CHAPTER 6

Complex State Management with Redux

As you've seen throughout this book, it is possible to structure complete applications using React (despite the fact that is primarily an UI library - not a full stack framework). This is possible due to two React principles: The unidirectional data flow and the separation on components into container and pure:

1. **Unidirectional data flow.**
   In a React application, data flows from parent components to child components via props. This one-way data flow allows for clear, explicit and easy to follow code.

2. **Applications can be structured around Container and Pure components.**
   This is not a React characteristic, but rather a stablished best practice.
   Pure components (frequently called "Presentational components") are concerned with how things look, rendering DOM markup. They usually don't have their own state and instead receive data and callbacks through props from container components.
   Container components are responsible for providing the data and behavior to other components. They usually don't have any DOM markup of their own (except for some wrapping divs) and instead render presentational components.

All these make for an architectural pattern that holds really well for small to medium-sized applications. But as your application grows and state mutations get more complex, new challenges arise. A typical example is the tedious and error-prone task of passing down data and callbacks to nested components many levels deep.

Ryan Florence, React Router co-author and prominent community member, uses an analogy to describe this: drilling your application. If you have many nested components, you have a lot of drill work going on, and if you want to refactor (move some components around), you must do a whole lot of drilling all over again.

In scenarios like this, it pays off to use a dedicated state management library such as Redux. It let you bring data and, most importantly, the callbacks to manipulate that data closer to each of these components when the applications grow.

To begin creating React+Redux applications, there are three pieces that you need to understand: The store, actions and reducer functions:

Store

As mentioned earlier, one of the main points you are trying to address is how to bring data closer to each of the application's container components. Our ideal view of the world looks like Figure 6-1. Data is completely separated from the component, but you want the component to be notified when data changes so it can re-render.

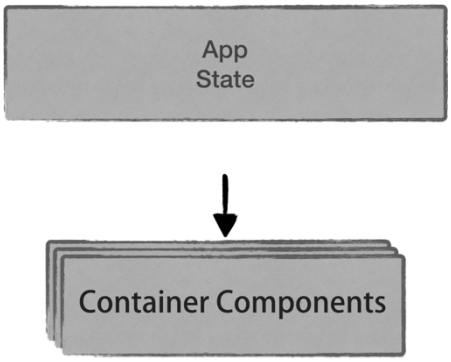

Figure 6-1.

That's exactly what is achievable by using Redux. Redux provides a store to hold the entire application state in a single, centralized JavaScript object. The store can be directly accessed by any container components in the application:

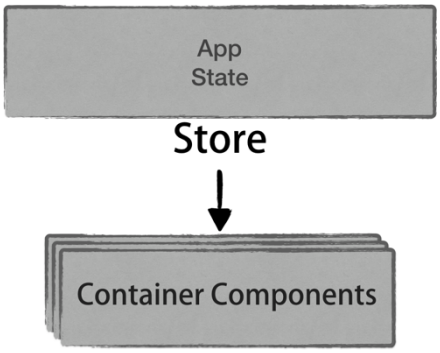

Figure 6-2.

The store exposes three methods you can call from your container components:

- `getState`: Used by container components on any hierarchy level to read the current state from the store.
- `subscribe:` React Container components can subscribe to the store to be notified of state changes and re-render themselves - as well as their children.
- `dispatch`: Used to dispatch an action to the store. Dispatched actions are used to trigger state changes.

In other words, in Redux the store is a state container - the place where all your application state lives. Container components can get the state and subscribe to the store to be notified of state changes. As you will see in the next topic, container components at any nesting level can also "dispatch" state changes directly to the store, without having to rely on callbacks passed as props.

But before moving on to action dispatching and the other parts that make Redux, it is important to stress that a React+Redux applications always contain only one store, and all your application state lives in a single object in this single store. Conceptually it's just like a local database for your app – Though you will store any kind of state inside it (even state that you would normally put inside de view layer).

Having all application state centralized in a single place may sound radical, but it brings lots of benefits: It eliminates the need to sync states across many different stateful components (The state of this widget needs to be synced with the application state, which needs to be synced with some other widget, etc.); it makes state changes predictable; makes it easier to introspect and debug an application and the list goes on.

Actions

As you've seen in the previous topic, the Redux store provides methods for getting the state and subscribing to state changes – but there is no setter method to directly manipulate state inside the Store. As far as the rest of the application is concerned, the store is read-only – no part of the application can change the state inside it (or, to be precise, the only part of the application that can update the store is the store itself.).

Obviously there are many circumstances where you might want to change the state in the store: Whether as a result of user interactions (such as: clicking on a button, leaving a comment, requesting search results and so on...) or as result of AJAX requests, timers, web socket events etc. Only the store can update its state, but it provides a mechanism by which any other part of the application can indicate that the state needs to change: dispatching actions.

Dispatching an action is like sending a message to the store saying that something happened and that the store should update itself in response.

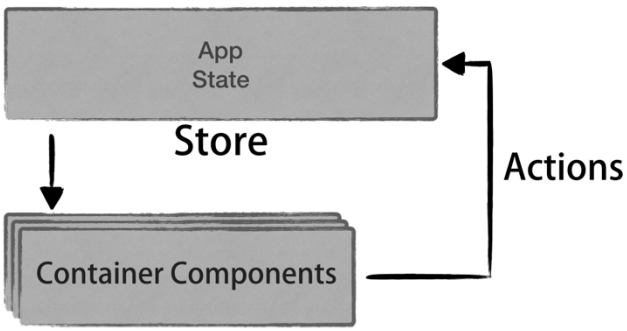

Figure 6-3.

That's the basic flow in a React+Redux application: React container components dispatches actions to the store to indicate that something that impacts application state happened (say, when the user clicks on a button); The store updates its own state and notify the subscribed components; Container components re-render the UI with the latest data. It's a mechanism that expands on React's own unidirectional data flow principle with added flexibility.

Inside the Store: Reducer functions

The third piece in Redux are the reducer functions. They are used internally by the store and are responsible for mutating the store state when actions are dispatched.

But what exactly is a reducer function? To put it in the simplest possible terms, a reducer is a function that calculates a new state given the previous state and an action – in a similar way of how Array's reduce works.

When you instantiate your application's store, you pass one (or more) reducer functions. These functions will be called every time an action is dispatched and their jobs is to calculate the new, updated state. Using reducer functions in this way is one of the most ingenious aspects of Redux (in fact that's why the library itself is called "Redux").

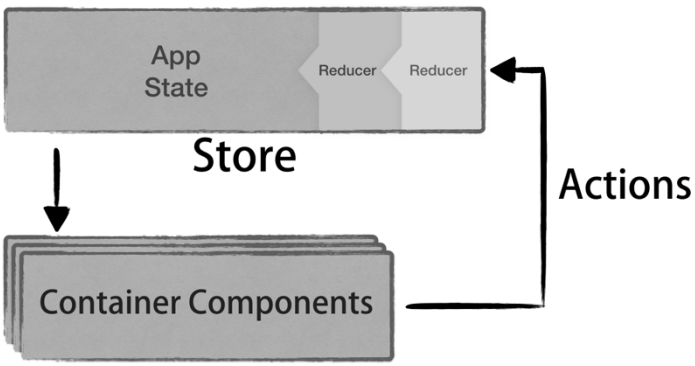

Figure 6-4.

The Unrealistic, Minimal Redux App

When used in complex applications, Redux helps keep the code easier to read, maintain, and grow. It certainly reduces complexity, and as a consequence in many cases it also reduces the number of lines of code in the project (although that's not an appropriate metric for code complexity). But no such thing will happen in your first sample project, where the use of Redux will actually increase the amount of code necessary to build it. This is because the purpose of this first example is to help you grasp all of the elements in a React + Redux application. Redux can be tricky to newcomers, so you will start on a very basic project, with almost no UI (or practical use, for that matter) but one that designed to make explicit and well-defined use of all of the elements in a React + Redux application. In the next section, you will move to complete, real-world examples.

The Bank Account Application

There is a great analogy that can be used to describe the Redux flow: a bank account. A bank account is defined by two things: a transaction and a balance. With every transaction, we update the balance, as shown in tables 6-1 and 6-2:

Table 6-1. The first transaction initiates the balance

Transaction	Amount	Balance
Create Account	$0	$0

Table 6-2. With every transaction, we update the balance.

Transaction	Amount	Balance
Create Account	$0	$0
Deposit	$200	$200
Withdrawal	($50)	$150
Deposit	$100	$250
		$250

These transactions are how we're interacting with our bank; they modify the state of our account.

In Redux terms, the transactions on the left are our actions, and the balance on the right is the state in the store. Our sample application structure will include:

- A constants.js file: Since all actions should have unique names ("types", to be precise), and those names will be referenced across the app, we will store them as constants. This is not mandatory nor it's a Redux requirement, but rather a good practice to use every time you need to provide uniquely identifiable names across the application.
- bankReducer.js: The app will have a single reducer function that will be used by the store.
- bankStore.js: Instantiate a Redux store (which will keep track of the user's balance) using the reducer function.
- The App.js file, which contains both the presentational component and a container component we will use in this project.

Start by creating a new project and installing the Redux library with npm:

```
npm install --save redux
```

The next topics will walk through each step in the project.

Application's constants

Let's get started by defining the constants file. We need three constants to uniquely identify our actions across the app for creating an account, depositing in the account and withdrawing from the account. Listing 6-1 shows the file:

Listing 6-1. The constants.js file

```
export default {
  CREATE_ACCOUNT: 'create account',
  WITHDRAW_FROM_ACCOUNT: 'withdraw from account',
  DEPOSIT_INTO_ACCOUNT: 'deposit into account'
};
```

Reducer

In sequence, let's define a reducer to be used in the Store. The reducer function will receive the current store state and the dispatched action, and it should return the new state. Notice that the state in the Redux store must necessarily be treated as immutable – so be careful use one of the following constructs to represent the state:

- Single primitive values (a String, Boolean value or a number)
- An Array or primitive values (in. E.g.: [1,2,3,4])
- An object of primitive values (in. E.g.: {name:'cassio', age:35})
- An object with nested objects that will be manipulated using React immutable helpers.
- An immutable structure (using something like immutable.js library, for example).

In this simple application, we will use an object with a single primitive numeric value: `{balance: 0}`

The complete code is shown in listing 6-2:

Listing 6-2. The bankReducer.js file

```
import constants from './constants';

const initialState = {
  balance: 0
}

const bankReducer = (state, action) => {

    console.log(action); //Temporarily logging all actions

    switch (action.type) {
      case constants.CREATE_ACCOUNT:
        return initialState;

      case constants.DEPOSIT_INTO_ACCOUNT:
        return {balance: state.balance + parseFloat(action.amount)};

      case constants.WITHDRAW_FROM_ACCOUNT:
        return {balance: state.balance - parseFloat(action.amount)};

      default:
        return state;
    }
}

export default bankReducer;
```

Notice that we are using the reducer to log all dispatched actions. It's not a good practice to use the log here, so it will remain temporarily here – this code will be refactored at later steps.

Most importantly, observe how a switch statement is used to make different transformations on the state based on the dispatched action:

- When the CREATE_ACCOUNT actions is dispatched, it returns the default initial state for a new bank account (balance zero).
- If the actions DEPOSIT_INTO_ACCOUNT or WITHDRAW_FROM_ACCOUNT are dispatched, the function adds or subtracts the `amount` value passed with the action and return a new, updated object.

- Finally, the reducer has a default case to prevent errors. If an unknown action is dispatched, it simply returns the old state again.

Store

In the sequence, let's define your Store file. In a Redux application, the store owns state and is instantiated with the reducers that manipulates that state. Creating a new store is pretty easy – all you have to do is invoke Redux's `createStore` method passing the reducer function, as shown in listing 6-3:

Listing 6-3. The bankStore.js source file:

```
import { createStore } from 'redux'
import bankReducer from './bankReducer';

const bankStore = createStore(bankReducer);
export default bankStore;
```

UI Components

Finally, all we need is some UI. In the App.js file, you will create two components: The first, "BankApp" will be a pure, presentational component. It will receive the bank balance and the callbacks to deposit and withdraw via props and simply mount the interface to show the balance, text field and buttons. The second one, BankAppContainer will have access to the application store and pass down props to the presentational component.

BankApp pure (presentational) component

Let's start with the BankApp component, as shown in listing 6-4.

Listing 6-4. Partial source code for App.js showing the BankApp presentational/pure component

```
import React, { Component, PropTypes } from 'react';
import { render } from 'react-dom';
import bankStore from './bankStore';
import constants from './constants';

class BankApp extends Component {
  handleDeposit() {
    this.props.onDeposit(this.refs.amount.value);
    this.refs.amount.value = '';
  }

  handleWithdraw() {
    this.props.onWithdraw(this.refs.amount.value);
    this.refs.amount.value = '';
  }

  render() {
    return (
      <div>
        <header>
          <img src="//www.pro-react.com/logos/redux-bank.svg" width="150" />Redux Bank
        </header>
        <h1>Your balance is ${(this.props.balance).toFixed(2)}</h1>
        <div className="atm">
          <input type="text" placeholder="Enter Ammount" ref="amount" />
          <button onClick={this.handleWithdraw.bind(this)}>Withdraw</button>
```

```
          <button onClick={this.handleDeposit.bind(this)}>Deposit</button>
        </div>
      </div>
    );
  }
}
BankApp.propTypes = {
  balance: PropTypes.number,
  onDeposit: PropTypes.func,
  onWithdraw: PropTypes.func
};
```

Notice in the code above that the BankApp component expects to receive three props from the parent container component: balance (the balance state on the store), onDeposit and onWithdraw (callbacks that it calls passing the amount typed by the user).

BankAppContainer component

The container component will have access to the store, and it has two main responsibilities:

- **Map state to props**: The container component will get the state values from the store (only the necessary ones - the balance in this case) and pass it down to the presentational component via props. In other words, it will map certain state keys to props.
- **Map dispatch to props**: In a similar fashion, the container component will pass down some callbacks that ultimately will end up dispatching actions. So you need to write the methods that will dispatch actions and map them to props to pass down.

It may sound confusing at first, so let's dig down into some code. Listing 6-5 shows the whole source code for the App.js file (including the BankAppContainer and the render call):

Listing 6-5. The complete App.js

```
import React, { Component, PropTypes } from 'react';
import { render } from 'react-dom';
import bankStore from './bankStore';
import constants from './constants';

class BankApp extends Component {...} // Ommited for brevity
BankApp.propTypes = {...} // Ommited for brevity

class BankAppContainer extends Component {
  constructor(...args) {
    super(...args);
    bankStore.dispatch({type:constants.CREATE_ACCOUNT})
    this.state = {
      balance: bankStore.getState().balance
    }
  }

  componentDidMount() {
    this.unsubscribe = bankStore.subscribe(() =>
      this.setState({balance: bankStore.getState().balance})
    );
  }

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

```
  render(){
    return(
      <BankApp
         balance={ bankStore.getState().balance }
         onDeposit={ (amount)=>bankStore.dispatch(
                       {type:constants.DEPOSIT_INTO_ACCOUNT, amount:amount} )}
         onWithdraw={ (amount)=>bankStore.dispatch(
                      {type:constants.WITHDRAW_FROM_ACCOUNT, amount:amount} )}
      />
    )
  }
}

render(<BankAppContainer />, document.getElementById('root'));
```

Let's review the code above in parts. In the class constructor you dispatch the "CREATE_ACCOUNT" action and define the local state containing a balance key. The value for this key comes from the `bankStore` (`bankStore.getState().balance`). In the sequence, you use the lifecycle methods `componentDidMount` and `componentWillUnmount` to manage listening for changes in the `bankStore`. Whenever the store changes, the component's state gets updated (and, as you already know, as the state changes, the component will re-render itself). Notice that the store's `subscribe` method returns a function that is used to unsubscribe from it – you are storing this function in **`this`**.unsubscribe.

It's also worth noticing in the render function of the code above that it's returning the **`BankApp`** component and mapping the state of balance to a prop "balance". It' also mapping two callbacks (onDeposit and onWithdraw) that will be used to dispatch actions to the store.

Everything is wired up. Let's throw some styling (as shown in listing 6-6) and test the application.

Listing 6-6. Basic styling for the fake banking application:

```
body {
  margin: 0;
  font-family: "Myriad Pro", Helvetica, Arial, sans-serif;
  background-color: #00703B;
  color: #fff;
  text-align: center;
}
header {
  width:100%;
  padding: 15px;
  background-color: #D8D8BF;
  color: #00703B;
  font-size: 30px;
}
h1{
  font-size: 30px;
}
.atm {
  width: 300px;
  height: 115px;
  border-radius: 10px;
  background-color: #1D4F27;
  margin: 10px auto 0 auto;
  padding: 20px;
}
.atm input {
  font-size:25px;
  width: 272px
}
.atm button {
  margin: 20px 10px;
```

```
  padding: 20px 40px;
}
```

If you follow along, try withdraw and deposit operations and make sure you have the browser console opened so you can see the all the actions logged by the reducer function, as shown in figure 6-5:

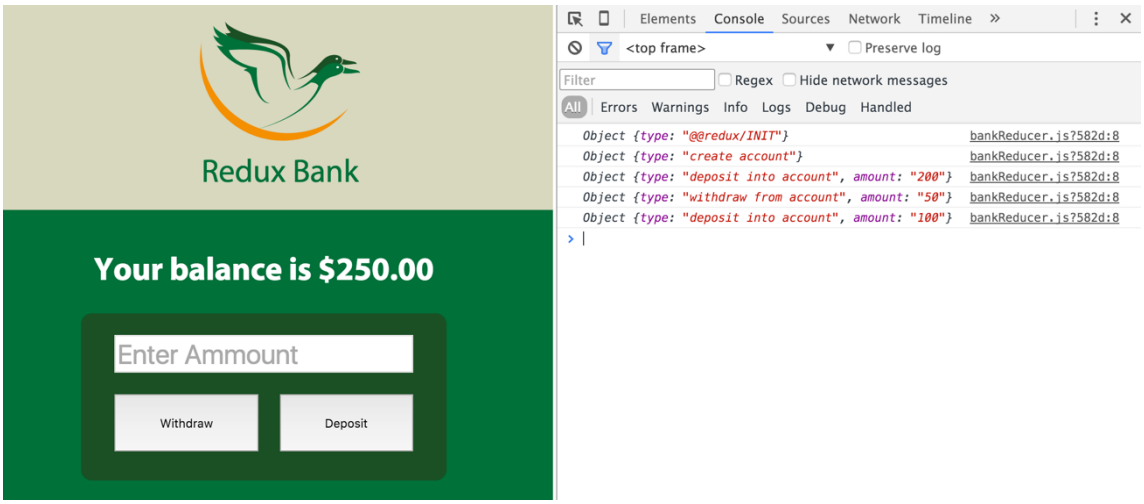

Figure 6-5. The fake banking app with actions logged.

Refactoring the fake Bank Application

Your fake bank application works and uses Redux to manage its state. It has a store that keeps the centralized state of the whole app; the components dispatches actions to notify of things that happened in the application and the store receives the actions and uses the reducer to calculate the new state.

Although everything works, there are a few improvements that can be done right away: setup the balance state on the init action and decouple the action creation from the components.

Init Action

Take a look again on figure 6-11 - Notice that the Redux store always dispatches an INIT action. The fact that an initial action is always automatically dispatched can be used to set up the initial state of the store – thus removing the need for you to manually dispatch an "create account" action. Additionally, in this initial dispatch, the store state is empty, so we can refactor the reducer function to always set up the initial value automatically using an ES6 default function parameter (**Default function parameters** allow formal parameters to be initialized with default values if no value or undefined is passed.). In the end, you will update both the bankReducer.js file (to use a default function parameter to initialize the store State) and the AppContainer component (since it no longer needs to dispatch the initial CREATE_ACCOUNT action).

Listing 6-7 shows the updated bankReducer.js, and listing 6-8 the updated AppContainer within the App.js file.

Listing 6-7. The updated bankReducer.

```
import constants from './constants';

const initialState = {
  balance: 0
}

const bankReducer = (state = initialState, action) => {
    console.log(state, action);

    switch (action.type) {
      case constants.DEPOSIT_INTO_ACCOUNT:
        return {balance: state.balance + parseFloat(action.amount)};

      case constants.WITHDRAW_FROM_ACCOUNT:
        return {balance: state.balance - parseFloat(action.amount)};

      default:
        return state;
    }
}

export default bankReducer;
```

Notice that the CREATE_ACCOUNT case was also removed from the code above.

Listing 6-8. The updated App.js file.

```
import React, { Component, PropTypes } from 'react';
import { render } from 'react-dom';
import bankStore from './bankStore';
import constants from './constants';


class BankApp extends Component {...}
BankApp.propTypes = {...}


class BankAppContainer extends Component {
  componentDidMount() {
    this.unsubscribe = bankStore.subscribe(() =>
      this.setState({balance: bankStore.getState().balance})
    );
  }

  componentWillUnmount() {
    this.unsubscribe();
  }

  render(){
    return(
      <BankApp
         balance={ bankStore.getState().balance }
         onDeposit={ (amount)=>bankStore.dispatch(
                       {type:constants.DEPOSIT_INTO_ACCOUNT, amount:amount} )}
         onWithdraw={ (amount)=>bankStore.dispatch(
                      {type:constants.WITHDRAW_FROM_ACCOUNT, amount:amount} )}
      />
    )
  }
}
```

```
render(<BankAppContainer />, document.getElementById('root'));
```

Notice in the source code for the BankAppContainer that the whole constructor function was removed, since it was no longer necessary.

Action Creators

Another improvement we can make on the application is introduce Action Creators. Remember that an action is simply a JavaScript object with a required "type" key (along with any other desired keys to represent your data). So far, you've been creating these objects inline in the components using literal notation (curly brackets) to dispatch them, but it is a good convention to create separate functions to generate them. For example, instead of this:

```
bankStore.dispatch( {type:constants.WITHDRAW_FROM_ACCOUNT, amount:amount} )
```

You would have a function to generate the object in a separate file:

```
const bankActionCreators = {
  withdrawFromAccount(amount) {
    return {
      type: constants.DEPOSIT_INTO_ACCOUNT,
      amount: amount
    };
  }
}
```

and in your component you would call that function to generate the object instead of using an object literal:

```
bankStore.dispatch( bankActionCreators.withdrawFromAccount(amount) )
```

From the point of view of the Redux library, both are exactly the same. The difference is that now you have decoupled the creation of an action from the actual component dispatching that action. It's a subtle difference but one that can have a big impact on your application in the architectural perspective.

Let's implement this pattern in the fake bank application. Start by creating a new file, bankActionCreators.js, with two functions: depositIntoAccount and withdrawFromAccount, as shown in listing 6-9.

Listing 6-9. The bankActionCreators.js File

```
import constants from './constants';

const bankActionCreators = {
  /**
   * @param  {number} ammount to whithdraw
   */
  depositIntoAccount(amount) {
    return {
      type: constants.DEPOSIT_INTO_ACCOUNT,
      amount: amount
    };
  },

  /**
   * @param  {number} amount to whithdraw
   */
  withdrawFromAccount(amount) {
    return {
      type: constants.WITHDRAW_FROM_ACCOUNT,
      amount: amount
    };
  }
```

```
};

export default bankActionCreators;
```

Next, simply edit the BankAppContainer in app.js file to use the action creator functions instead of object literals for the actions, as shown in listing 6-10:

Listing 6-10. The updated BankAppContainer.

```
import React, { Component, PropTypes } from 'react';
import { render } from 'react-dom';
import bankStore from './bankStore';
import constants from './constants';
import bankActionCreators from './bankActionCreators';

class BankApp extends Component {...}
BankApp.propTypes = {...}

class BankAppContainer extends Component {
  componentDidMount() {
    this.unsubscribe = bankStore.subscribe(() =>
      this.setState({balance: bankStore.getState().balance})
    );
  }

  componentWillUnmount() {
    this.unsubscribe();
  }

  render(){
    return(
      <BankApp
         balance={ bankStore.getState().balance }
         onDeposit={ (amount)=>bankStore.dispatch(   ➥
                      bankActionCreators.depositIntoAccount(amount) ) }
         onWithdraw={ (amount)=>bankStore.dispatch(  ➥
                      bankActionCreators.withdrawFromAccount(amount) ) }
      />
    )
  }
}

render(<BankAppContainer />, document.getElementById('root'));
```

React bindings for Redux

So far, you have managed to build a very basic React + Redux application by installing the Redux library, creating a store and a reducer function and separating your components into presentational components and container components (who actually get to talk to the Redux store). But it's possible to refine and further simplify the work you've done so far. Redux authors also provides a different library called "react-redux" that provide two utilities making it even easier and more straightforward to connect react components to a Redux store. The utilities are:

- A `connect` function. Creating container functions for the purpose of mapping state and dispatch calls to props for child, purely presentational components is such a common practice in React + Redux apps that the React bindings library provides a utility function that does just that: It generates container functions for you. The container functions generated by "connect" will automatically subscribe to the store (and unsubscribe on unmount).
- A `provider` component: Right now, the BankAppComponent source code is using a variable to access the Redux Store. This might work well for this small example, but in bigger applications where the store might be required by many different container components, making sure every one of them has access to the same store (which is not a singleton) can quickly get tricky. Wrapping your application around the Provider component will make sure that every container component created with connect has access to the same store.

Let's see how this works in practice. You will also need to install Redux bindings for React, which come in a separate npm package:

```
npm install --save react-redux
```

In the App.js source code, you will completely remove the BankAppContainer component code and substitute it by a call to connect. Connect is a curried function that you need to call twice. First, call it passing functions to map the store state and dispatch to props. Then, call it again passing the child component (the BankApp, in this example).

Just to recap, in this example, the container component is passing three props down to the BankApp presentational component: balance (mapping to the state from the store), onDeposit and onWithdraw (both mapping to dispatches to the store). The connect function expects you to declare separately the props mapped to store state and the props mapped to store dispatches. Listing 6-11 shows the updated code for the container component only:

Listing 6-11. Automatically generating the BankAppContainer using the connect function.

```
const mapStateToProps = (state) => {
  return {
    balance: state.balance
  }
}

const mapDispatchToProps = (dispatch) => {
  return {
    onDeposit: (amount) => dispatch(bankActionCreators.depositIntoAccount(amount)),
    onWithdraw: (amount) => dispatch(bankActionCreators.withdrawFromAccount(amount))
  }
}

const BankAppContainer = connect(mapStateToProps, mapDispatchToProps)(BankApp)
```

For all effects, the code above generates a container component that behaves exactly the same as before – except that it's smaller, functional and more robust (it automatically deals with edge cases and makes performance improvements).

To finish, don't forget to use the Provider component to make the store available for every connect calls. Listing 6-12 shows the complete source code for App.js, including the updated imports.

Listing 6-12. The updated source code for App.js

```
import React, { Component, PropTypes } from 'react';
import { render } from 'react-dom';
import { connect, Provider } from 'react-redux'
import bankStore from './bankStore';
import constants from './constants';
import bankActionCreators from './bankActionCreators';


class BankApp extends Component {...} // Ommited for brevity
BankApp.propTypes = {...} // Ommited for brevity

// Generate a container app by Mapping state and dispatch to props
const mapStateToProps = (state) => {
  return {
    balance: state.balance
  }
}
const mapDispatchToProps = (dispatch) => {
  return {
    onDeposit: (amount) => dispatch(bankActionCreators.depositIntoAccount(amount)),
    onWithdraw: (amount) => dispatch(bankActionCreators.withdrawFromAccount(amount))
  }
}
const BankAppContainer = connect(mapStateToProps, mapDispatchToProps)(BankApp)


render(
  <Provider store={bankStore}>
    <BankAppContainer />
  </Provider>,
  document.getElementById('root')
);
```

From now on, all container components in this chapter whose purpose is to connect to a Redux store and maps its state and dispatch to props for child, presentational components will be created using the connect function.

Applying a Middleware

Middleware is a core concept in Redux that provides the ability to create functions that extend the library.

When you register a middleware function with a Redux store, it will sit between the dispatcher method and the reducer. Every time an action gets dispatched, Redux will call all the registered middlewares before the action reaches the reducer, allowing the middleware developer to execute any arbitrary code or even make changes to the action. You can see an updated graph representing the whole Redux architecture including the middlewares in Figure 6-6:

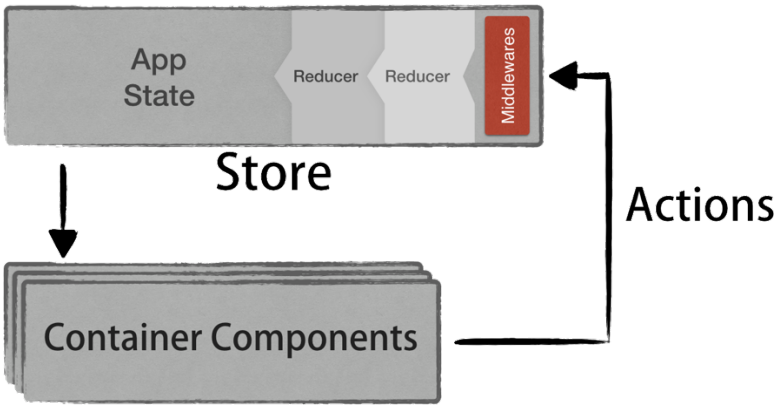

Figure 6-6. The Redux flow.

Common uses for middleware includes logging, crash reporting, talking to an asynchronous API and more. If you remind, in your previous fake bank application example, you inserted a console.log line inside the reducer to log the dispatched actions. At the time, it was mentioned that it was a temporary "hack", so let's remove that line and implement proper logging in the application using a middleware.

To start, remove the "console.log" call from the bankReducer. The updated code is shown below:

Listing 6-13. The bankReducer.js without the console.log call.

```
import constants from './constants';

const initialState = {
  balance: 0
}

const bankReducer = (state = initialState, action) => {
    switch (action.type) {
      case constants.DEPOSIT_INTO_ACCOUNT:
        return {balance: state.balance + parseFloat(action.amount)};

      case constants.WITHDRAW_FROM_ACCOUNT:
        return {balance: state.balance - parseFloat(action.amount)};

      default:
        return state;
    }
}

export default bankReducer;
```

Next, let's create a custom middleware to log all dispatched actions. A middleware must be a curried function that receives the `store`, a "`next`" callback and the current dispatched `action`. You can do any computations you need using any of these parameters, making sure to call the "next" callback with the action in the end, to assure the action continues its flow and reaches the other middlewares and reducers. Using ES6 arrow functions to create the curried function, the bare-bones, absolute minimum Redux middleware looks like this:

```
const myMiddleware = (store) => (next) => (action) => {
  /*
   * Your custom code
   */
```

```
  return next(action);  // Move along, we're done here.
}
```

Following in this template, to create a logger all you need to do is console.log the action being dispatched:

```
const logger = (store) => (next) => (action) => {
  console.log('dispatching:', action);
  return next(action);
}
```

Finally, you need to configure the store to use a middleware. This can be done using the redux `applyMiddleware` method – simply import this method from redux and use it together with createStore to extend the store with the desired middleware. Listing 6-14 shows the updated source code for the bankStore.js file.

Listing 6-14. The updated bankStore.js

```
import { createStore, applyMiddleware } from 'redux'
import bankReducer from './bankReducer';

const logger = (store) => (next) => (action) => {
  console.log('dispatching:', action);
  return next(action);
}

const bankStore = createStore(
  bankReducer,
  applyMiddleware(logger) // enhance the store with the logger middleware
);

export default bankStore;
```

To summarize, you just injected a custom code in the middle of the dispatch process using applyMiddleware. Redux expects a middleware to be curried function, and will call it with 3 parameters: the store, a "next" callback to resume the dispatch execution flow and the current action. In this case, your custom code simply logged the action being dispatched, but it could basically do anything, including transforming the action itself before it reaches the reducers. If you followed along, you should see something like figure 6-7 after making some deposit and withdraw operations:

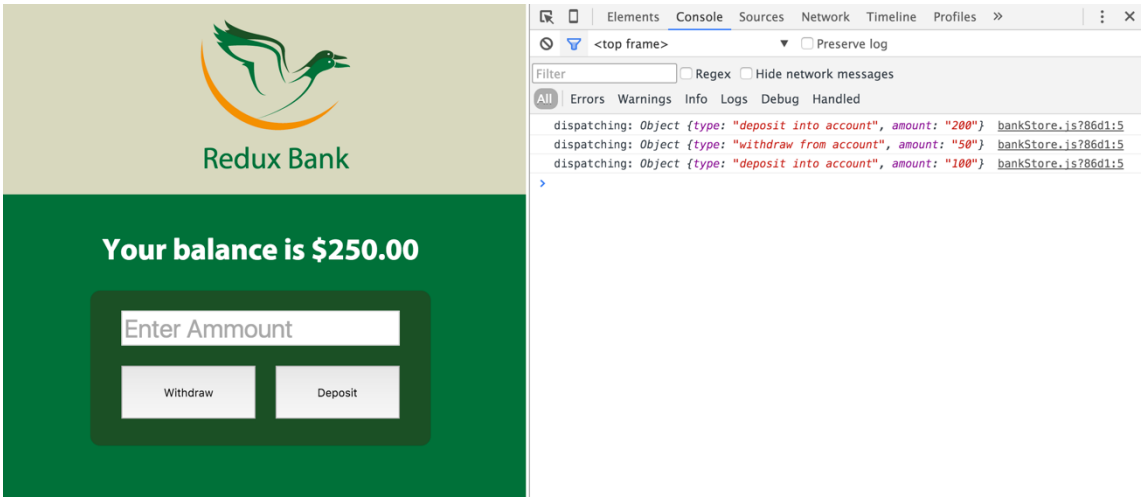

Figure 6-7 – Your own redux middleware.

Working with multiple reducers

As your app grows, you will most certainly have to manage different and unrelated types of state in your application – state relating to the UI, state relating to data the user has entered or data that was fetched from a server… In situations like these, relying on a single reducer to do all your state transformations can get cluttered, so it is a good practice to split your reducer into many different functions - each managing independent parts of the state. Redux provides a `combineReducers` helper function to turn multiple reducing functions into a single reducer that you can pass to store.

To illustrate, let's add a new UI element to the bank application whose state needs to be managed: a toggable list of additional info about the user's bank account, as shown in figure 6-8:

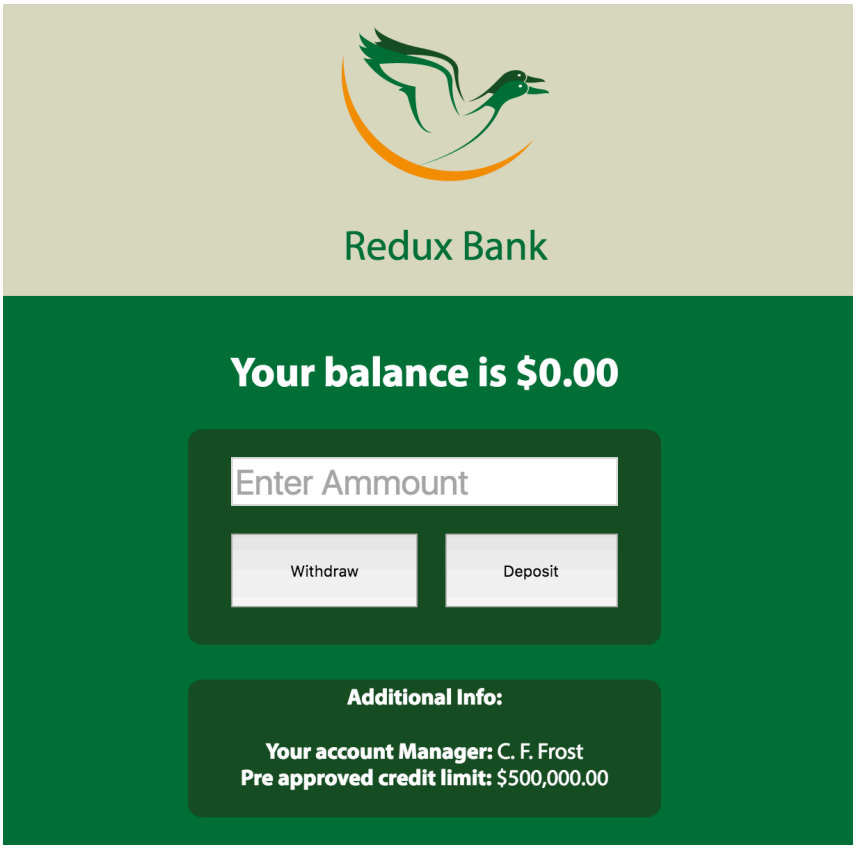

Figure 6-8. Toggable Additional Info

Let's start by adding a new constant and action creator that will be used to dispatch a toggle interaction, as shown in listings 6-15 and 6-16:

Listing 6-15. Adding a new "TOGGLE_INFO" constant.

```
export default {
  WITHDRAW_FROM_ACCOUNT: 'withdraw from account',
  DEPOSIT_INTO_ACCOUNT: 'deposit into account',
  TOGGLE_INFO: 'toggle info'
};
```

Listing 6-16. Adding a new "toggleExchange" action creator.

```
import constants from './constants';

const bankActionCreators = {
  depositIntoAccount(amount) {...}, // Ommited for brevity
  withdrawFromAccount(amount) {...}, // Ommited for brevity

  /**
   * Toggle the visibility of the exchange rate
   */
  toggleInfo() {
```

```
    return {
      type: constants.TOGGLE_INFO
    };
  }

};

export default bankActionCreators;
```

Next, you will refactor the bankReducer. There are a lot of changes to be done:

- Change the object representing the initial state of the application
- Refactor the current account balance reducer to be a separate function
- Create a new reducer function to manage the interface state
- Finally, use the combineReducers helper function to combine and export the balance and ui reducers.

Let's make these changes in steps. First, you will lay out the basic skeleton for the new bankReducer.js with the new initial state, empty functions for each reducer and the combineReducers helper:

Listing 6-17: Starting the bankReducer refactor to use combineReducers.

```
import constants from './constants';
import { combineReducers } from 'redux';

const initialState = {
  initialBalance: 0,
  initialUI: {
    showInfo: true,
  },
}

const balanceReducer = (...) => {

};

const uiReducer = (...) => {

};

const bankReducer = combineReducers({balance: balanceReducer, ui: uiReducer});

export default bankReducer;
```

The `combineReducers` function accepts an object: Each key on this object represents an entry on the centralized App state tree. Each reducer function referenced in this object will manage only this entry and have no access to other values in the application state – which helps in keeping the pieces isolated and organized.

In plain English, the Bank application will have two entries in the centralized app state tree: `balance` and `ui`. The `balanceReducer` function is responsible for managing only the "balance" entry of the App state, while the `uiReducer` is responsible for the "ui" entry.

In sequence, let's reimplement the `balanceReducer.` As you may remember, in the previous incarnation the bankReducer managed an object containing the balance numeric value. The new `balanceReducer`, in turn, will manage the numeric value directly, as shown in listing 6-18

Listing 6-18. The balanceReducer.

```
const balanceReducer = (state = initialState.initialBalance, action) => {
  switch (action.type) {
    case constants.DEPOSIT_INTO_ACCOUNT:
      return state + parseFloat(action.amount);

    case constants.WITHDRAW_FROM_ACCOUNT:
      return state - parseFloat(action.amount);

    default:
      return state;
  }
};
```

Next, let's implement the `uiReducer` function. It will receive an object containing the `showInfo` key and respond to the `TOGGLE_INFO` action type.

Remember that the reducer function should always treat the state as immutable, so you will use the update helper function from the react immutability helpers add-on (don't forget to install it with `npm install --save react-addons-update` and import it.).

The source code for the UI reducer is shown below (listing 6-19):

Listing 6-19. The UIReducer.

```
const uiReducer = (state = initialState.initialUI, action) => {
  switch (action.type) {
    case constants.TOGGLE_INFO:
      return update(state, { showInfo: { $apply: currentState => !currentState }});
    default:
      return state;
  }
};
```

The complete, refactored source code for the bankReducer.js file is shown in listing 6-20 for reference:

Listing 6-20. bankReducer.js refactored source code:

```
import constants from './constants';
import update from 'react-addons-update';
import { combineReducers } from 'redux';

const initialState = {
  initialBalance: 0,
  initialUI: {
    showExchange: true,
  },
}

const balanceReducer = (state = initialState.initialBalance, action) => {
  switch (action.type) {
    case constants.DEPOSIT_INTO_ACCOUNT:
      return state + parseFloat(action.amount);

    case constants.WITHDRAW_FROM_ACCOUNT:
      return state - parseFloat(action.amount);

    default:
      return state;
  }
```

```
};

const uiReducer = (state = initialState.initialUI, action) => {
  switch (action.type) {
    case constants.TOGGLE_INFO:
      return update(state, { showInfo: { $apply: currentState => !currentState }});
    default:
      return state;
  }
};

const bankReducer = combineReducers({balance:balanceReducer, ui: uiReducer});

export default bankReducer;
```

Finally, you need to take care of the component and create the toggable information box. You will enter the markup for the currency exchange widget and add new entries on both mapStateToProps and mapDispatchToProps mapping functions to provide the component with the showExchange Boolean prop and onToggle callback prop. Listing 6-21 shows the updated App.js source code.

Listing 6-21. The updated App.js source code.

```
import React, { Component, PropTypes } from 'react';
import { render } from 'react-dom';
import { connect, Provider } from 'react-redux'
import bankStore from './bankStore';
import constants from './constants';
import bankActionCreators from './bankActionCreators';

class BankApp extends Component {
  handleDeposit() {
    this.props.onDeposit(this.refs.amount.value);
    this.refs.amount.value = '';
  }

  handleWithdraw() {
    this.props.onWithdraw(this.refs.amount.value);
    this.refs.amount.value = '';
  }

  render() {
    return (
      <div>
        <header>
          <img src="//www.pro-react.com/logos/redux-bank.svg" width="150" />Redux Bank
        </header>
        <h1>Your balance is ${(this.props.balance).toFixed(2)}</h1>
        <div className="atm">
          <input type="text" placeholder="Enter Ammount" ref="amount" />
          <button onClick={this.handleWithdraw.bind(this)}>Withdraw</button>
          <button onClick={this.handleDeposit.bind(this)}>Deposit</button>
        </div>

        <div className="exchange" onClick={this.props.onToggle}>
          <strong>Exchange Rates:</strong>
          <div className={this.props.showExchange? 'exchange--visible' : 'exchange--closed'}>
            <strong>$1 USD =</strong>
            <span className="rate">0.9990 EUR</span>
            <span className="rate">0.7989 GBP</span>
            <span className="rate">710.15 JPY</span>
          </div>
        </div>
```

```
      </div>
    );
  }
}
BankApp.propTypes = {
  balance: PropTypes.number,
  showExchange: PropTypes.bool,
  onDeposit: PropTypes.func,
  onWithdraw: PropTypes.func,
  onToggle: PropTypes.func,

};

// Generate a container app by Mapping state and dispatch to props
const mapStateToProps = (state) => {
  return {
    balance: state.balance,
    showExchange: state.ui.showExchange,
  }
}
const mapDispatchToProps = (dispatch) => {
  return {
    onDeposit: (amount) => dispatch(bankActionCreators.depositIntoAccount(amount)),
    onWithdraw: (amount) => dispatch(bankActionCreators.withdrawFromAccount(amount)),
    onToggle: () => dispatch(bankActionCreators.toggleExchange()),
  }
}
const BankAppContainer = connect(mapStateToProps, mapDispatchToProps)(BankApp)


render(
  <Provider store={bankStore}>
    <BankAppContainer />
  </Provider>,
  document.getElementById('root')
);
```

Asynchronous Action Creators

Redux dispatches are synchronous: Every time a dispatch happens, all registered reducers are immediately invoked with the dispatched actions (in the order they were dispatched), and a new state object is produced.

Remember that actions are just plain JavaScript objects with a “type” field and optional data:

```
{ type: ACTION_TYPE, result: 'whatever' };
```

This raises a question: How to deal with asynchronous operations (such as data fetching, for example) in Redux?

The answer is provided by Redux in the form of a middleware: Redux-thunk.

Redux-thunk teaches Redux to recognize a special kind of action - an asynchronous action. An asynchronous action is in fact a function that receives the dispatch method as a parameter and can decide when to dispatch the actual action object. In practice, when using redux thunk you can keep using standard synchronous action creators like this:

```
syncActionCreator() {
  return { type: constants.ACTION_TYPE, result: 'whatever' };
}
```

But you can also have asynchronous action creators like this:

```
asyncActionCreator() {
  return (dispatch) => {
    // Any time consuming async operation (in. eg. data fetching)
    timeConsumingOperation.then((result) => {
      dispatch({ type: constants.ACTION_TYPE, result: result });
    });
  };
}
```

In the sample asyncActionCreator code above, a function is returned instead of an action object. Redux thunk will call this function passing the store's dispatch as a parameter – the function then can perform asynchronous operations and dispatch the actual action object when appropriate. As soon as the actual action object get dispatched, the code is executed in a synchronous fashion from the point of view of the store, reducers and the components, and this makes it easier to reason about them.

■ **Note** What's with the name? "Thunk" is a term used in computer science to describe when a function is used to replace a computation in order to calculate it later, when the actual value is needed.

```
// calculation of 1 + 2 is immediate
// x === 3
let x = 1 + 2;

// calculation of 1 + 2 is delayed
// foo can be called later to perform the calculation
// foo is a thunk!

let foo = () => 1 + 2;
```

Installation and usage

Redux thunk is an external module, so make sure to declare it as a project dependency and install it (both can be done with the command:

```
npm install --save redux-thunk
```

Then, to enable Redux Thunk, use `applyMiddleware` in your Redux Store:

```
import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';
import reducers from './reducers/index';


const store = createStore(
  reducers,
  applyMiddleware(thunk)
);
```

Data Fetching Considerations

Although you can run any sort of asynchronous code inside the asynchronous action creator (including data fetching), a best practice that emerged from the community is to create a separate module to wrap all your data requests and API calls (i.e. a file such as APIutils.js). The API utils can then be called from the asynchronous action creators.

Remember that when we call an asynchronous API, there are two crucial moments in time: the moment we start the call, and the moment when we receive an answer (or an error). For that reason, the asynchronous action creator doing data fetching through the API utility module will always dispatch at least three different actions: An action informing the store that the request began, an action informing the store that the request finished successfully and an action informing the store that the request failed.

Let's exemplify this by creating a new application – a site for airline tickets.

AirCheap Application

The application will fetch a list of airports as soon as it loads, and when the user fills the origin and destination airports, the application will talk to an API to fetch airline ticket prices. Figure 6-9 shows the working application.

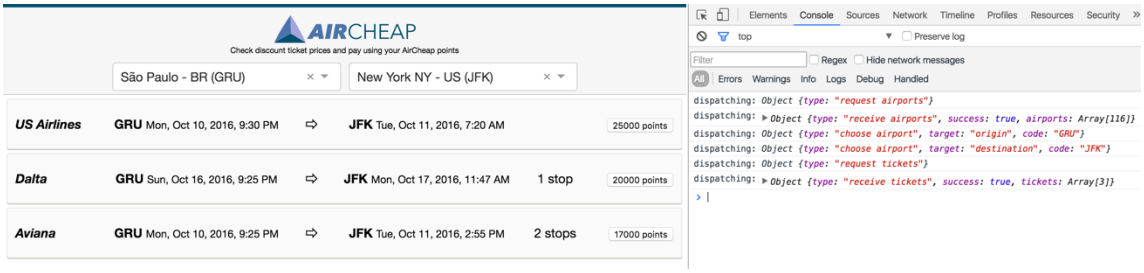

Figure 6-9. The "AirCheap" tickets app.

Setup – project organization and basic files

To start the project in an organized way, we're going to create folders for Redux-related files (action creators, stores as well as an API folder for API utility modules) and a folder for React components. The initial project structure will look like this (Figure 6-10):

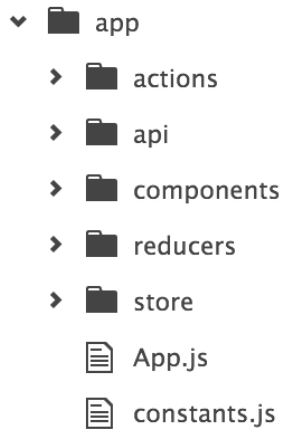

Figure 6-10. The App structure for the AirCheap project.

Let's start creating the project files, beginning with the constants.js file. When developing an application in a real world scenario, you would probably start the constants.js file with just a few constants and increase them as needed – but in our case we already know beforehand all the constants we want to use:

- `REQUEST_AIRPORTS` to name the action we will dispatch as the application starts to fetch all the airports. And since this will be an async operation, we will also create the `RECEIVE_AIRPORTS` constant to represent the success/error on the operation;
- `CHOOSE_AIRPORT` to name a synchronous action of the user selection an airport (as both origin OR destination);
- Finally, we will have the `REQUEST_TICKETS` constant to name the action that we will dispatch when both an origin and a destination are selected. This will be an asynchronous data fetching operation, so we will also need a constant to represent success and, eventually, an error on the fetch operation: `RECEIVE_TICKETS`.

Listing 6-22 shows the final constants.js file.

Listing 6-22. The constants.js file

```
export const REQUEST_AIRPORTS = 'request airports';
export const RECEIVE_AIRPORTS = 'receive airports';
export const CHOOSE_AIRPORT = 'choose airport';
export const REQUEST_TICKETS = 'request tickets';
export const RECEIVE_TICKETS = 'receive tickets';
```

Creating the API Helper and ActionCreators for fetching airports

Let's create an API helper for dealing with the airport and ticket fetching: As we've discussed earlier, creating a segregated helper module to interact with the API will help keep the the action creators clean and minimal. For convenience, there is an online test API available (but you can also place static json files in our public folder containing sample return instead of using the actual remote API – the mock data files are available at the book's github page: https://github.com/pro-react/sample-code/blob/master/chapter%206/aircheap/public). There

are two end-points available: airports (containing a list of the world's busiest airports) and tickets (generate random tickets for a given origin and destination airports).

Let's begin by fetching the airport list in the `api/AirCheapAPI.js` file. It will contain a function called fetchAirports, as shown in Listing 6-23:

Listing 6-23. First take at api/AirCheapAPI.js file

```
import 'whatwg-fetch';

let AirCheapAPI = {
  fetchAirports() {
    return fetch('https://aircheapapi.pro-react.com/airports')
    .then((response) => response.json());
  },

};

export default AirCheapAPI;
```

■ **NOTE** As in earlier examples, we're using the native fetch function to load the json file and importing the 'whatwg-fetch' npm module that provides support for fetch in older browsers. Don't forget to install it with `npm install --save whatwg-fetch`.

Moving to the AirportActionCreators, remember that actions are like messages that gets dispatched through the store and all its reducers: they just communicate what happened to the app - there is no place for business logic or computations on an action - with this in mind, developing an ActionCreator module is pretty straightforward.

You will create and action called fetchAirports that be called from the component. This action creator will return a function (only possible because you're going to use redux-thunk). Since the user will also be able to choose both an origin and destination airport, you will also create an "chooseAirport" action creator. Listing 6-24 shows the AirportActionCreators file:

Listing 6-24. The actions/AirportActionCreators.js file:

```
import { REQUEST_AIRPORTS, RECEIVE_AIRPORTS, CHOOSE_AIRPORT } from '../constants'
import AirCheapAPI from '../api/AirCheapApi';

let AirportActionCreators = {
  // Thunk Action creator
  fetchAirports(origin, destination) {
    return (dispatch) => {
      dispatch({ type: REQUEST_AIRPORTS });
      AirCheapAPI.fetchAirports().then(
        (airports) => dispatch({ type: RECEIVE_AIRPORTS, success:true, airports }),
        (error) => dispatch({ type: RECEIVE_AIRPORTS, success:false })
      );
    };
  },


  // Regular Action creator
  chooseAirport(target, airport) {
    return {
```

```
      type: CHOOSE_AIRPORT,
      target: target,
      code: airport? airport.value : ''
    }
  }

};

export default AirportActionCreators;
```

Reducers

Since the AirCheap application will contain many reducers, we will split them in separate files – Each individual reducer function will be a separate file in the reducers folder; the reducers folder will also contain an index.js whose role is to combine all the individual reducers into one root reducer function that can be used by the store.

The store will then create a Redux store using the root reducer and the thunk middleware. You will also create a custom middleware to log all actions dispatches (as you did earlier on the bank account app).

Let's get started with the airports reducer. It will respond to the "RECEIVE_AIRPORTS" action and populate the state with the loaded airports. Listing 6-25 shows the reducers/airports.js

Listing 6-25 – Airports reducer

```
import { RECEIVE_AIRPORTS } from '../constants';

const airports = (state = [], action) => {
  switch (action.type) {
    case RECEIVE_AIRPORTS:
      return action.airports;
    default:
      return state;
  }
};

export default airports;
```

In sequence, create the route.js reducer. It will act on the "CHOOSE_AIRPORT" action and store the state for the user's selections for both origin and destination airports. Listing 6-26 shows the reducers/route.js:

Listing 6-26 – Route reducer

```
import { CHOOSE_AIRPORT } from '../constants';
import update from 'react-addons-update'

const initialState = {
  origin: '',
  destination: '',
};

const route = (state = initialState, action) => {
  switch (action.type) {
    case CHOOSE_AIRPORT:
      // action.target can be either "origin" or "destination"
      // action.code contains the selected airport code

      return update(state, { [action.target]: { $set: action.code } })
    default:
```

```
      return state;
  }
};

export default route;
```

Finally, you will create index.js file in the reducers folder. Its responsibility is to combine all individual reducers into one root reducer, as shown in listing 6-27:

```
import { combineReducers } from 'redux';
import airports from './airports';
import route from './route';

const rootReducer = combineReducers({
  airports,
  route
});

export default rootReducer;
```

The AirCheap Store

Finally, let's create the AirCheapStore.js. The store will be created with the root reducer and two middlewares: The redux-thunk (for asyncronous requests) and a custom reducer to log all dispatched actions (similar to the one you created for the bank account application). Listing 6-27 shows the store/aircheapStore.js:

Listing 6-27. The Application store.

```
import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';
import reducers from '../reducers';

const logger = (store) => (next) => (action) => {
  if(typeof action !== "function"){
    console.log('dispatching:', action);
  }
  return next(action);
}

const aircheapStore = createStore(
  reducers,
  applyMiddleware(logger, thunk)
);

export default aircheapStore;
```

Notice in the code above that the logger middleware has a small difference from the previous one, used on the Bank Account app – since we're now using redux-thunk to dispatch action creators that may return a function instead of a plain JavaScript object, the code now checks if the action is a function before logging (and only logs objects).

App Component

Next, let's implement the interface for the AirCheap Application. The user will interact with the application by filling two text fields (Origin and Destination), and to make things easier to the user we will implement and auto-suggest feature that suggests airports as the user types, as shown in Figure 6-11.

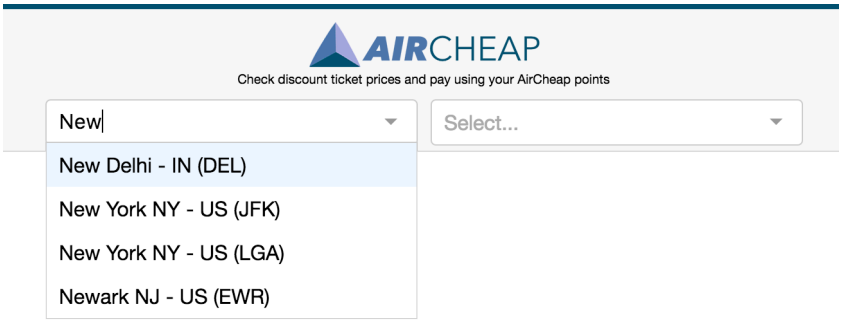

Figure 6-11. Component with auto complete boxes.

There are lots of auto-complete libraries available (as a quick search on npmjs.com reveals) – in this example we will use the react-select, so be sure to install it using NPM (`npm install --save react-select`).

We will start by creating a basic structure for our App component: It will contain the markup (including the two Select components). Listing 6-28 shows the basic structure for the App.js file:

Listing 6-28. The basic app.js basic component structure

```
import React, { Component, PropTypes } from 'react';

import { render } from 'react-dom';
import { Provider, connect } from 'react-redux';
import aircheapStore from './store/aircheapStore';
import Select from 'react-select';
import AirportActionCreators from './actions/AirportActionCreators';

class App extends Component {

  render() {
    return (
      <div>
        <header>
          <div className="header-brand">
            <img src="logo.png" height="35"/>
            <p>Check discount ticket prices and pay using your AirCheap points</p>
          </div>
          <div className="header-route">

           <Select
               name="origin"
               value={this.props.origin}
               options={this.props.airports}
           />

          <Select
               name="destination"
               value={this.props.destination}
               options={this.props.airports}
           />

          </div>

        </header>
      </div>
```

```
    );
  }
}
App.propTypes = {
  airports: PropTypes.array.isRequired,
  origin: PropTypes.string,
  destination: PropTypes.string,
};
```

Notice in the code above that the Select component requires three props: A name, the complete list of options to display (the airports list) and the selected value.

In the sequence, let's use the Redux `connect` function to create a Container component mapping state and dispatches to props. The necessary state from the store are already outlined in the component Props (Airports, origin and destination), and we will also map the dispatch of our action creators: fetchAirports and chooseAirport. Listing 6-29 shows the mapStateToProps and mapDispatchToProps functions as well as the usage of Redux's connect and Provider to create a container component for the Application and make the store available.

Listing 6-29. Configuring Redux in the App.js.

```
const mapStateToProps = (state) => (
  {
    airports: state.airports
      .map(airport => ({
         value: airport.code,
         label: `${airport.city} - ${airport.country} (${airport.code})`
      })
    ),
    origin: state.route.origin,
    destination: state.route.destination
  }
);

const mapDispatchToProps = (dispatch) => (
  {
    fetchAirports: () => dispatch(AirportActionCreators.fetchAirports()),
    onChooseAirport: (target, airport) => dispatch(            ➥
       AirportActionCreators.chooseAirport(target, airport)    ➥
    )
  }
);

const AppContainer = connect(mapStateToProps, mapDispatchToProps)(App);

render(
  <Provider store={aircheapStore}>
    <AppContainer />
  </Provider>,
  document.getElementById('root')
);
```

Notice in the code above how, in the `mapStateToProps` function, how the airport list is mapped to produce a prettier label output in the format "city name – country initials (airport code)".

Next, let's wire the `fetchAirports` and `onChooseAirport` dispatches. The `fetchAirports` will be dispatched on the `componentDidMount` lifecycle method to trigger the async loading of the airports immediately at component render. Additionally, the Select components will dispatch `onChooseAirport` when the user chooses an origin or

destination airport (Notice that by using JavaScript's bind function you can have just one callback function and pass a different parameter for each field.). The updated code is shown in listing 6-30.

Listing 6-30. Dispatching Action Creators.

```
class App extends Component {

  componentDidMount(){
    this.props.fetchAirports();
  }

  render() {
    return (
      <div>
        <header>
          <div className="header-brand">...</div>
          <div className="header-route">

           <Select
               name="origin"
               value={this.props.origin}
               options={this.props.airports}
               onChange={this.props.onChooseAirport.bind(this,'origin')}
           />

          <Select
               name="destination"
               value={this.props.destination}
               options={this.props.airports}
               onChange={this.props.onChooseAirport.bind(this,'destination')}
           />

          </div>
        </header>
      </div>
    );
  }
}
App.propTypes = {
  airports: PropTypes.array.isRequired,
  origin: PropTypes.string,
  destination: PropTypes.string,
  fetchAirports: PropTypes.func.isRequired,
  onChooseAirport: PropTypes.func.isRequired,
};
```

For convenience, the complete App.js file is shown below, in listing 6-31.

Listing 6-31. The complete source code for the App.js.

```
import React, { Component, PropTypes } from 'react';

import { render } from 'react-dom';
import { Provider, connect } from 'react-redux';
import aircheapStore from './store/aircheapStore';
import Select from 'react-select';
import AirportActionCreators from './actions/AirportActionCreators';

class App extends Component {

  componentDidMount(){
```

```
    this.props.fetchAirports();
  }

  render() {
    return (
      <div>
        <header>
          <div className="header-brand">
            <img src="logo.png" height="35"/>
            <p>Check discount ticket prices and pay using your AirCheap points</p>
          </div>
          <div className="header-route">

           <Select
               name="origin"
               value={this.props.origin}
               options={this.props.airports}
               onChange={this.props.onChooseAirport.bind(this,'origin')}
           />

          <Select
               name="destination"
               value={this.props.destination}
               options={this.props.airports}
               onChange={this.props.onChooseAirport.bind(this,'destination')}
           />

          </div>

        </header>
      </div>
    );
  }
}
App.propTypes = {
  airports: PropTypes.array.isRequired,
  origin: PropTypes.string,
  destination: PropTypes.string,
  fetchAirports: PropTypes.func.isRequired,
  onChooseAirport: PropTypes.func.isRequired,
};


const mapStateToProps = (state) => (
  {
    airports: state.airports
      .map(airport => ({
        value: airport.code,
        label: `${airport.city} - ${airport.country} (${airport.code})`
    })),
    origin: state.route.origin,
    destination: state.route.destination,
  }
);

const mapDispatchToProps = (dispatch) => (
  {
    fetchAirports: () => dispatch(AirportActionCreators.fetchAirports()),
    onChooseAirport: (target, airport) => dispatch(
      AirportActionCreators.chooseAirport(target, airport)
    ),
  }
);

const AppContainer = connect(mapStateToProps, mapDispatchToProps)(App);
```

```
render(
  <Provider store={aircheapStore}>
    <AppContainer />
  </Provider>,
  document.getElementById('root')
);
```

Additionally, a matching CSS file with the application style is shown in listing 6-32.

Listing 6-32. The AirCheap Application Style Sheet.

```
* {
  box-sizing: border-box;
}
body {
  margin: 0;
  font-family: "Helvetica Neue", Helvetica, Arial, sans-serif;
}
header {
  padding-top: 10px;
  border-bottom: 1px solid #ccc;
  border-top: 4px solid #08516E;
  height: 115px;
  background-color: #f6f6f6;
}
p {
  margin:0;
  font-size: 10px;
}
.header-brand {
  text-align: center;
}
.header-route {
  margin-top: 10px;
  margin-left: calc(50% - 300px)
}
.Select {
  position: relative;
  width: 290px;
  float: left;
  margin: 0 5px;
}
.Select-control {
  background-color: #fff;
  border-radius: 4px;
  border: 1px solid #ccc;
  display: table;
  width: 100%;
}
.Select-placeholder {
  color: #aaa;
}
.Select-placeholder,
.Select-control .Select-value {
  line-height: 34px;
  padding: 0 10px;
  position: absolute;
```

```
}
.Select-input {
  height: 34px;
  padding: 0 10px;
}
.Select-input > input {
  background: none transparent;
  border: 0 none;
  font-size: inherit;
  outline: none;
  padding: 7px 0 12px;
}
.has-value.is-pseudo-focused .Select-input {
  opacity: 0;
}
.Select-clear-zone, .Select-arrow-zone {
  color: #999;
  cursor: pointer;
  display: table-cell;
  width: 17px;
  padding-right: 5px;
}
.Select-arrow {
  border-color: #999 transparent transparent;
  border-style: solid;
  border-width: 5px 5px 2.5px;
  display: inline-block;
}
.Select-menu-outer {
  background-color: #fff;
  border: 1px solid #ccc;
  margin-top: -3px;
  max-height: 200px;
  position: absolute;
  width: 100%;
  z-index: 1;
}
.Select-menu {
  max-height: 198px;
  overflow-y: auto;
}
.Select-option {
  cursor: pointer;
  padding: 8px 10px;
}
.Select-option.is-focused {
  background-color: rgba(0, 126, 255, 0.08);
}
```

Finishing the AirCheap Application: Loading tickets

You're fetching the airport data asynchronously as soon as the app component mounts, and as the user interacts with the select field, the app is dispatching a `chooseAirport` action creator - but there's one more fetch to be done: we need to fetch the actual ticket list when the user chooses the desired origin and destination.

The process is very similar to what we did for fetching airports: We will put all the code that handles the actual data fetching in an API helper module. We will create Action Creators to signal the data fetching steps (loading initiated and loaded data successfully or error in loading) and make a new reducer to keep the loaded tickets in its state. Finally, you will create a new `TicketItem` component to display information for each loaded ticket.

Let's begin by editing the AirCheapApi.js module to add methods to fetch the tickets and dispatch the corresponding Actions. Listing 6-33 shows the updated file:

Listing 6-33. The updated AirCheapAPI.js file to fetch tickets.

```
import 'whatwg-fetch';

let AirCheapAPI = {
  fetchAirports() {
    return fetch('https://aircheapapi.pro-react.com/airports')
    .then((response) => response.json());
  },

  fetchTickets(origin, destination) {
    return fetch(
      `https://aircheapapi.pro-react.com/tickets?origin=${origin}&destination=${destination}`
    )
    .then((response) => response.json());
  }
};

export default AirCheapAPI;
```

Action Creators

Moving on, let's edit the AirportActionCreators.js file. You will add a new asynchronous action creator that will be handled by redux-thunk. Listing 6-34 shows the updated AirportActionCreators:

Listing 6-34. Adding action creators for ticket fetching:

```
import { REQUEST_AIRPORTS,
         RECEIVE_AIRPORTS,
         CHOOSE_AIRPORT,
         REQUEST_TICKETS,
         RECEIVE_TICKETS } from '../constants'
import AirCheapAPI from '../api/AirCheapApi';

let AirportActionCreators = {
  fetchAirports(origin, destination) {
    return (dispatch) => {
      dispatch({ type: REQUEST_AIRPORTS });
      AirCheapAPI.fetchAirports().then(
        (airports) => dispatch({ type: RECEIVE_AIRPORTS, success:true, airports }),
        (error) => dispatch({ type: RECEIVE_AIRPORTS, success:false })
      );
    };
  },

  chooseAirport(target, airport) {
    return {
      type: CHOOSE_AIRPORT,
      target: target,
      code: airport? airport.value : ''
    }
  },

  fetchTickets(origin, destination) {
```

```
    return (dispatch) => {
      dispatch({ type: REQUEST_TICKETS });
      AirCheapAPI.fetchTickets(origin, destination).then(
        (tickets) => dispatch({ type: RECEIVE_TICKETS, success: true, tickets }),
        (error) => dispatch({ type: RECEIVE_TICKETS, success: false })
      );
    }
  }
};

export default AirportActionCreators;
```

Ticket Reducer

You will next create a new reducer (thickets.js) which will hold the list of airline tickets and will update its state when the RECEIVE_TICKETS action is dispatched. Listing 6-35 shows the complete source:

Listing 6-35. Source code for the reducers/tickets.js file

```
import { REQUEST_TICKETS, RECEIVE_TICKETS } from '../constants';

const tickets = (state = [], action) => {
  switch (action.type) {
    case REQUEST_TICKETS:
      return [];
    case RECEIVE_TICKETS:
      return action.tickets;
    default:
      return state;
  }
};

export default tickets;
```

Notice that the tickets reducer also responds to the REQUEST_TICKETS action by resetting its state to an empty array. This way, every time we try to fetch different tickets, the interface can be immediately updated to clear any previous tickets that may exist.

Interface Components

Let's begin our work on the interface creating a new component, the TicketItem.js. It will receive the component info as a prop and display a single ticket row.

The tickets service returns a JSON structure that looks like this (for each ticket):

```
{
  "id": "fc704c16fd79",
  "company": "US Airlines",
  "points": 25000,
  "duration": 590,
  "segment": [
    {"duration": 590,
    "departureTime": "2016-05-25T17:45:00.000Z",
    "arrivalTime": "2016-05-26T03:35:00.000Z",
    "origin": "GRU",
```

```
    "destination": "JFK"}
  ]
}
```

With this data signature in mind, the component's code is shown in listing 6-36.

Listing 6-36. The components/TicketItem.js component:

```
import React, { Component, PropTypes } from 'react';

// Default data configuration
const dateConfig = {
  weekday: "short",
  year: "numeric",
  month: "short",
  day: "numeric",
  hour: "2-digit",
  minute: "2-digit"
};

class TicketItem extends Component {
  render() {
    let {ticket} = this.props;
    let departureTime = new Date(ticket.segment[0].departureTime)  ➥
                        .toLocaleDateString("en-US",dateConfig);
    let arrivalTime = new Date(ticket.segment[ticket.segment.length-1].arrivalTime)  ➥
                      .toLocaleDateString("en-US",dateConfig);

    let stops;
    if(ticket.segment.length-1 === 1)
      stops = '1 stop';
    else if(ticket.segment.length-1 > 1)
      stops = ticket.segment.length-1 + ' stops';

    return(
      <div className='ticket'>
        <span className="ticket-company">{ticket.company}</span>
        <span className="ticket-location">
          <strong>{ticket.segment[0].origin}</strong>{' '}
          <small>{departureTime}</small>
        </span>
        <span className="ticket-separator">
        &#8680;
        </span>
        <span className="ticket-location">
          <strong>{ticket.segment[ticket.segment.length-1].destination}</strong>{' '}
          <small>{arrivalTime}</small>
        </span>
        <span className="ticket-connection">
          {stops}
        </span>
        <span className="ticket-points">
          <button>{ticket.points} points</button>
        </span>
      </div>
    );
  }
}
TicketItem.propTypes = {
  ticket: PropTypes.shape({
    id: PropTypes.string,
    company: PropTypes.string,
```

```
    points: PropTypes.number,
    duration: PropTypes.number,
    segment: PropTypes.array
  }),
};

export default TicketItem;
```

Let's also add some CSS styling to the TicketItem component (Shown below, listing 6-37):

Listing 6-37. Additional CSS rules for the TicketItem component.

```
.ticket {
  padding: 20px 10px;
  background-color: #fafafa;
  margin: 5px;
  border: 1px solid #e5e5df;
  border-radius: 3px;
  box-shadow: 0 1px 0 rgba(0, 0, 0, 0.25);
}
.ticket span {
  display: inline-block;
}
.ticket-company {
  font-weight: bold;
  font-style: italic;
  width: 13%;
}
.ticket-location {
  text-align: center;
  width: 29%;
}
.ticket-separator {
  text-align: center;
  width: 6%;
}
.ticket-connection {
  text-align: center;
  width: 10%;
}
.ticket-points {
  width: 13%;
  text-align: right;
}
```

In the sequence, let's update the main App component. There are a few things we need to do:

- Map the tickets state to component props and the fetchTickets action to a component dispatch:

```
const mapStateToProps = (state) => (
  {
    airports: state.airports
      .map(airport => ({
         value: airport.code,
         label: `${airport.city} - ${airport.country} (${airport.code})`
    })),
    origin: state.route.origin,
    destination: state.route.destination,
    tickets: state.tickets,
  }
);
```

```
const mapDispatchToProps = (dispatch) => (
  {
    fetchAirports: () => dispatch(AirportActionCreators.fetchAirports()),
    onChooseAirport: (target, airport) => dispatch(
      AirportActionCreators.chooseAirport(target, airport)
    ),
    fetchTickets: (origin, destination) => dispatch(
      AirportActionCreators.fetchTickets(origin, destination)
    )
  }
);
```

- Dispatch the fetchTickets action creator when the user choosed both an origin and a destination airports. You can do this on the componentWillUpdate lifecycle method – every time the user selects an airport we invoke the `chooseAirport` action creator, and as a consequence the store will dispatch a change event and the App component will be updated. You will check for two things before invoking the action creator: if both origin and destination were chosen and if either one has changed since the last update (so we only fetch once):

```
componentWillUpdate(nextProps, nextState){
  let originAndDestinationSelected = nextProps.origin && nextProps.destination;
  let selectionHasChangedSinceLastUpdate = nextProps.origin !== this.props.origin ||
                                           nextProps.destination !== this.props.destination;
  if(originAndDestinationSelected && selectionHasChangedSinceLastUpdate){
    this.props.fetchTickets(nextProps.origin, nextProps.destination);
  }
}
```

- Finally, import and implement the TicketItem component we just created to show the loaded tickets:

```
render() {
  let ticketList = this.state.tickets.map((ticket)=>(
    <TicketItem key={ticket.id} ticket={ticket} />
  ));
  return (
    <div>
      <header>
        <div className="header-brand">...</div>
        <div className="header-route">
          <Select name='origin' ... />
          <Select name='destination' ... />
        </div>
      </header>
      <div>
        {ticketList}
      </div>

    </div>
  );
 }
```

Listing 6-38 shows the complete updated App component with all the mentioned changes.

Listing 6-38. The updated App component:

```
import React, { Component, PropTypes } from 'react';

import { render } from 'react-dom';
import { Provider, connect } from 'react-redux';
import aircheapStore from './store/aircheapStore';
import Autosuggest from 'react-autosuggest-legacy';
import Select from 'react-select';
import TicketItem from './components/TicketItem';
import AirportActionCreators from './actions/AirportActionCreators';

class App extends Component {

  componentDidMount(){
    this.props.fetchAirports();
  }

  componentWillUpdate(nextProps, nextState){
    let originAndDestinationSelected = nextProps.origin && nextProps.destination;
    let selectionHasChangedSinceLastUpdate = nextProps.origin !== this.props.origin ||
                                             nextProps.destination !== this.props.destination;
    if(originAndDestinationSelected && selectionHasChangedSinceLastUpdate){
      this.props.fetchTickets(nextProps.origin, nextProps.destination);
    }
  }

  render() {
    let ticketList = this.props.tickets.map((ticket)=>(
      <TicketItem key={ticket.id} ticket={ticket} />
    ));
    return (
      <div>
        <header>
          <div className="header-brand">
            <img src="logo.png" height="35"/>
            <p>Check discount ticket prices and pay using your AirCheap points</p>
          </div>
          <div className="header-route">

           <Select
               name="origin"
               value={this.props.origin}
               options={this.props.airports}
               onChange={this.props.onChooseAirport.bind(this,'origin')}
           />

          <Select
               name="destination"
               value={this.props.destination}
               options={this.props.airports}
               onChange={this.props.onChooseAirport.bind(this,'destination')}
           />

          </div>

        </header>
        <div>
          { ticketList }
        </div>
      </div>
    );
  }
}
App.propTypes = {
  airports: PropTypes.array.isRequired,
```

```
  origin: PropTypes.string,
  destination: PropTypes.string,
  tickets: PropTypes.array.isRequired,
  fetchAirports: PropTypes.func.isRequired,
  onChooseAirport: PropTypes.func.isRequired,
  fetchTickets: PropTypes.func.isRequired,
};


const mapStateToProps = (state) => (
  {
    airports: state.airports
      .map(airport => ({
        value: airport.code,
        label: `${airport.city} - ${airport.country} (${airport.code})`
      })
    ),
    origin: state.route.origin,
    destination: state.route.destination,
    tickets: state.tickets,
  }
);

const mapDispatchToProps = (dispatch) => (
  {
    fetchAirports: () => dispatch(AirportActionCreators.fetchAirports()),
    onChooseAirport: (target, airport) => dispatch(
       AirportActionCreators.chooseAirport(target, airport)
    ),
    fetchTickets: (origin, destination) => dispatch(
      AirportActionCreators.fetchTickets(origin, destination)
    )
  }
);

const AppContainer = connect(mapStateToProps, mapDispatchToProps)(App);

render(
  <Provider store={aircheapStore}>
    <AppContainer />
  </Provider>,
  document.getElementById('root')
);
```

If you test the AirCheap application now you should be able to see a list of airports, and a random set of tickets should be loading when you select both an origin and a destination airport (as shown in figure 6-12.)

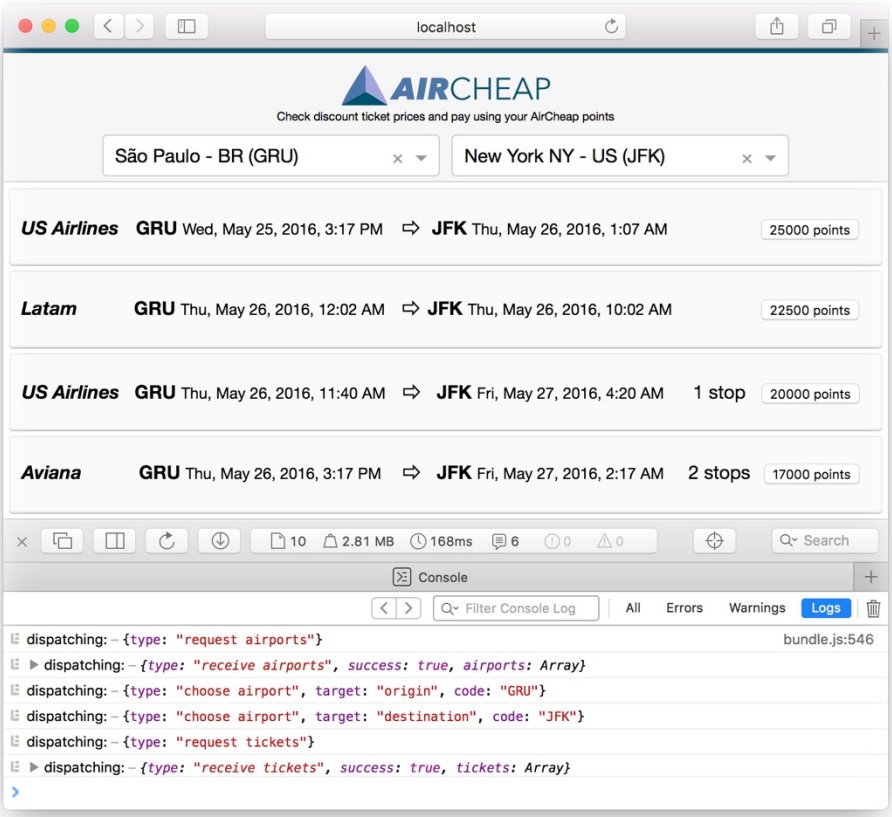

Figure 6-12. The running Aircheap application.

Summary

In this chapter you learned what is Redux and which problems it solves. We've seen how to integrate Redux in a React Application and how to architect complex applications including async API communication.