this.props.store.getState();
        return Object.entries(this.selectors).map(([k, v]) => [k, v(storeState)])
            .reduce((result, [k, v]) => ({ ...result, [k]: v}), {});
    }

    handleDataStoreChange() {
        let newData = this.selectData();
        Object.keys(this.selectors)
            .filter(key => this.state[key] !== newData[key])
            .forEach(key => this.setState({ [key]: newData[key]}));
    }

    componentDidMount() {
        this.unsubscriber =
            this.props.store.subscribe(() => this.handleDataStoreChange());
    }
```

```
    componentWillUnmount() {
        this.unsubscriber();
    }
}
```

I subscribe to updates in the `componentDidMount` method. The result from the `subscribe` method is a function that can be used to unsubscribe from updates, which I invoke in the `componentWillUnmount` method.

The argument to the `subscribe` method is a function that will be invoked when there have been changes to the data store. No argument is provided to the function, which is just a signal that there has been a change and that the `getState` method can be used to get the new contents of the data store.

Redux doesn't provide any information about what data has been changed and so I defined the `handleStoreChange` method to inspect the data obtained by each of the selector functions to see whether the data that the component renders has changed. I use the component state data feature to keep track of the displayed data and use the `setState` method to trigger updates. It is important to perform a state change only when the data displayed by the component has changed; otherwise, an update would be performed for every change made to the data store.

To see the effect of a change, click the Edit button for the Trail Shoes product, make a change to the Name field, and click the Save button. As you go through this process, the data displayed by the `StoreAccess` component will reflect the changes in the data store, as shown in Figure 20-4.

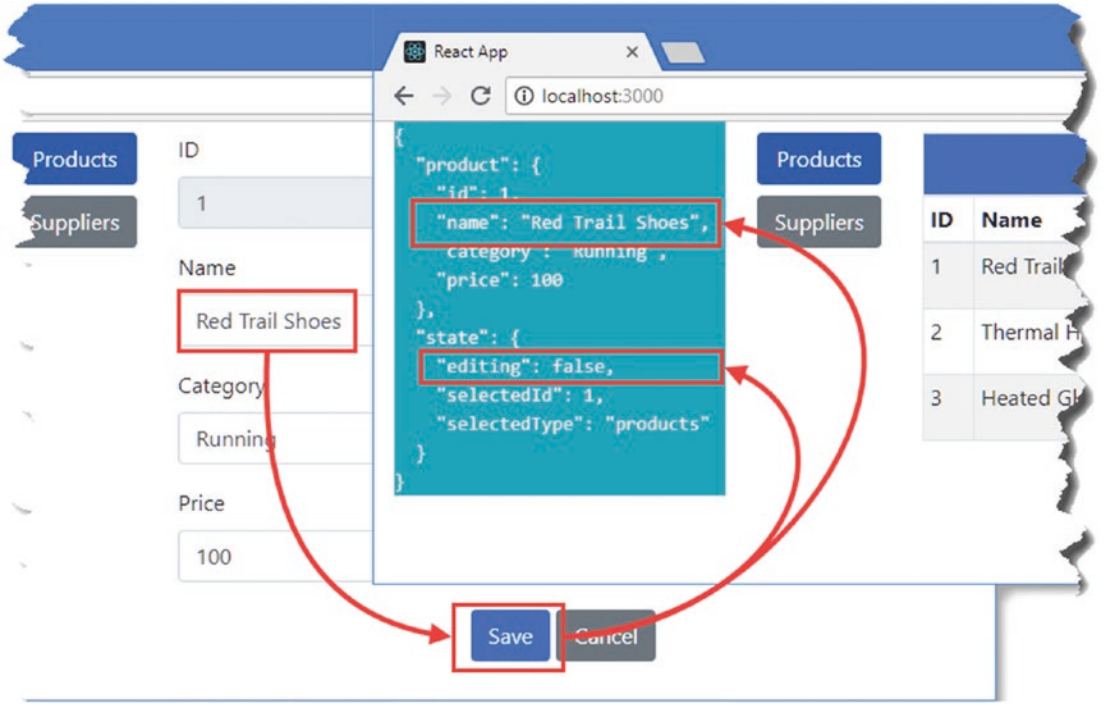

Figure 20-4. *Receiving change notifications from the data store*

Dispatching Actions

Actions can be dispatched using the `dispatch` method, which is the same `dispatch` to which the React-Redux package provided access in Chapter 19 when I needed to dispatch multiple actions.

As I explained in Chapter 19, actions are created through action creators. In Listing 20-6, I added a `button` to the `StoreAccess` component that uses an action creator to obtain an action object, which is then sent to the data store using the `dispatch` method.

Listing 20-6. Dispatching an Action in the StoreAccess.js File in the src/store Folder

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

export class StoreAccess extends Component {

    constructor(props) {
        super(props);
        this.selectors = {
            product: (storeState) => storeState.modelData.products[0],
            state: (storeState) => storeState.stateData
        }
        this.state = this.selectData();
    }

    render() {
        return <React.Fragment>
            <div className="text-center">
                <button className="btn btn-primary m-1"
                    onClick={ this.dispatchAction }>
                        Dispatch Action
                </button>
            </div>
            <div className="bg-info">
                <pre className="text-white">
                    { JSON.stringify(this.state, null, 2) }
                </pre>
            </div>
        </React.Fragment>
    }

    dispatchAction = () => {
        this.props.store.dispatch(startCreatingProduct())
    }

    selectData() {
        let storeState = this.props.store.getState();
        return Object.entries(this.selectors).map(([k, v]) => [k, v(storeState)])
            .reduce((result, [k, v]) => ({ ...result, [k]: v}), {});
    }
```

```
    handleDataStoreChange() {
        let newData = this.selectData();
        Object.keys(this.selectors)
            .filter(key => this.state[key] !== newData[key])
            .forEach(key => this.setState({ [key]: newData[key]}));
    }

    componentDidMount() {
        this.unsubscriber =
            this.props.store.subscribe(() => this.handleDataStoreChange());
    }

    componentWillUnmount() {
        this.unsubscriber();
    }
}
```

The button responds to the click event by invoking the `dispatchAction` method, which invokes the `startCreatingProduct` action creator and passes the result to the data store's `dispatch` method. The result is that clicking the button toggles the display to show the editor, as shown in Figure 20-5.

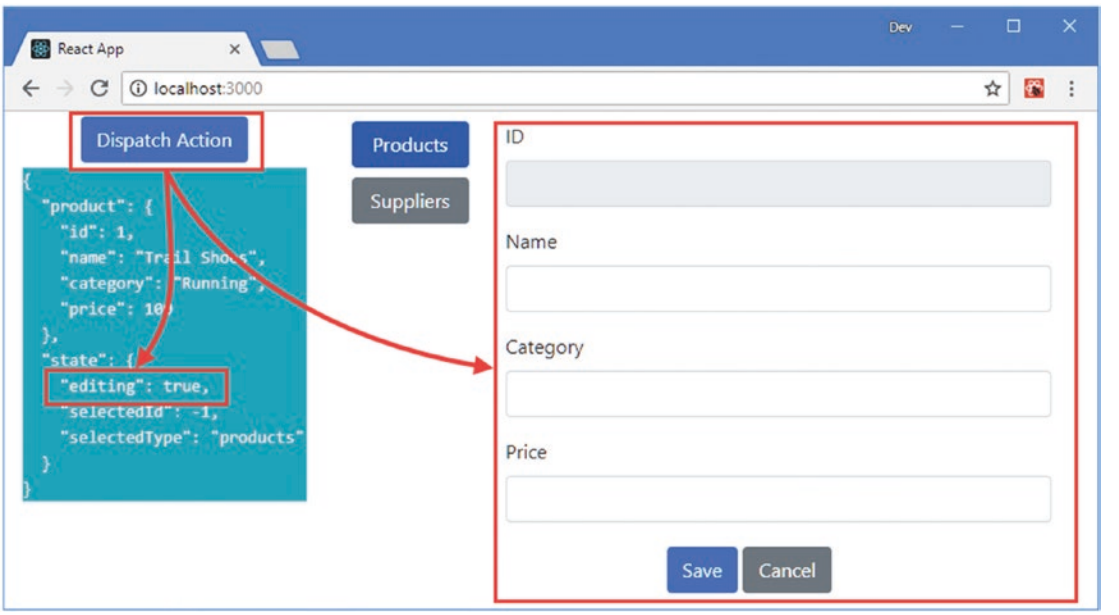

Figure 20-5. *Dispatching an action*

Creating a Connector Component

The ability to get the current data from the store, receive change notifications, and dispatch actions provides all the features to create a basic connector component that provides a rudimentary equivalent to the React-Redux package I used in the example application. To create the facility to connect components to the data store through the Redux API, I added a file called `CustomConnector.js` to the `store` folder and added the code shown in Listing 20-7.

■ **Caution** I don't recommend using a custom connector in real projects. The React-Redux package has additional features and has been thoroughly tested, but combining core React features with the Redux data store API provides a useful example of how advanced features can be created.

Listing 20-7. The Contents of the CustomConnector.js File in the src/store Folder

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

export const CustomConnectorContext = React.createContext();

export class CustomConnectorProvider extends Component {

    render() {
        return <CustomConnectorContext.Provider value={ this.props.dataStore }>
            { this.props.children }
        </CustomConnectorContext.Provider>
    }
}

export class CustomConnector extends React.Component {
    static contextType = CustomConnectorContext;

    constructor(props, context) {
        super(props, context);
        this.state = this.selectData();
        this.functionProps = Object.entries(this.props.dispatchers)
            .map(([k, v]) => [k, (...args) => this.context.dispatch(v(...args))])
            .reduce((result, [k, v]) => ({...result, [k]: v}), {})
    }

    render() {
        return  React.Children.map(this.props.children, c =>
            React.cloneElement(c, { ...this.state, ...this.functionProps }))
    }

    selectData() {
        let storeState = this.context.getState();
        return Object.entries(this.props.selectors).map(([k, v]) =>
                [k, v(storeState)])
            .reduce((result, [k, v]) => ({ ...result, [k]: v}), {});
    }

    handleDataStoreChange() {
        let newData = this.selectData();
        Object.keys(this.props.selectors)
            .filter(key => this.state[key] !== newData[key])
            .forEach(key => this.setState({ [key]: newData[key]}));
    }
```

```
    componentDidMount() {
        this.unsubscriber =
            this.context.subscribe(() => this.handleDataStoreChange());
    }

    componentWillUnmount() {
        this.unsubscriber();
    }
}
```

I have used the context API to make the data store available through a `CustomConnectorProvider` component, which is received by a `CustomConnector` component that receives selector and action creator props. The selector props are processed to set the state of the component so that changes can be detected and processed, while the action creator props are wrapped in the `dispatch` method so they can be invoked as function props by the connected child components. To demonstrate the custom connector, I added the content shown in Listing 20-8 to the `App` component.

Listing 20-8. Using the Custom Connector in the App.js File in the src Folder

```
import React, { Component } from "react";
import { Provider } from "react-redux";
import dataStore, { deleteProduct } from "./store";
import { Selector } from "./Selector";
import { ProductDisplay } from "./ProductDisplay";
import { SupplierDisplay } from "./SupplierDisplay";
import { StoreAccess } from "./store/StoreAccess";
import { CustomConnector, CustomConnectorProvider } from "./store/CustomConnector";
import { startEditingProduct } from "./store/stateActions";
import { ProductTable } from "./ProductTable";

const selectors = {
    products: (store) => store.modelData.products
}

const dispatchers = {
    editCallback: startEditingProduct,
    deleteCallback: deleteProduct
}

export default class App extends Component {

    render() {
        return <div className="container-fluid">
            <div className="row">
                <div className="col-3">
                    <StoreAccess store={ dataStore } />
                </div>
                <div className="col">
                    <Provider store={ dataStore }>
                        <Selector>
                            <ProductDisplay name="Products" />
                            <SupplierDisplay name="Suppliers" />
```

```
                    </Selector>
                </Provider>
            </div>
        </div>
        <div className="row">
            <div className="col">
                <CustomConnectorProvider dataStore={ dataStore }>
                    <CustomConnector selectors={ selectors }
                            dispatchers={ dispatchers }>
                        <ProductTable/>
                    </CustomConnector>
                </CustomConnectorProvider>
            </div>
        </div>
    </div>
  }
}
```

I don't want to replace the existing application content, so I added a row to the Bootstrap grid layout and used it to display a `ProductTable` component that is connected to the data store using the components defined in Listing 20-8. The `CustomConnector` component is defined as the child of the `CustomConnectorProvider` component for brevity, and the effect is to map the selectors and action creators to props that are presented to the `ProductTable` component. The result is that the application displays a second table of products, both of which reflect changes to the data they display, as shown in Figure 20-6.

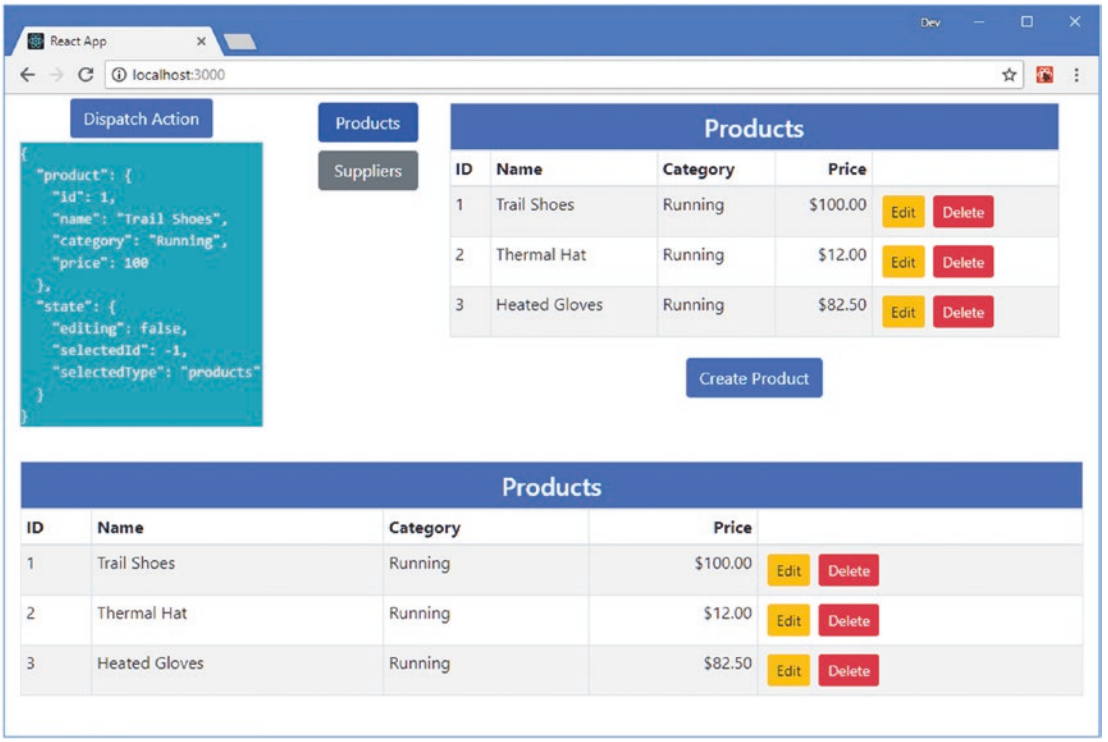

Figure 20-6. *Using a custom data store connector*

Enhancing Reducers

As I explained in Chapter 19, a reducer is a function that processes an action and updates the data store. A *reducer enhancer* is a function that accepts one or more normal reducers and uses them to add additional features to the data store.

Redux has no special awareness when a reducer enhancer is used because the result appears just like a regular reducer and is passed to the `createStore` method in just the same way, as this statement from the `index.js` file in the `src/store` folder shows:

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

The `combineReducers` function is a reducer enhancer that comes built-in to Redux and that I used in Chapter 19 to keep the reducer logic for the model and state data separate.

Reducer enhancers are useful because they receive the actions before they are processed, which means they can alter actions, reject them, or handle them in special ways, such as processing them using multiple reducers, which is what the `combineReducers` function does.

To demonstrate a reducer enhancer, I added a file called `customReducerEnhancer.js` to the `store` folder and added the code shown in Listing 20-9.

Listing 20-9. The Contents of the customReducerEnhancer.js File in the src/store Folder

```
import { initialData } from "./initialData";

export const STORE_RESET = "store_clear";

export const resetStore = () => ({ type: STORE_RESET });

export function customReducerEnhancer(originalReducer) {

    let intialState = null;

    return (storeData, action) => {
        if (action.type === STORE_RESET && initialData != null) {
            return intialState;
        } else {
            const result = originalReducer(storeData, action);
            if (intialState == null) {
                intialState = result;
            }
            return result;
        }
    }
}
```

The customReducerEnhancer function accepts a reducer as its argument and returns a new reducer function that can be used by the data store. The enhancer function makes a note of the initial state of the data store, which is obtained by the first action that is sent to the reducers. A new action type, STORE_RESET, causes the enhancer function to return the initial data store state, which has the effect of resetting the data store. All other actions are passed on to the normal reducer. To help implement the store reset feature, Listing 20-9 defines a resetStore action creator function. In Listing 20-10, I have applied the reducer enhancer to the data store.

Listing 20-10. Applying a Reducer Enhancer in the index.js File in the src/store Folder

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

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

export default createStore(enhancedReducer);

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

Reducer enhancers can be combined. In this listing, I use the reducer created by the combineReducers function as the argument to the customReducerEnhancer function. In Listing 20-11, I used the resetStore action creator to create an action when the user clicks the button rendered by the StoreAccess component.

Listing 20-11. Changing Actions in the StoreAccess.js File in the src/store Folder

```
import React, { Component } from "react";
//import { startCreatingProduct } from "./stateActions";
import { resetStore } from "./customReducerEnhancer";

export class StoreAccess extends Component {

    constructor(props) {
        super(props);
        this.selectors = {
            product: (storeState) => storeState.modelData.products[0],
            state: (storeState) => storeState.stateData
        }
        this.state = this.selectData();
    }
```

```
    render() {
        return <React.Fragment>
            <div className="text-center">
                <button className="btn btn-primary m-1"
                    onClick={ this.dispatchAction }>
                        Dispatch Action
                </button>
            </div>
            <div className="bg-info">
                <pre className="text-white">
                    { JSON.stringify(this.state, null, 2) }
                </pre>
            </div>
        </React.Fragment>
    }

    dispatchAction = () => {
        this.props.store.dispatch(resetStore())
    }

    // ...other methods omitted for brevity...
}
```

The effect of the enhancer is that the application's state and model data are reset when the user clicks the Dispatch Action button, with the result that any changes are discarded, as shown in Figure 20-7.

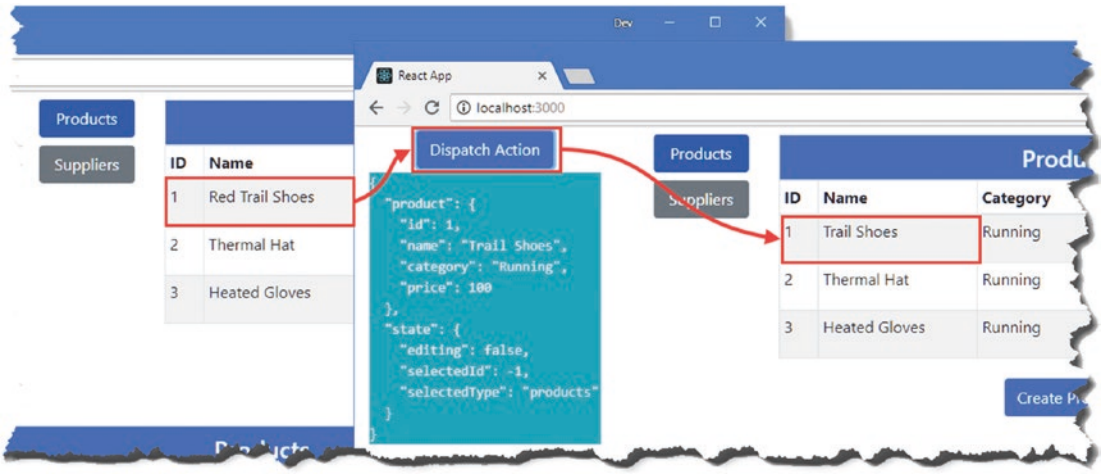

Figure 20-7. *Resetting the data in the store*

Using Data Store Middleware

Redux provides support for data store *middleware*, which are functions that receive actions after they have been passed to the `dispatch` method and before they reach the reducer, allowing them to be intercepted, transformed, or processed in some other way. The most common uses for middleware are to add support for actions that perform asynchronous tasks and to wrap actions in functions so they can be dispatched conditionally or in the future.

■ **Note** There are middleware packages available that address common project needs and that you should consider instead of writing custom code. The `redux-promise` package supports asynchronous actions (see `https://github.com/redux-utilities/redux-promise`), and the `redux-thunk` package supports action creators that return functions (see `https://github.com/reduxjs/redux-thunk`). I find, however, that neither of these packages works in just the way I require, so I prefer to create my own middleware.

To demonstrate the use of middleware, I added a file called `multiActionMiddleware.js` to the `src/store` folder and added the code shown in Listing 20-12.

Listing 20-12. The Contents of the multiActionMiddleware.js File in the src/store Folder

```
export function multiActions({dispatch, getState}) {
    return function receiveNext(next) {
        return function processAction(action) {
            if (Array.isArray(action)) {
                action.forEach(a => next(a));
            } else {
                next(action);
            }
        }
    }
}
```

Middleware is expressed as a set of functions that return other functions, and to make it easier to understand, I have used the `function` keyword in Listing 20-12. The outer function, `multiActions`, is called when the middleware is registered with the data store, and it receives the data store's `dispatch` and `getState` method, like this:

```
...
export function multiActions({dispatch, getState}) {
...
```

This provides the middleware with the ability to dispatch actions and to get the data currently in the data store. A data store can use multiple middleware components; actions are passed from one to the next in a chain, and then they are passed to the data store's `dispatch` method. The job of the `multiActions` function is to return a function that will be invoked once the middleware chain has been assembled and that provides the next middleware component in the chain.

```
...
export function multiActions({dispatch, getState}) {
    return function receiveNext(next) {
...
```

A middleware component will usually process an action and then pass it on to the next component in the chain by invoking the next function.

The result of the receiveNext function is to return the innermost function, which is invoked when an action has been dispatched to the data store and which I have called processAction in Listing 20-12.

```
...
export function multiActions({dispatch, getState}) {
    return function receiveNext(next) {
        return function processAction(action) {
...
```

This function is able to change or replace the action object before it is passed on to the next middleware component. It is also possible to short-circuit the chain by invoking the dispatch method received by the outer function or do nothing at all (in which case the action won't be processed by the data store). The middleware component that I defined in Listing 20-12 checks to see whether the action is an array, in which case it passes on each object contained in the array to the next middleware component for processing.

Defining the nested functions helps explain how a middleware component is defined, but the convention is to use fat arrow functions, as shown in Listing 20-13.

Listing 20-13. Using Fat Arrow Functions in the multiActionMiddleware.js File in the src/store Folder

```
export const multiActions = ({dispatch, getState}) => next => action => {
    if (Array.isArray(action)) {
        action.forEach(a => next(a));
    } else {
        next(action);
    }
}
```

This is the same functionality as Listing 20-12 but expressed more concisely. Redux provides an applyMiddlware function that is used to create the middleware chain for use with the data store and which I used in Listing 20-14 to add the new middleware component to the application.

Listing 20-14. Registering Middleware in the index.js File in the src/store Folder

```
import { createStore, combineReducers, applyMiddleware } from "redux";
import modelReducer from "./modelReducer";
import stateReducer from "./stateReducer";
import { customReducerEnhancer } from "./customReducerEnhancer";
import { multiActions } from "./multiActionMiddleware";

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

```
export default createStore(enhancedReducer, applyMiddleware(multiActions));

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

The middleware function is passed as an argument to the Redux `applyMiddleware` function, whose result is then passed as an argument to the `createStore` function.

■ **Tip** Multiple middleware functions can be passed as separate arguments to the `applyMiddleware` function, which will chain them together in the order they have been specified.

Now that the data store can process arrays of actions, I can define action creators that generate more complex results and that allow connector components to be expressed more simply. I added a file called `multiActionCreators.js` in the `src/store` folder and used it to define the action creator shown in Listing 20-15.

Listing 20-15. The Contents of the multiActionCreators.js File in the src/store Folder

```
import { PRODUCTS } from "./dataTypes";
import { saveProduct, saveSupplier } from "./modelActionCreators";
import { endEditing } from "./stateActions";

export const saveAndEndEditing = (data, dataType) =>
    [dataType === PRODUCTS ? saveProduct(data) : saveSupplier(data), endEditing()];
```

It is not a requirement to put such action creators in a separate file, but this creator mixes actions that affect model and state data, and I prefer to keep them apart. The `saveAndEndEditing` action receives a data object and type and uses it to produce an array of actions that will be received by the middleware and dispatched in sequence. In Listing 20-16, I have replaced the statements in the `EditorConnector` component that used the `dispatch` method directly to send multiple events.

Listing 20-16. Dispatching Multiple Actions in 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";
import { saveAndEndEditing } from "./multiActionCreators";

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: (data) => saveAndEndEditing(data, dataType)
    }

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

There is no change in the behavior of the application, but the code is more concise and easier to understand.

Enhancing the Data Store

Most projects will not need to modify the behavior of the data store, and if they do, the middleware features described in the previous chapter will be sufficient. But if middleware doesn't provide sufficient flexibility, a more advanced option is to use an *enhancer function*, which is a function that takes responsibility for creating the data store object and that can provide wrappers around the standard methods or define new ones.

The `applyMiddleware` function I used earlier is an enhancer function. This function replaces the data store's `dispatch` method so that it can channel actions through its chain of middleware components before they are passed to the reducer.

To demonstrate the use of enhancer functions, I am going to add a new method to the data store that dispatches actions asynchronously. I added a file called `asyncEnhancer.js` to the `src/store` folder and added the code shown in Listing 20-17.

Listing 20-17. The Contents of the asyncEnhancer.js File in the src/store Folder

```
export function asyncEnhancer(delay) {
    return function(createStoreFunction) {
        return function(...args) {
            const store = createStoreFunction(...args);
            return {
                ...store,
                dispatchAsync: (action) => new Promise((resolve, reject) => {
                    setTimeout(() => {
                        store.dispatch(action);
                        resolve();
                    }, delay);
                })
            };
        }
    }
}
```

The enhancer dispatches actions asynchronously, returning a `Promise` that is resolved once the action has been dispatched. There are currently no tasks in the example application that need asynchronous work, and so I have introduced a delay before dispatching actions to similar a background activity.

This is another Redux feature that requires a nested set of functions, which I defined using the `function` keyword so I can explain how they fit together. The outer function is invoked when the enhancer is applied to the data store and provides an opportunity to receive arguments that will configure the enhancer's behavior. The outermost function in Listing 20-17 receives the length of the delay that will be applied before an action is dispatched.

```
...
export function asyncEnhancer(delay) {
...
```

Now it gets more complex: the result of the outer function is a function that receives the `createStore` function. The word *function* appears too many times for the previous sentence to make immediate sense, and it is worth unpacking what happens.

To give enhancers complete control, Redux lets them replace the `createStore` function with a custom alternative. But most reducers will just need to add features to the standard data store and so Redux provides the existing `createStore` function.

```
...
export function asyncEnhancer(delay) {
    return function(createStoreFunction) {
...
```

When the enhancer is applied, this function will be invoked, and the result will be used to replace the standard `createStore` function, which takes us to the innermost function in Listing 20-17, which is the one that does all the work.

```
...
return function(...args) {
    const store = createStoreFunction(...args);
    return {
        ...store,
        dispatchAsync: (action) => new Promise((resolve, reject) => {

            // ...statements omitted for brevity...

        })
    };
}
...
```

When a data store is created, Redux invokes the function provided by the enhancer and uses the result as the data store object, ensuring that any additional features are available to the rest of the application. In this case, the enhancer uses the standard `createStore` function and then adds a `dispatchAsync` method to the result. The new method receives an action and dispatches it after a delay. Using the `function` keyword makes it easier to see the relationship between the nested functions, but enhancers are typically expressed using fat arrow functions, as shown in Listing 20-18. This is the same functionality but expressed more concisely.

Listing 20-18. Using Arrow Functions in the asyncEnhancer.js File in the src/store Folder

```
export const asyncEnhancer = delay => createStoreFunction => (...args) => {
    const store = createStoreFunction(...args);
    return {
        ...store,
        dispatchAsync: (action) => new Promise((resolve, reject) => {
            setTimeout(() => {
                store.dispatch(action);
                resolve();
```

```
            }, delay);
        })
    };
}
```

Applying the Enhancer

The standard `createStore` function can accept only a single enhancer function, and I am already using the `applyMiddleware` enhancer. Fortunately, reducer functions can be composed so that the result from one enhancer can be passed to another. To simplify the process of combining functions, Redux provides the `compose` function, which I have used in Listing 20-19 to apply the new enhancer to the data store.

Listing 20-19. Adding an Enhancer 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";

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

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

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

The result from the `compose` function is passed to `createStore`, and both enhancers are applied to the data store, adding middleware and the new `dispatchAsync` method. In Listing 20-20, I updated the `StoreAccess` component so that it uses the enhanced data store method when it dispatches an action and disables the `button` element until the background task is complete.

Listing 20-20. Using the Enhanced Data Store in the StoreAccess.js File in the src/store Folder

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

export class StoreAccess extends Component {

    constructor(props) {
        super(props);
        this.selectors = {
```

```
            product: (storeState) => storeState.modelData.products[0],
            state: (storeState) => storeState.stateData
        }
        this.state = this.selectData();
        this.buttonRef = React.createRef();
    }

    render() {
        return <React.Fragment>
            <div className="text-center">
                <button className="btn btn-primary m-1" ref={ this.buttonRef }
                    onClick={ this.dispatchAction }>
                        Dispatch Action
                </button>
            </div>
            <div className="bg-info">
                <pre className="text-white">
                    { JSON.stringify(this.state, null, 2) }
                </pre>
            </div>
        </React.Fragment>
    }

    dispatchAction = () => {
        this.buttonRef.current.disabled = true;
        this.props.store.dispatchAsync(resetStore())
            .then(data => this.buttonRef.current.disabled = false);
    }

    // ...other methods omitted for brevity...
}
```

The result is that clicking the button dispatches the action, which will be processed after a two-second display. The component receives a `Promise` when it dispatches the action, which is resolved once it has been dispatched, allowing the component to enable the button element again, as shown in Figure 20-8.

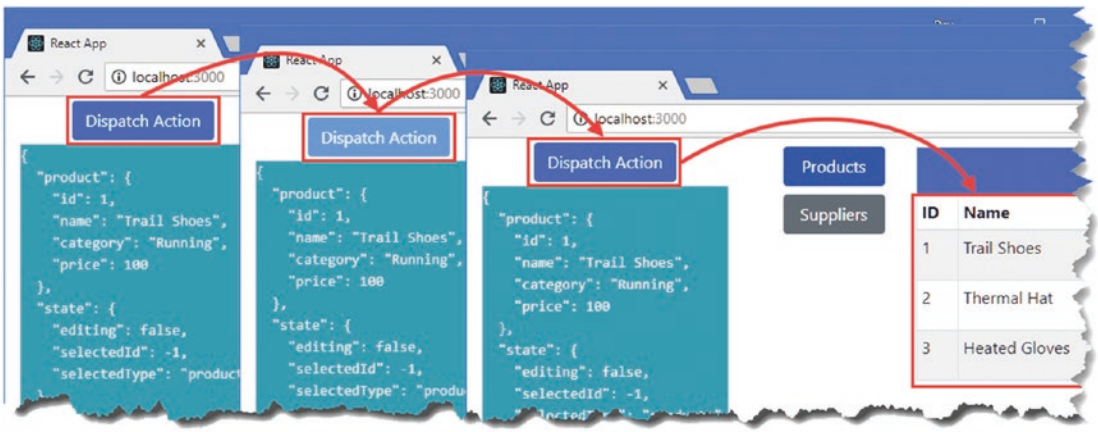

Figure 20-8. *Using an enhanced data store*

Using the React-Redux API

The previous sections demonstrated that you can work directly with the Redux API to connect components to the data store. For most projects, however, it is simpler and easier to use the React-Redux package as demonstrated in Chapter 19. In the sections that follow, I describe the advanced options that the React-Redux package provides for configuring how components are connected to the data store.

Advanced Connect Features

The `connect` method is typically used with two arguments, which select the data props and the function props, like this statement from the `TableConnector` component:

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

The `connect` function can accept some additional arguments for advanced features and can receive arguments expressed in different ways. In this section, I explain the options for using the arguments you are already familiar with and introduce the new arguments and demonstrate their use.

Mapping Data Props

The first argument to the `connect` function selects the data from the store for the component's data props. Usually, the selectors are defined as a function that receives the value from the store's `getState` method and returns an object whose properties correspond to the prop names. The selector function is invoked when there is a change to the data store, and the higher-order component created by the `connect` function uses the `shouldComponentUpdate` lifecycle method (described in Chapter 13) to see whether any the changed values require the connector component to update.

The selection of data values is flexible and isn't just about mapping data store properties to props. In the `TableConnector` component, for example, I used the selector function to map data values from different parts of the store, like this:

```
...
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)
    }))
})
...
```

The selector function can also be expressed with a second argument, which is used to receive the props provided for the connector component by its parent. This allows for the component's props to be used in selecting data and ensures that the selector function will be reevaluated when the component's props change as well as when there are changes to the data store. Listing 20-21 demonstrates the use of the additional argument.

Listing 20-21. Using an Additional Selector Argument 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";

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

    const mapStateToProps = (storeData, ownProps) => {
        if (!ownProps.needSuppliers) {
            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 = {
        editCallback: dataType === PRODUCTS
            ? startEditingProduct : startEditingSupplier,
        deleteCallback: dataType === PRODUCTS ? deleteProduct : deleteSupplier
    }

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

One problem with creating connectors that are applied to multiple components is that too much data is selected, which can lead to unnecessary updates when a change in the data store affects a prop that is used by one component but not another. The `TableConnector` component is a connector for both tables of product and supplier data, but only the supplier data requires the `suppliers` prop to be mapped from the data store. For the product table, not only does this mean that the computation of the `suppliers` property is wasted, but it causes updates when data that it does not display is changed.

The additional argument—which is conventionally named `ownProps`—allows each instance of a connector component to be customized through the standard React prop features. In Listing 20-21, I used the `ownProps` argument to decide which props are mapped to the data store based on the value of a prop named `needSuppliers` applied to the connector component. If the value is `true`, then a `suppliers` prop is mapped to the data store and the `products` prop is mapped otherwise.

In Listing 20-22, I have added the `needSuppliers` prop to the `ConnectedTable` component rendered by the `SupplierDisplay` component, which will ensure that it maps the data that its presentation component requires. The corresponding `ConnectedTable` component rendered by the `ProductDisplay` component doesn't have the `needSuppliers` prop and won't receive the data from the store.

Listing 20-22. Adding a Prop in the SupplierDisplay.js File in the src Folder

```
...
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 needSuppliers={ true } />
                    <div className="text-center">
                        <button className="btn btn-primary m-1"
                            onClick={ this.props.createSupplier }>
                                Create Supplier
                        </button>
                    </div>
                </div>
            }
        }
    })
...
```

There is no difference in the behavior of the application, but behind the scenes, each presentation component that is connected to the data store by the `ConnectedTable` components uses different props.

THE DANGERS OF PREMATURE OPTIMIZATION

Don't worry too much about optimizing updates until you find you have a performance problem. Almost all optimization adds complexity to a project, and you may find that the performance penalty incurred by your unoptimized code is not discernable or not enough of a problem to worry about. It is easy to get bogged down in trying to optimize away problems that may not exist, and a better approach is to write the clearest, simplest code you can and then optimize the parts that don't behave the way you require.

Mapping Function Props

As I explained in Chapter 19, the second `connect` argument, which maps between function props, can be specified as an object or a function. When an object is provided, the value of each of the object's properties is assumed to be an action creator function and is automatically wrapped in the `dispatch` method and mapped to a function prop. When a function is provided, the function is given the `dispatch` method and is responsible for using it to create function prop mappings.

■ **Tip** You can omit the second argument to the `connect` function, in which the `dispatch` method is mapped to a prop, also named `dispatch`, which allows the component to create actions and dispatch them directly.

If you specify a function, then you can also choose to receive the connector components props, as described in the previous section. This allows for advanced components to receive direction from their parent about the set of function props that are mapped to the data store. In Listing 20-23, I have used a function to configure function props and defined a second argument to receive the component's props.

Listing 20-23. Using Props 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";

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

    const mapStateToProps = (storeData, ownProps) => {
        if (!ownProps.needSuppliers) {
            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) => {
        if (!ownProps.needSuppliers) {
            return {
                editCallback: (...args) => dispatch(startEditingProduct(...args)),
                deleteCallback: (...args) => dispatch(deleteProduct(...args))
            }
        } else {
            return {
                editCallback: (...args) => dispatch(startEditingSupplier(...args)),
                deleteCallback: (...args) => dispatch(deleteSupplier(...args))
            }
        }
    }

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

Merging Props

The connect function accepts a third argument that is used to compose the props before they are passed to the presentation component. This argument, known as mergeProps, is a function that receives the mapped data props, the function props, and the connected components props and returns an object that merges them into the object used as the props for the presentation component.

By default, the props are composed starting with the props received from the parent, which are then combined with the data props and function props. This means a prop received from the parent will be replaced with a mapped data prop that has the same name, and both will be replaced if there is a mapped function prop with the same name. The mergeProps function can be used to change the priority when there is a name clash, as well as binding actions so they are dispatched using values received as props from the parent. Listing 20-24 shows how props can be merged explicitly using the mergeProps argument.

Listing 20-24. Merging Props in the EditorConnector.js File in the src/store Folder

```
import { connect } from "react-redux";
import { endEditing } from "./stateActions";
import { PRODUCTS, SUPPLIERS  } from "./dataTypes";
import { saveAndEndEditing } from "./multiActionCreators";

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: (data) => saveAndEndEditing(data, dataType)
    }

    const mergeProps = (dataProps, functionProps, ownProps) =>
        ({ ...dataProps, ...functionProps, ...ownProps })

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

The mergeProps function in Listing 20-24 combines the properties from each prop's object. The properties are copied from the objects in the order specified, which means the function copies from ownProps last and also means props received from the parent will be used when there are props with the same name.

Setting Connection Options

The final argument to the `connect` method is conventionally named `options` and is an object used to configure the connection to the data store. The `options` object can be defined with properties whose names are shown in Table 20-4.

Table 20-4. *The Options Object Property Names*

Name	Description
`pure`	By default, a connector component will be updated only when its own props change or when one of the selected values from the data store changes, which allows the higher-order component (HOC) created by `connect` to prevent updates when no prop or data change has been made. Setting this property to `false` indicates that the connector component may rely on other data, and the HOC will not try to prevent updates. The default values for this property is `true`.
`areStatePropsEqual`	This function is used to override the default equality comparison for the `mapStateToProps` values to minimize updates when the `pure` property is `true`.
`areOwnPropsEqual`	This function is used to override the default equality comparison for the `mapDispatchToProps` values to minimize updates when the `pure` property is `true`.
`areMergedPropsEqual`	This function is used to override the default equality comparison for the `mergeProps` values to minimize updates when the `pure` property is `true`.
`areStatesEqual`	This function is used to override the default equality comparison for the entire component state to minimize updates when the `pure` property is `true`.

Summary

In this chapter, I described the APIs provided by Redux and the React-Redux package and demonstrated how they can be used. I showed you how to connect a component directly to the data store using the Redux API, how to enhance the data store and its reducers, and how to define middleware components. I also demonstrated the advanced options available when using the React-Redux package, which can be used to manage a component's connection to the data store. In the next chapter, I introduce URL routing to the example application.