Front End Development Libraries

# React and Redux

Now that you know how to manage the flow of shared data with Redux, it's time to combine that knowledge with React. In the React and Redux courses, you'll build a React component and learn how to manage state locally at the component level, and throughout the entire application with Redux.

## Introduction

This series of challenges introduces how to use Redux with React. First, here's a review of some of the key principles of each technology. React is a view library that you provide with data, then it renders the view in an efficient, predictable way. Redux is a state management framework that you can use to simplify the management of your application's state. Typically, in a React Redux app, you create a single Redux store that manages the state of your entire app. Your React components subscribe to only the pieces of data in the store that are relevant to their role. Then, you dispatch actions directly from React components, which then trigger store updates.

Although React components can manage their own state locally, when you have a complex app, it's generally better to keep the app state in a single location with Redux. There are exceptions when individual components may have local state specific only to them. Finally, because Redux is not designed to work with React out of the box, you need to use the react-redux package. It provides a way for you to pass Redux state and dispatch to your React components as props.

Over the next few challenges, first, you'll create a simple React component which allows you to input new text messages. These are added to an array that's displayed in the view. This should be a nice review of what you learned in the React lessons. Next, you'll create a Redux store and actions that manage the state of the messages array. Finally, you'll use react-redux to connect the Redux store with your component, thereby extracting the local state into the Redux store.

# Format:

## Syntax:

# Topics Covered:

## Getting Started with React Redux

First, you'll create a simple React component which allows you to input new text messages. These are added to an array that's displayed in the view. This should be a nice review of what you learned in the React lessons.

## Example:

class DisplayMessages extends React.Component {

  // Change code below this line

  constructor(props){

    super(props);

    this.state = {

      input: "",

      messages: []

    }

  }

  // Change code above this line

  render() {

    return <div />

  }

};

## Manage State Locally First

### Explanation:

Here you'll finish creating the DisplayMessages component.

### Example:

class DisplayMessages extends React.Component {

constructor(props) {

super(props);

this.state = {

input: '',

messages: []

}

this.handleChange = this.handleChange.bind(this);

this.submitMessage = this.submitMessage.bind(this);

}

// Add handleChange() and submitMessage() methods here

handleChange(event){

this.setState({input: event.target.value});

}

submitMessage(){

this.setState({messages: this.state.messages.concat(this.state.input)});

this.setState({input: ''});

}

render() {

const myMessages = this.state.messages.map((input) => <li>{input}</li>);

return (

<div>

<h2>Type in a new Message:</h2>

{ /\* Render an input, button, and ul below this line \*/ }

<input value={this.state.input} onChange={this.handleChange}/>

<button onClick={this.submitMessage}>Submit</button>

<ul>You Entered following Data:

{myMessages}

</ul>

{ /\* Change code above this line \*/ }

</div>

);

}

};

## Extract State Logic to Redux

### Explanation:

Now that you finished the React component, you need to move the logic it's performing locally in its state into Redux. This is the first step to connect the simple React app to Redux. The only functionality your app has is to add new messages from the user to an unordered list. The example is simple in order to demonstrate how React and Redux work together.

### Example:

// Define ADD, addMessage(), messageReducer(), and store here:

const ADD = 'ADD';

const defaultState = [];

const messageReducer = (state=defaultState, action) => {

  switch (action.type){

    case ADD:

      return state.concat(action.message);

    default:

      return state;

  }

};

const addMessage = (input) =>{

  return {

    type: ADD,

    message: input};

};

const store = Redux.createStore(messageReducer);

//store.dispatch(addMessage("Hello"));

## Use Provider to Connect Redux to React

### Explanation:

In the last challenge, you created a Redux store to handle the messages array and created an action for adding new messages. The next step is to provide React access to the Redux store and the actions it needs to dispatch updates. React Redux provides its react-redux package to help accomplish these tasks.

React Redux provides a small API with two key features: Provider and connect. Another challenge covers connect. The Provider is a wrapper component from React Redux that wraps your React app. This wrapper then allows you to access the Redux store and dispatch functions throughout your component tree. Provider takes two props, the Redux store and the child components of your app. Defining the Provider for an App component might look like this:

<Provider store={store}>

<App/>

</Provider>

### Example:

// Redux:

const ADD = 'ADD';

const addMessage = (message) => {

  return {

    type: ADD,

    message

  }

};

const messageReducer = (state = [], action) => {

  switch (action.type) {

    case ADD:

      return [

        ...state,

        action.message

      ];

    default:

      return state;

  }

};

const store = Redux.createStore(messageReducer);

// React:

class DisplayMessages extends React.Component {

  constructor(props) {

    super(props);

    this.state = {

      input: '',

      messages: []

    }

    this.handleChange = this.handleChange.bind(this);

    this.submitMessage = this.submitMessage.bind(this);

  }

  handleChange(event) {

    this.setState({

      input: event.target.value

    });

  }

  submitMessage() {

    this.setState((state) => {

      const currentMessage = state.input;

      return {

        input: '',

        messages: state.messages.concat(currentMessage)

      };

    });

  }

  render() {

    return (

      <div>

        <h2>Type in a new Message:</h2>

        <input

          value={this.state.input}

          onChange={this.handleChange}/><br/>

        <button onClick={this.submitMessage}>Submit</button>

        <ul>

          {this.state.messages.map( (message, idx) => {

              return (

                 <li key={idx}>{message}</li>

              )

            })

          }

        </ul>

      </div>

    );

  }

};

const Provider = ReactRedux.Provider;

class AppWrapper extends React.Component {

  // Render the Provider below this line

  render(){

    return(

    <Provider store={store}>

      <DisplayMessages />

    </Provider>

    );

  }

  // Change code above this line

};

## Map State to Props

### Explanation:

The Provider component allows you to provide state and dispatch to your React components, but you must specify exactly what state and actions you want. This way, you make sure that each component only has access to the state it needs. You accomplish this by creating two functions: mapStateToProps() and mapDispatchToProps().

In these functions, you declare what pieces of state you want to have access to and which action creators you need to be able to dispatch. Once these functions are in place, you'll see how to use the React Redux connect method to connect them to your components in another challenge.

Note: Behind the scenes, React Redux uses the store.subscribe() method to implement mapStateToProps().

### Explanation:

const state = [];

// Change code below this line

function mapStateToProps(state) {

  return {messages: state};

}

## Map Dispatch to Props

### Explanation:

The mapDispatchToProps() function is used to provide specific action creators to your React components so they can dispatch actions against the Redux store. It's similar in structure to the mapStateToProps() function you wrote in the last challenge. It returns an object that maps dispatch actions to property names, which become component props. However, instead of returning a piece of state, each property returns a function that calls dispatch with an action creator and any relevant action data. You have access to this dispatch because it's passed in to mapDispatchToProps() as a parameter when you define the function, just like you passed state to mapStateToProps(). Behind the scenes, React Redux is using Redux's store.dispatch() to conduct these dispatches with mapDispatchToProps(). This is similar to how it uses store.subscribe() for components that are mapped to state.

For example, you have a loginUser() action creator that takes a username as an action payload. The object returned from mapDispatchToProps() for this action creator would look something like:

{

submitLoginUser: function(username) {

dispatch(loginUser(username));

}

}

### Example:

const addMessage = (message) => {

  return {

    type: 'ADD',

    message: message

  }

};

// Change code below this line

const mapDispatchToProps = (dispatch) => {

  return {

    submitNewMessage: function(message) {

      dispatch(addMessage(message))

    }

  }

};

## Connect Redux to React

### Explanation:

Now that you've written both the mapStateToProps() and the mapDispatchToProps() functions, you can use them to map state and dispatch to the props of one of your React components. The connect method from React Redux can handle this task. This method takes two optional arguments, mapStateToProps() and mapDispatchToProps(). They are optional because you may have a component that only needs access to state but doesn't need to dispatch any actions, or vice versa.

To use this method, pass in the functions as arguments, and immediately call the result with your component. This syntax is a little unusual and looks like:

connect(mapStateToProps, mapDispatchToProps)(MyComponent)

Note: If you want to omit one of the arguments to the connect method, you pass null in its place.

### Example:

const addMessage = (message) => {

  return {

    type: 'ADD',

    message: message

  }

};

const mapStateToProps = (state) => {

  return {

    messages: state

  }

};

const mapDispatchToProps = (dispatch) => {

  return {

    submitNewMessage: (message) => {

      dispatch(addMessage(message));

    }

  }

};

class Presentational extends React.Component {

  constructor(props) {

    super(props);

  }

  render() {

    return <h3>This is a Presentational Component</h3>

  }

};

const connect = ReactRedux.connect;

// Change code below this line

ReactRedux.connect

const ConnectedComponent = connect(mapStateToProps, mapDispatchToProps)(Presentational)

## Connect Redux to the Messages App

### Explanation:

Now that you understand how to use connect to connect React to Redux, you can apply what you've learned to your React component that handles messages.

In the last lesson, the component you connected to Redux was named Presentational, and this wasn't arbitrary. This term generally refers to React components that are not directly connected to Redux. They are simply responsible for the presentation of UI and do this as a function of the props they receive. By contrast, container components are connected to Redux. These are typically responsible for dispatching actions to the store and often pass store state to child components as props.

### Example:

// Redux:

const ADD = 'ADD';

const addMessage = (message) => {

  return {

    type: ADD,

    message: message

  }

};

const messageReducer = (state = [], action) => {

  switch (action.type) {

    case ADD:

      return [

        ...state,

        action.message

      ];

    default:

      return state;

  }

};

const store = Redux.createStore(messageReducer);

// React:

class Presentational extends React.Component {

  constructor(props) {

    super(props);

    this.state = {

      input: '',

      messages: []

    }

    this.handleChange = this.handleChange.bind(this);

    this.submitMessage = this.submitMessage.bind(this);

  }

  handleChange(event) {

    this.setState({

      input: event.target.value

    });

  }

  submitMessage() {

    this.setState((state) => {

      const currentMessage = state.input;

      return {

        input: '',

        messages: state.messages.concat(currentMessage)

      };

    });

  }

  render() {

    return (

      <div>

        <h2>Type in a new Message:</h2>

        <input

          value={this.state.input}

          onChange={this.handleChange}/><br/>

        <button onClick={this.submitMessage}>Submit</button>

        <ul>

          {this.state.messages.map( (message, idx) => {

              return (

                 <li key={idx}>{message}</li>

              )

            })

          }

        </ul>

      </div>

    );

  }

};

// React-Redux:

const mapStateToProps = (state) => {

  return { messages: state }

};

const mapDispatchToProps = (dispatch) => {

  return {

    submitNewMessage: (newMessage) => {

       dispatch(addMessage(newMessage))

    }

  }

};

const Provider = ReactRedux.Provider;

const connect = ReactRedux.connect;

// Define the Container component here:

const Container = connect(mapStateToProps, mapDispatchToProps)(Presentational)

class AppWrapper extends React.Component {

  constructor(props) {

    super(props);

  }

  render() {

    // Complete the return statement:

    return (

      <Provider store={store}>

        <Container />

      </Provider>

    );

  }

};

## Extract Local State into Redux

### Explanation:

You're almost done! Recall that you wrote all the Redux code so that Redux could control the state management of your React messages app. Now that Redux is connected, you need to extract the state management out of the Presentational component and into Redux. Currently, you have Redux connected, but you are handling the state locally within the Presentational component.

### Example:

// Redux:

const ADD = 'ADD';

const addMessage = (message) => {

  return {

    type: ADD,

    message: message

  }

};

const messageReducer = (state = [], action) => {

  switch (action.type) {

    case ADD:

      return [

        ...state,

        action.message

      ];

    default:

      return state;

  }

};

const store = Redux.createStore(messageReducer);

// React:

const Provider = ReactRedux.Provider;

const connect = ReactRedux.connect;

// Change code below this line

class Presentational extends React.Component {

  constructor(props) {

    super(props);

    this.state = {

      input: ''

    }

    this.handleChange = this.handleChange.bind(this);

    this.submitMessage = this.submitMessage.bind(this);

  }

  handleChange(event) {

    this.setState({

      input: event.target.value

    });

  }

  submitMessage() {

    this.props.submitNewMessage(this.state.input)

    this.setState((props) => ({

      input: ''

    }));

  }

  render() {

    return (

      <div>

        <h2>Type in a new Message:</h2>

        <input

          value={this.state.input}

          onChange={this.handleChange}/><br/>

        <button onClick={this.submitMessage}>Submit</button>

        <ul>

          {this.props.messages.map( (message, idx) => {

              return (

                 <li key={idx}>{message}</li>

              )

            })

          }

        </ul>

      </div>

    );

  }

};

// Change code above this line

const mapStateToProps = (state) => {

  return {messages: state}

};

const mapDispatchToProps = (dispatch) => {

  return {

    submitNewMessage: (message) => {

      dispatch(addMessage(message))

    }

  }

};

const Container = connect(mapStateToProps, mapDispatchToProps)(Presentational);

class AppWrapper extends React.Component {

  render() {

    return (

      <Provider store={store}>

        <Container/>

      </Provider>

    );

  }

};

## Moving Forward From Here

### Explanation:

Congratulations! You finished the lessons on React and Redux. There's one last item worth pointing out before you move on. Typically, you won't write React apps in a code editor like this. This challenge gives you a glimpse of what the syntax looks like if you're working with npm and a file system on your own machine. The code should look similar, except for the use of import statements (these pull in all of the dependencies that have been provided for you in the challenges). The "Managing Packages with npm" section covers npm in more detail.

Finally, writing React and Redux code generally requires some configuration. This can get complicated quickly. If you are interested in experimenting on your own machine, the [Create React App](https://www.freecodecamp.org/news/install-react-with-create-react-app/) comes configured and ready to go.

Alternatively, you can enable Babel as a JavaScript Preprocessor in CodePen, add React and ReactDOM as external JavaScript resources, and work there as well.

### Example:

/\*

import React from 'react'

import ReactDOM from 'react-dom'

import { Provider, connect } from 'react-redux'

import { createStore, combineReducers, applyMiddleware } from 'redux'

import thunk from 'redux-thunk'

import rootReducer from './redux/reducers'

import App from './components/App'

const store = createStore(

  rootReducer,

  applyMiddleware(thunk)

);

ReactDOM.render(

  <Provider store={store}>

    <App/>

  </Provider>,

  document.getElementById('root')

);

\*/

// Only change code below this line