MANNING

Isomorphic Web Applications

Universal development with React

SAMPLE CHAPTER

Elyse Kolker Gordon

Isomorphic Web Applications
Universal development with React
by Elyse Kolker Gordon

Sample Chapter 6

brief contents

6 Redux

This chapter covers

- Managing your application state with Redux
- Implementing Redux as an architecture pattern
- Managing your application state with actions
- Enforcing immutability with reducers
- Applying middleware for debugging and asynchronous calls
- Using Redux with React

Redux is a library that provides an architecture for writing your business logic. With React apps, you can handle much of your application state within your root components. But as your application grows, you end up with a complex set of callbacks that need to be passed down to all the children in order to manage application state updates. Redux provides an alternative for storing your application state by doing the following:

- Dictating a clear line of communication between your view and your business logic

- Allowing your view to subscribe to the application state so it can update each time the state updates
- Enforcing an immutable application state

DEFINITION Immutable objects are read-only. To update an immutable object, you need to clone it. When you change an object in JavaScript, it affects all references to that object. This means mutable changes can have unintended side effects. By enforcing immutability in your store, you prevent this from happening in your app.

6.1 Introduction to Redux

Redux dictates a single-directional flow of writing application state updates into a single root store. The store can be a simple or a complex JavaScript object depending on your app's requirements. Redux handles wiring updates into the store. It also handles any subscribers to the store and notifies them of updates to the store object.

DEFINITION The Redux store is a *singleton* (only one instance per app) object that holds all your application state. The store can be passed into your view in order to display and update your app.

Redux can be hooked up to any view, but it works especially well with React. React's top-down flow of props and state through nested components work well with Redux's single-direction state update flow.

NOTE React state isn't the same as Redux application state! React state is localized to each component in your app. It can be updated and affected within the React lifecycle. It should be used infrequently but is often needed in components that handle user input and sometimes in container components. Chapter 3 explains React state in more detail.

6.1.1 Getting started with notifications example app

The code for this chapter can be found at https://github.com/isomorphic-dev-js/chapter6-redux. All the code is provided on the master branch, or you can follow along and build it yourself. To run the app:

```
$ npm install
$ npm start
```

Then the app will be running at http://localhost:3000.

You'll be building a notifications app that displays messages in three states (Error, Warning, or Success). The idea is that the app receives updates from various paging apps, continuous integration build tools, and other systems (think GitHub, TravisCI, CircleCI, VictorOps, PagerDuty, and so forth). It then displays the notifications in the appropriate shelf. The app also has a settings panel that can be updated and a debug panel that lets you dispatch notifications for testing. Figure 6.1 shows the running application.

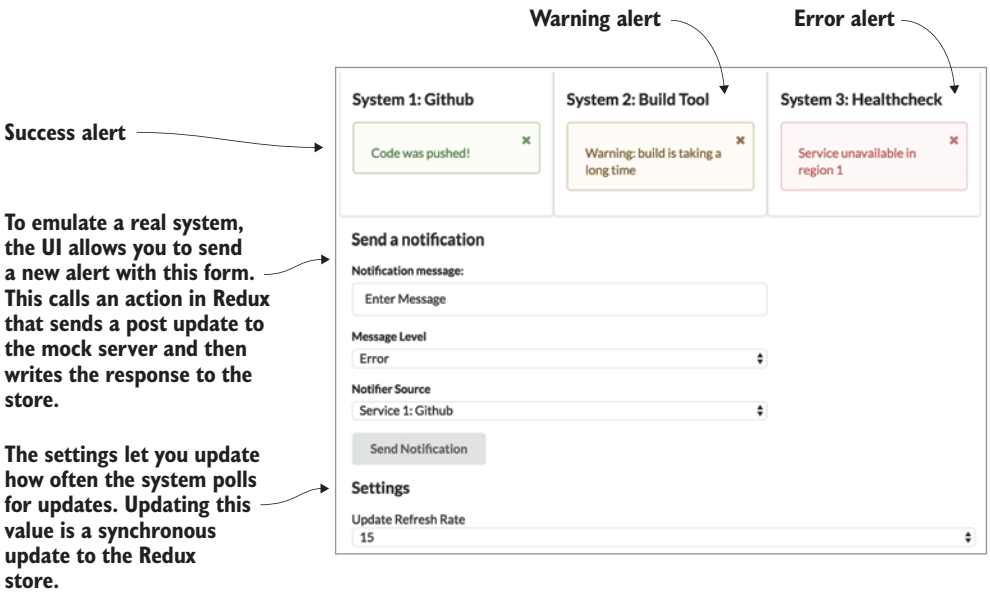

Figure 6.1 Notifications update app—send and receive notifications

The code has some React components and webpack already set up. I'm not going to spend much time on these topics so you can stay focused on learning Redux. If you want to review React, you can review chapters 3 and 4. For webpack, review chapter 5.

Also note that there's an in-memory object on the Node server that backs up the simple CRUD (create, read, update, delete) service for this project. If you were to build this in the real world, you'd want to explore using a WebSocket connection and connect a database. The "Send a notification" section of the interface allows you to emulate the app receiving alerts from services without having to hook it up to any real inputs.

6.1.2 Redux overview

In the first part of this chapter, we'll walk through all the pieces of Redux that are required to get updates flowing in your application. Figure 6.2 reviews Redux's single-direction update flow in the context of the notifications app and introduces you to the three main parts of Redux:

- *Actions*—Implement business logic, things like updating settings or adding new notifications to the list
- *Reducers*—Write state changes triggered by actions to the store
- *Store*—Current application state, holds the notification array and the values of any settings for the app

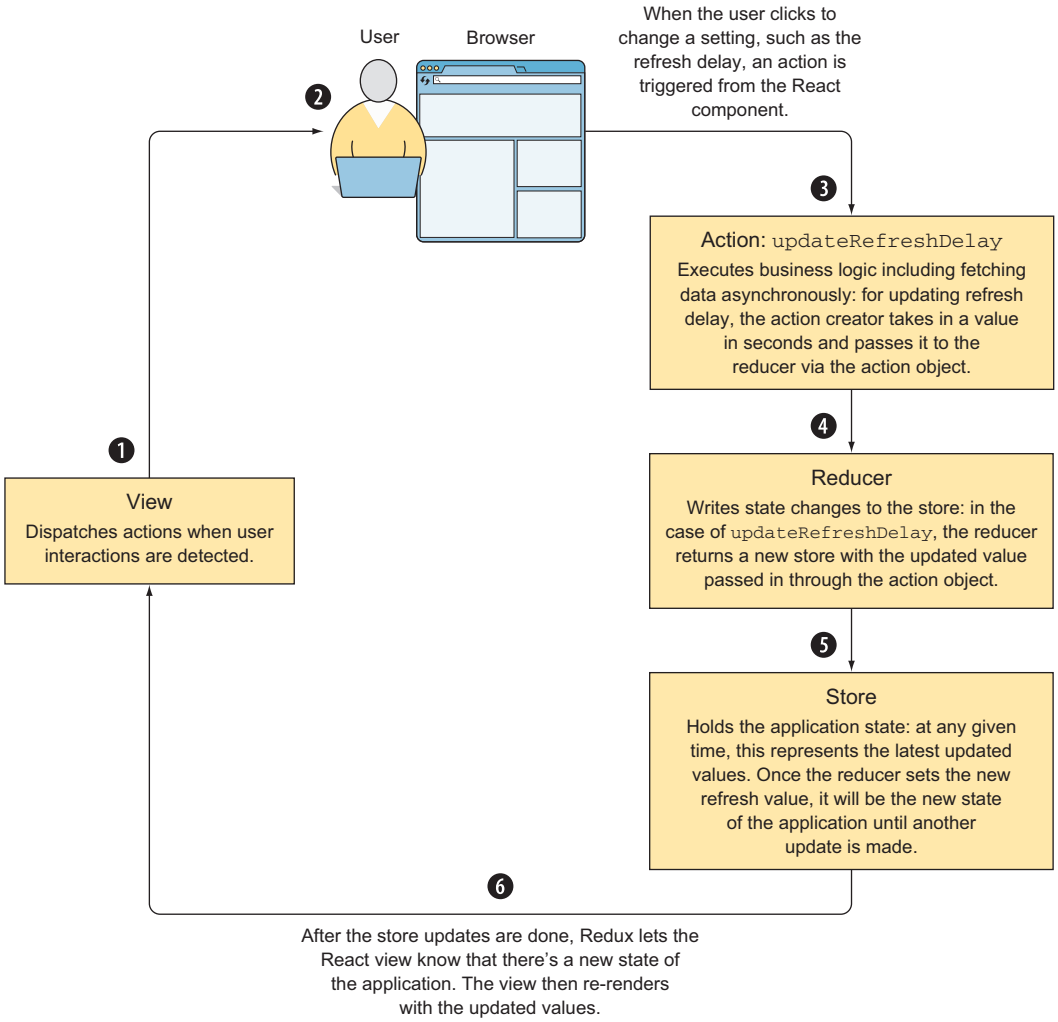

Figure 6.2 Redux single-direction flow from view

CONNECTING REACT AND REDUX

In the second part of this chapter, you'll learn how to use the React Redux library to connect your React view to your Redux application state. This includes using a top-level component provided by the library called Provider that takes in the store and makes it available to another component called connect. The connect component is a higher-order component that wraps some components in your application. These wrapped components are then able to receive store updates in the form of properties. The connect component has React state, so your other components don't need to have React state! Figure 6.3 illustrates how these pieces fit into your application structure.

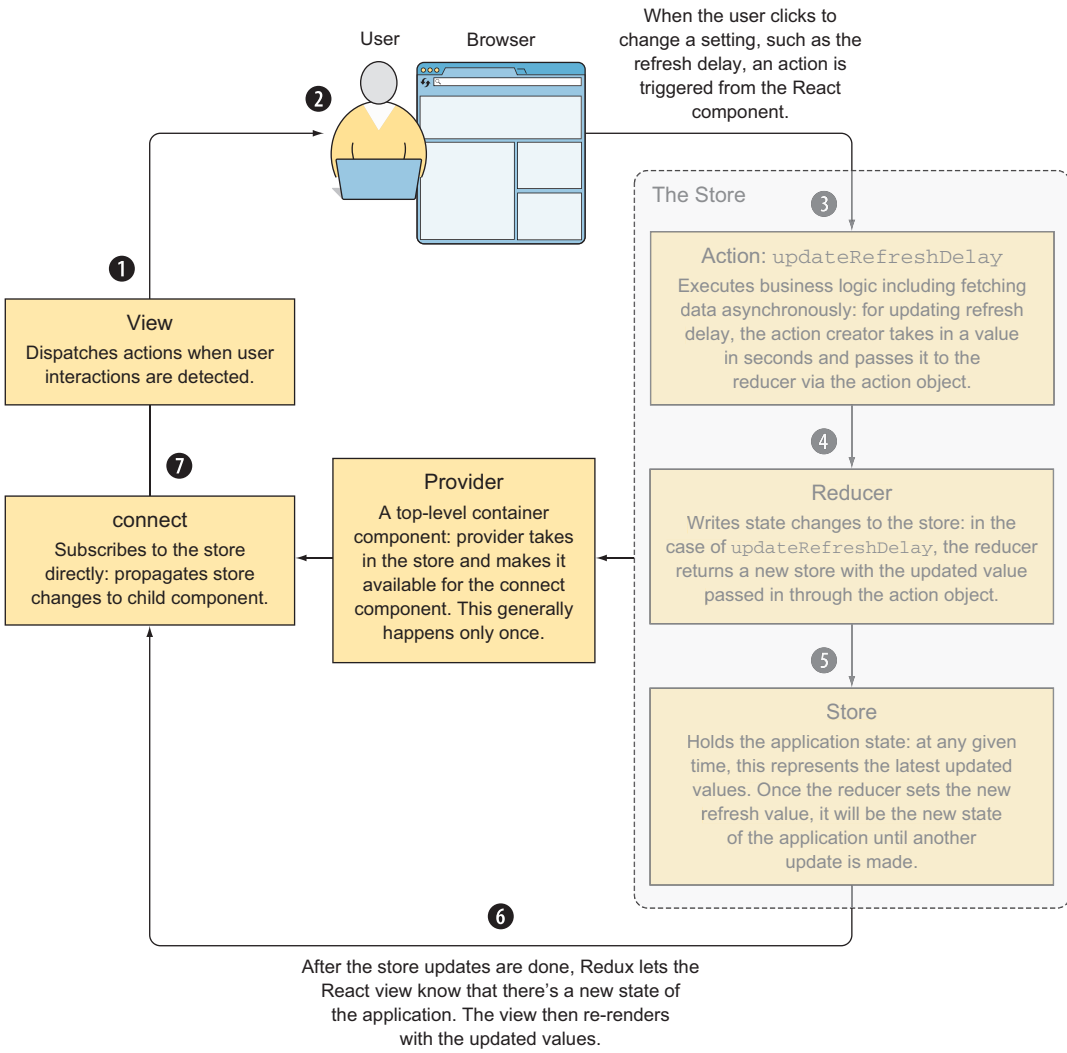

Figure 6.3 Using React Redux's Provider and connect components to hook up the React view with the application state

6.2 Redux as an architecture pattern

Often, when building web applications, you use a Model-View-Controller (MVC) pattern. Many common frameworks use this pattern. In this case, there's a view, the HTML of the application, a model that's some sort of representation of application state, and a controller that's the interface that the user interacts with. The business logic is also handled by the controller.

Frameworks such as Angular 1 and Ember each have their own implementations of MVC but historically have used two-way binding to handle the View-Controller part of

the framework. The flow of Angular 1 differs from the traditional MVC in that the view is really a View-Controller (always the same as a container component, as we discussed in chapter 3). But the framework still tries to follow an MVC pattern. This leads to confusing flows and hard-to-debug code.

Let's walk through what this would look like if we applied it to the app you're going to build in this chapter. Figure 6.4 shows how the application flow works in this case.

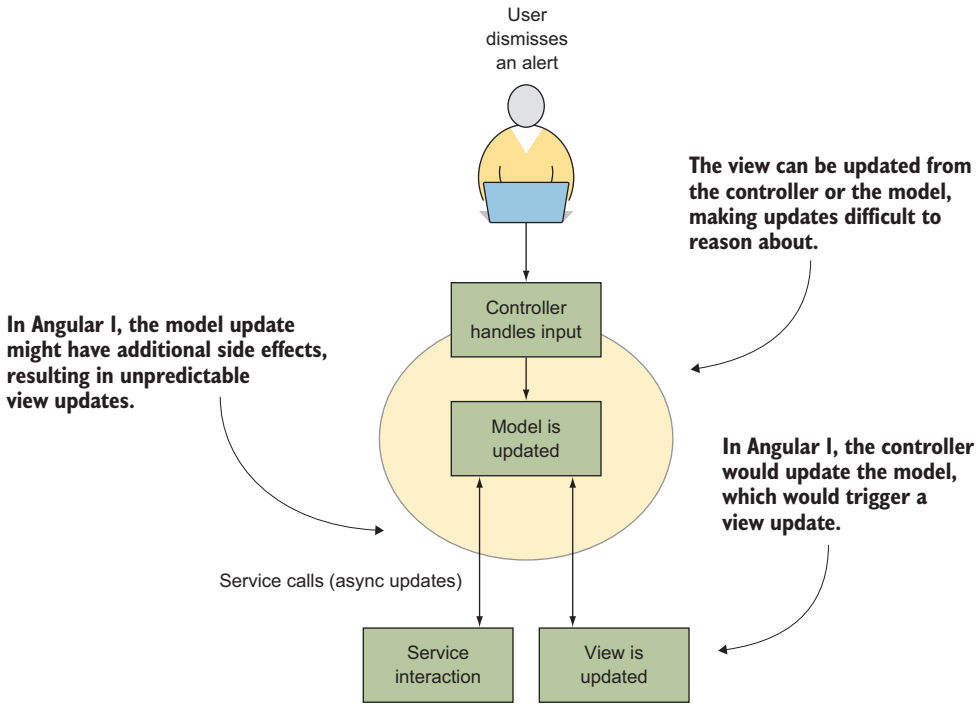

Figure 6.4 Model-View-Controller (MVC) flow in Angular 1

Redux's implementation has some overlap with MVC. I like to think of it as an evolution of MVC that works better for UI-based apps (as opposed to services/CRUD apps). There are a few major differences:

- Redux insists on a single-directional data flow resulting in easier-to-follow code and no side effects.
- There are no controllers. Rather, the views are also the controllers—called *view-controllers*. In this case, the View-Controller is React. This fits into the browser model well, where the view is rendered by the HTML and where user events are handled by the DOM.

- In Redux, there's always only one single root store, which represents the application state. That simplifies much of the logic, because views need to subscribe only to the root store and then pay attention to the specific subtrees they're interested in.

Redux flow relies on the store to dispatch actions. The `dispatch` function is a hook into the root store that allows you to trigger actions on the store. Sometimes you'll be triggering synchronous updates to the store and sometimes you'll be triggering an asynchronous call that will eventually update the store. Additionally, views are able to subscribe to the store and be notified when an update is complete. Figure 6.5 illustrates this flow.

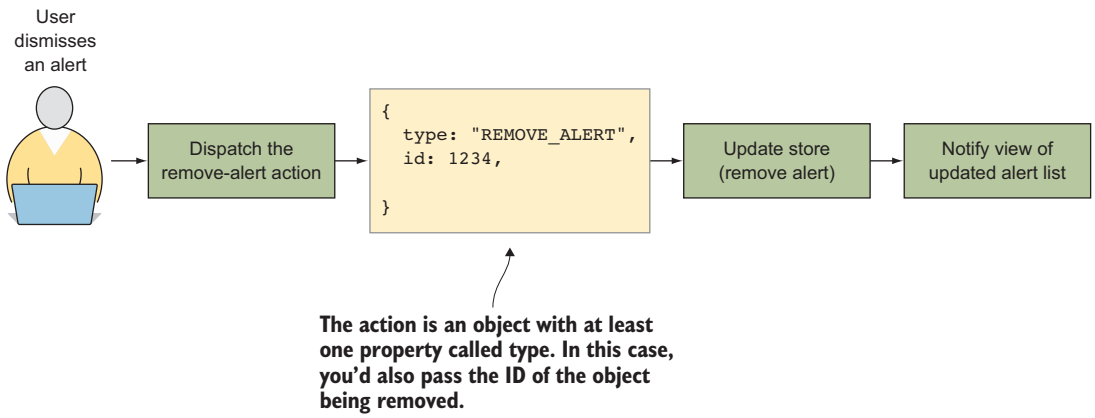

Figure 6.5 Redux flow when initiated by a user action

Redux implementation (the part of the code you'll write) is made up of the store, the actions, and the reducers. The store holds your application state. The actions take care of your business logic. The reducers are called to update the store.

> **DEFINITION** The *store* in Redux is the model of your application. It holds the current state of your application. I'll use *store* and *state* interchangeably to talk about the model in Redux.

To recap, Redux provides a concrete pattern for managing your application's state that's easy to use as a developer. It also makes reasoning about and debugging your application straightforward.

6.3 Managing application state

The primary job of Redux is to allow your state (or model) and the view to communicate. This is achieved by allowing the view to subscribe to state updates and trigger updates on the state. Figure 6.6 shows this flow in the context of the sample app.

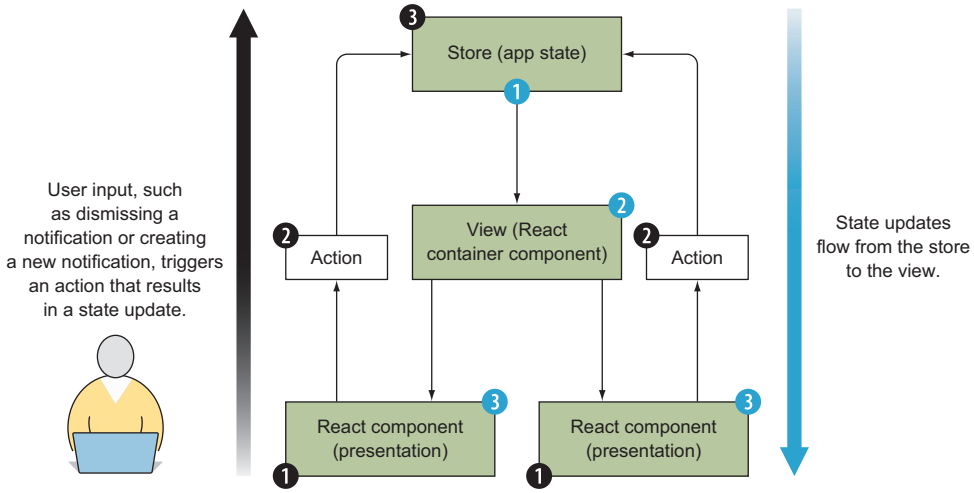

Figure 6.6 The flow of information between the view and Redux

Redux state can be a plain JavaScript object. The store (which contains the state object) has several methods that can be called on it. Here are the ones I'll cover:

- `dispatch(action)`—Triggers an update on the store (step 1 in figure 6.6).
- `getState()`—Returns the current store object (listing 6.1 shows what this looks like)
- `subscribe()`—Listens to changes on the store (step 2 in figure 6.6)

After actions are dispatched to the store, the state will match the code in the following listing.

Listing 6.1 An example store object (application state)

```
{
  notifications: {
    all: [
      {
        serviceId: 1,
        messageType: "success",
        message: "Code was pushed!"
      },
      {
        serviceId: 3,
        messageType: "error",
        message: "Service unavailable in region 1"
      },
      {
        serviceId: 2,
        messageType: "warning",
```

Inside root store, you can set up substores—this app has stores for notifications and settings.

The all array holds active notifications for your app.

```
        message: "Warning: build is taking a long time"
      }
    ]
  }
  settings: {
    refresh: 30
  }
}
```

The refresh property lets the user set the rate of long polling for updates.

Inside root store, you can set up substores—this app has stores for notifications and settings.

Redux provides a way to initialize the state (store). It manages the flow of updates to the store and notifies subscribers (the view). To configure the store in your app, you need to create your reducers and then initialize the store with them. The following listing shows how this works; you can find this code in src/init-redux.es6 in the repo.

Listing 6.2 Initialize Redux—src/init-redux.es6

```
import { createStore, combineReducers } from 'redux';
import notifications from './notifications-reducer';
import settings from './settings-reducer';

export default function (){
  const reducer = combineReducers({
    notifications,
    settings
     });
  return createStore(reducer)
}
```

Import helper methods from Redux.

Import app reducers.

Call combineReducers helper method from Redux; builds map of reducers from multiple reducers.

Export function that can be called from other modules (makes it reusable so it can be called from browser and server in isomorphic app).

Call createStore, pass in combined reducers—here you'll have store.notifications and store.settings.

If you aren't using Redux with React (later in the chapter you'll learn how to use redux-react to wire the two libraries together), you need to subscribe to store updates manually. The `subscribe` function works like a standard JavaScript event handler. You pass in a function that gets called every time a store update occurs. But the store doesn't pass its state to the update handler function; instead, you call `getState()` to access the current state. The following listing shows an example of this code, which you can find in main.jsx.

Listing 6.3 Subscribe to store, without React Redux—src/main.jsx

```
const store = initRedux();

store.subscribe(() => {
  console.log("Store updated", store.getState());
  // do something here
});
```

Initialize store (see listing 6.2).

Call the subscribe() method on the store and pass in a function to handle updates.

Log the current state of the store by calling getState().

Next you'll write a reducer and learn about maintaining immutability in Redux.

6.3.1 Reducers: updating the state

Reducers have a special name, but when broken down, they're pure functions. Each reducer takes in the store and an action and returns a new, modified store. Figure 6.7 shows the functional nature of a reducer function.

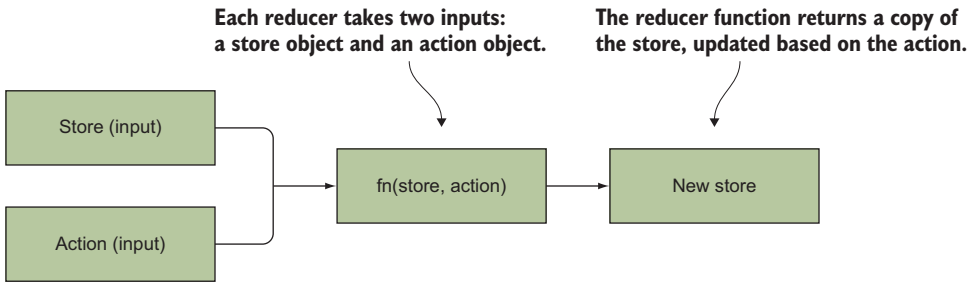

Figure 6.7 The input and output flow of a pure reducer function

The reducers in the notifications application are the wiring between the actions and the store. They're the only part of your code that should ever write updates to the store. Any other code that writes to the store is an antipattern. The following listing shows the reducer function for settings.

Listing 6.4 Settings reducers—src/settings-state

```
import {
  UPDATE_REFRESH_DELAY
} from './settings-action-creators';

export default function settings(state = {}, action) {
  switch (action.type) {
    case UPDATE_REFRESH_DELAY:
      return {
        ...state,
        refresh: action.time
      })
    default:
      return state
  }
}
```

Include the string constant for the action.

Function definition—each reducer takes two parameters, the store state and action. If the state doesn't exist, default it to empty object.

Use switch statement to declare your reducer logic—always determine which case to run based on value of action.type.

When the refresh value is updated, use the spread operator to copy and create new store to maintain immutability.

If no case matches, still return the store because this is a pure function.

There are two important points to understand about reducers:

- *Reducers must always be pure functions*—They take in values, use those values to create a new store, and then return a store.
- *Reducers must enforce the immutable nature of the store*—The store received by the function must be cloned if it needs to be updated.

Both concepts prevent unintended side effects. The next sections explain pure functions and immutability.

Pure functions

One of the most important parts of writing reducers is making sure the function stays pure (no side effects). *Pure* functions take in arguments that are used to calculate the return value—they don't use any state or do work on state. Code without side effects has many benefits, including being more testable and easier to understand and preventing hard-to-debug issues. Let's take a look at an example of a function with side effects and then compare it to a pure function. The following listing shows the difference between a pure and not pure function.

Listing 6.5 Pure function example

```
// side effect
let result;
function add(a, b) {
  result = a + b;
}

add(1, 2);
console.log(result); // logs 3

// functional - no side effects
function add(a, b) {
  return a + b;
}

console.log(add(1,2); // logs 3
```

Function doesn't return anything, but updates the value of result.

When add is called in this case, you can log the result to see what happened (global state).

In this function, result of add is returned.

This time log result of calling add function—there's no state.

Enforcing the immutable store

Another way to keep your code easy to understand and debug is to make sure the app state (or the store) is always immutable. The risk of not enforcing immutability is that you end up with issues that are difficult to track down and caused by changes in other parts of your code. By creating a new object each time, you ensure that other code won't accidentally change the whole app state.

You need to pay attention to a few things in order to enforce immutability in your store. Let's start with how to make sure your objects stay immutable, as shown in the following listing.

Listing 6.6 Mutating vs. immutable object

```
// mutation: bad
function addNotification(item, key, state) {
  return state[key] = item;
}

//immutable: good
function addNotification(item, key, state) {
```

In the bad example, item is inserted directly into the state object, then the state is returned.

Function declaration that takes three params: item, key, and state

```
  return {
    ...state,
    key: item
  }
}
```

In the good example, the object is cloned using spread operator, which takes state that was passed in and creates the object with its keys. Then the new copied object is returned.

Here, you can see that the immutable way of returning the store object involves the JavaScript function spread operator. You create a new object by spreading the old object and then adding any new or updated keys. The new keys will overwrite the old. But if you have a deeply nested object, you need to build the full object here or use a helper library to manage deeply nested keys.

Similarly, arrays need to be kept immutable. With arrays, pushing directly into the array is a mutation, so it's necessary to create a new array instead. The following listing demonstrates the wrong and the right way to do this.

Listing 6.7 Immutable arrays

```
// bad: mutating the original array
function addItem(item) {
  return itemsArray.push(item)
}

// good: creating a new array
function addItem(item) {
  return [...itemsArray, item]
}
```

Pushes item into array, returns original array—this is a mutation.

Shows immutable way: return brand-new array with items from original array and new item; uses spread operator to push items into an array.

6.3.2 Actions: triggering state updates

Actions are the only way to trigger an update to your application state in a Redux application. This is important to ensure that your app enforces the single-direction flow. (It's technically possible to update the store directly, but you should *never* do that). Only reducers triggered by actions should update the state.

Because actions are synchronous by default, any update that needs to be made can happen quickly. In fact, the dispatcher itself is completely synchronous. By default, Redux supports only synchronous actions. (Later in this chapter, you'll learn to use middleware with Redux in order to allow asynchronous actions.)

> **TIP** You can't dispatch an action from a reducer. That breaks the single-directional flow of Redux and could lead to unwanted side effects. Don't worry, Redux won't let you do it, but it's important to avoid thinking about updates in that way.

The simplest action is an object with one property called `type`:

```
{ type: 'UPDATE' }
```

Actions will often be objects that contain data to be updated in the store in addition to the `type` property. Because most actions in your application will be reused by more

than one view, it's recommended to create reusable functions called *action creators* that return the action you want to dispatch.

Action creator files are also a good place to define your string constants for actions. This reduces errors by ensuring that the action creator dispatches the same action type value the reducer is looking for. This can also lead to gains in developer speed in some IDEs if you have static type checking or similar features enabled.

You can see these two concepts in the next listing. This code can be found in the repo as well. The listing shows an action for updating the time interval for the long polling functionality of the app.

Listing 6.8 Synchronous actions—src/settings-action-creators.es6

```
export const UPDATE_REFRESH_DELAY
➥ = 'UPDATE_REFRESH_DELAY';

export function updateRefreshDelay(time) {
  return {
    type: UPDATE_REFRESH_DELAY,
    time: time
  }
}
```

Setting type value to a constant reduces errors

Action creator function declaration takes one parameter called time.

Returned action has two properties—type property is required and its value is always a string.

The time property is added to the action so that the value can be used by the view when it updates—each action will have different data properties.

You can use the `const` in the first line from the reducer to ensure that the action creator and the reducer point at the same value. To dispatch this update to the store, all you have to do is call `dispatch` on `store` and pass in the action. Because you're using an action creator, you call the action creator and pass the result into `dispatch`:

```
store.dispatch(updateRefreshDelay(5));
```

The reducer will then be triggered, and the store will be updated.

Next, you'll learn how to set up Redux with middleware so you can include additional functionality such as making asynchronous calls.

6.4 Applying middleware to Redux

Redux includes a helper method that lets you extend the default functionality of the dispatcher. For every middleware you apply to the dispatcher, it adds a function to the chain of calls that will happen before the final default dispatch behavior. Here's a simplified example of what that looks like:

```
middleware1(dispatchedAction).middleware2(dispatchedAction).middleware3(dispa
    tchedAction).dispatch(dispatchedAction)
```

This allows you to add functionality for debugging and making asynchronous calls. First, let's look at how you add debugging.

6.4.1 Middleware basics: debugging

It's possible to add improved debugging with middleware. One example of this is the Redux Logger library. This library helps you see the state changes clearly in the console. Figure 6.8 shows sample action logs.

Figure 6.8 Redux Logger console output

You add middleware when you instantiate your store. The following listing shows how to do that. The code can also be found in the repo.

Listing 6.9 Setting up middleware—src/init-redux.es6

```
export default function () {
  const reducer = combineReducers({…});
  let middleware = [logger];
  return compose(
    applyMiddleware(...middleware)
  )(createStore)(reducer);
}
```

Create middleware array so you can pass an arbitrary number of middleware and easily control the order.

Call compose and pass in the store so the middleware will be applied to store.

Call applyMiddleware on middleware array to set up middleware properly.

When you run the app, you'll see the logging in the console; this is helpful for debugging.

6.4.2 Handling asynchronous actions

Earlier in the chapter, you dispatched actions by writing functions that return an action object. As stated previously, we call those functions action creators. *Asynchronous action creators* apply the same principles, but instead of immediately returning the object, they wait for something to happen (for example, a network call to complete) and then return the action object.

To do that, you need access to the `dispatch` object inside your action creator function. This requires another middleware library, called Redux Thunk. To use the middleware, you need to add it to the middleware array in init-redux.es6 (refer to listing 6.9). It's already in the code in the repo.

Then to take advantage of this middleware, you write an action creator that looks like this:

```
export const UPDATE_ACTION = 'UPDATE_ACTION';

export function actionCreator() {
```

```
  return dispatch => {
   return dispatch({
       type: UPDATE_ACTION
     })
  }
}
```

By adding the Thunk middleware, you can now access the `dispatch` function on the store inside your action creator (all the middleware does is provide the `dispatch` parameter to your returned function). Note that you also need to export your action creator and the corresponding `const` for the action. This is identical to earlier in the chapter, when you created a synchronous action creator.

In the notifications app, you need three asynchronous actions: adding a notification, fetching the notifications, and deleting a notification. The following listing shows the Fetch Notifications action creator. The code can be found in the repo along with other action creators.

Listing 6.10 Asynchronous action creators—src/action-creators.es6

```
import request from 'isomorphic-fetch';

export const FETCH_NOTIFICATIONS
➥ = 'FETCH_NOTIFICATIONS';

export function fetchNotifications() {
  return dispatch => {
    let headers = new Headers({
      "Content-Type": "application/json",
    });
    return fetch(
        'http://localhost:3000/notifications',
        { headers: headers }
      )
    .then((response)=>{
      return response.json().then(data => {
        return dispatch({
          type: FETCH_NOTIFICATIONS,
          notifications: data
        })
      })
    })
  }
}
```

Use isomorphic fetch so both server and browser can handle the fetch call.

Const for action type

The action creator returns a function instead of an object. Thunk middleware calls this function and injects the dispatch method from store.

Create headers to talk to the API.

Call fetch with URL and options.

Promise handler

Get JSON out of the response—because this is also a promise, add second promise handler.

After you have data, dispatch the action.

Now that you've seen how the Redux reducers and actions work, let's go over how to hook up React and Redux.

6.5 Using Redux with React components

In a React app, the actions are typically dispatched from components. To have access to the store in a component, you need to wire up your React components to Redux. I recommend using the react-redux library, which is provided by the author of Redux as

the official bindings for React. It implements all the code necessary to subscribe to and receive updates from the Redux store.

There are two distinct parts to this. One is a top-level root component called Provider. The other is a higher-order component (HOC) called connect.

6.5.1 Wrapping your app with provider

First, you need to pass the store into your app. You want to pass it down as a React prop. Remember, React components have a property called `props`. The `props` object is created by passing down values from the parent React component to its children. This object is immutable and can be changed only from the parent component.

Because you also want to be able to subscribe to the store, you should use the Provider component that comes with React Redux. This React component acts as the root of your application and makes the store available to the connect HOC. The following listing shows how to do this.

Listing 6.11 Connecting Redux to React —src/main.jsx

```
import React from 'react';
import ReactDOM from 'react-dom';
import App from './components/app.jsx';
import { Provider } from 'react-redux';
import initRedux from './init-redux.es6';
require('./style.css');

const initialState = window.__INITIAL_STATE;
const store = initRedux(initialState);

store.subscribe(() => {
  console.log("Store updated", store.getState());
  // if not using React, do something here
});

ReactDOM.render(
  <Provider store={store}><App /></Provider>,
  document.getElementById('react-content')
);
```

The component takes in the store and properly passes it to its children.

Render the App component inside Provider so it'll have access to store and pass in store to Provider component.

Now you have access to the store in your components. But you need to do a couple more things to completely connect your app to Redux.

6.5.2 Subscribing to the store from React

The second part of getting store updates is wrapping your container components in the connect HOC. This component handles subscribing to the store for you. It holds all the React state that's necessary to pass down properties to its child component.

The connect HOC also provides helper methods that make it easier to map the store to properties and easier to call actions from the view. Wrapping a component with connect and then exporting it for use in your app looks like this:

```
export default connect(mapStateToProps, mapDispatchToProps)(Component);
```

The functions `mapStateToProps` and `mapDispatchToProps` are the two helper callbacks that connect runs. The first one, `mapStateToProps`, is run every time an update occurs to the store. Inside of it, you'll define what items from the store should be mapped to React `props`. The following listing shows this in action.

Listing 6.12 Connect React to Redux—src/components/app.jsx

```
class App extends React.Component {

  componentDidMount() {}

  getSystemNotifications(id) {
    let items = [];
    if (this.props.all) {
      this.props.all.forEach((item, index)=>{
        if (item.serviceId == id) {
          let classes = classnames("ui", "message", item.messageType);
          items.push(
            <div className={classes} key={index}>
              <i
                className="close icon"
                onClick={
                  this.dismiss.bind(this, index)
                }>
              </i>
              <p>
                {item.message}
              </p>
            </div>
          )
        }
      })
    }
    return items;
  }

  render() {}
}

function mapStateToProps(state) {
  let { all } = state.notifications;
  let { refresh } = state.settings;
  return {
    all,
    refresh
  }
}

function mapDispatchToProps(dispatch) {}

export default connect(
                mapStateToProps,
                mapDispatchToProps
              )(App)
```

Component accesses notifications directly on props.

Using notifications array, you build an array of notification items.

The function tells connect to pull specific keys out of the store and put them directly on props.

Pull out relevant items (notifications and refresh); refresh is required by the child component.

Return just the keys the component needs instead of the whole store.

Pass mapStateToProps into the connect function; it will be called during render cycle.

With `mapDispatchToProps`, you're making actions available to be dispatched directly from the component's properties. Normally, you'd need to fully write out `dispatch(actionCreator())` every time you wanted to initiate an action. This helper method lets you use JavaScript's `bind` to automatically dispatch actions when they're called from the view. The following listing shows how this works. Note that React Redux provides another helper method to automate the bind code.

Listing 6.13 Connect React to Redux—src/components/app.jsx

```
import React from 'react';
import { connect } from 'react-redux';
import { bindActionCreators } from 'redux';
import * as actionCreators from '../action-creators';
import * as settingsActionCreators
➥ from '../settings-action-creators';

import CreateNotification from './create-notification';
import Settings from './settings';
import classnames from 'classnames';

let intervalId;

class App extends React.Component {
  //...component implementation code

  componentDidMount() {
    intervalId = setInterval(() => {
      this.props.notificationActions.
      ➥ fetchNotifications();
    }, this.props.refresh * 1000);
  }
}

function mapDispatchToProps(dispatch) {
  return {
    notificationActions:
    ➥ bindActionCreators(actionCreators, dispatch),
    settingsActions:
    ➥ bindActionCreators(settingsActionCreators, dispatch)
  }
}

export default connect(null, mapDispatchToProps)(App)
```

Connect is the higher-order function provided by React Redux. It subscribes to the store and passes the updated store down as props into the connected component.

Import action creators so you can call actions in your component.

bindActionCreators is a helper method that takes in an action or an object with actions and creates a function that, when called, dispatches the requested action.

Call the fetchNotifications action on a regular interval; actions are passed down as props by connect.

Function passed into connect so connect component can pass down bound actions as properties—prevents having to call dispatch every time you want to call an action.

Call connect, passing in mapDispatchToProps and then passing in the component you want to connect to Redux

After you've wired up your container component (App) to connect it to Redux, all you have to do is pass the properties into the children. Then the child components can see any state you mapped to `props` and call any actions you've bound to `dispatch`.

Summary

In this chapter, you learned how Redux works, including how to implement unidirectional data flow, maintain an immutable store, and connect React with Redux.

- Redux implements an architecture pattern that's an evolution of the traditional MVC pattern.
- The single-directional flow of Redux, where the view dispatches actions and subscribes to store updates, makes reasoning about the system simpler for developers.
- The store, or state, of your application is a single root object that holds all the information for your view.
- Reducers are pure functions that make changes to the store. They never mutate the store and instead use immutable patterns to make updates to the store.
- Actions are used to trigger updates to the store.
- Middleware allows debugging tools and asynchronous actions to be used in Redux.
- Connecting React and Redux requires additional functionality provided by the React Redux library, which includes a higher-order component that subscribes to the store for its child component.

JAVASCRIPT/WEB DEVELOPMENT

Isomorphic Web Applications

Elyse Kolker Gordon

Build secure web apps that perform beautifully with high, low, or no bandwidth. Isomorphic web apps employ a pattern that exploits the full stack, storing data locally and minimizing server hits. They render flawlessly, maximize SEO, and offer opportunities to share code and libraries between client and server.

Isomorphic Web Applications teaches you to build production-quality web apps using isomorphic architecture. You'll learn to create and render views for both server and browser, optimize local storage, streamline server interactions, and handle data serialization. Designed for working developers, this book offers examples in relevant frameworks like React, Redux, Angular, Ember, and webpack. You'll also explore unique debugging and testing techniques and master specific SEO skills.

What's Inside

- Controlling browser and server user sessions
- Combining server-rendered and SPA architectures
- Building best-practice React applications
- Debugging and testing

To benefit from this book, readers need to know JavaScript, HTML5, and a framework of their choice, including React and Angular.

Elyse Kolker Gordon runs the growth engineering team at Strava. Previously, she was director of web engineering at Vevo, where she regularly solved challenges with isomorphic apps.

"A practical guide to performant and modern JavaScript applications."
—Bojan Djurkovic, Cvent

"Clear and powerful. If you need just one resource, this is it."
—Peter Perlepes, Growth

"Thorough and methodical coverage for novice users, with handy insights and many 'aha' moments for advanced users. Highly recommended."
—Devang Paliwal, Synapse

"An essential guide for anyone developing modern JavaScript applications."
—Mike Jensen, UrbanStems

To download their free eBook in PDF, ePub, and Kindle formats, owners of this book should visit
manning.com/books/isomorphic-web-applications

ISBN-13: 978-1-61729-439-6
ISBN-10: 1-61729-439-X
53999
9 781617 294396

MANNING $39.99 / Can $56.99 [INCLUDING eBOOK]