Introduction to Redux

- What is Flux?
- What is Redux?
- Basic Redux Principles

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

What is Redux?

- Redux is a popular interpretation of the **Flux** architecture that aims to make state management easier for JavaScript applications
- Redux works with React, but is not specific to React
- Redux supports JavaScript applications that run in the browser, on the server as well as native mobile apps



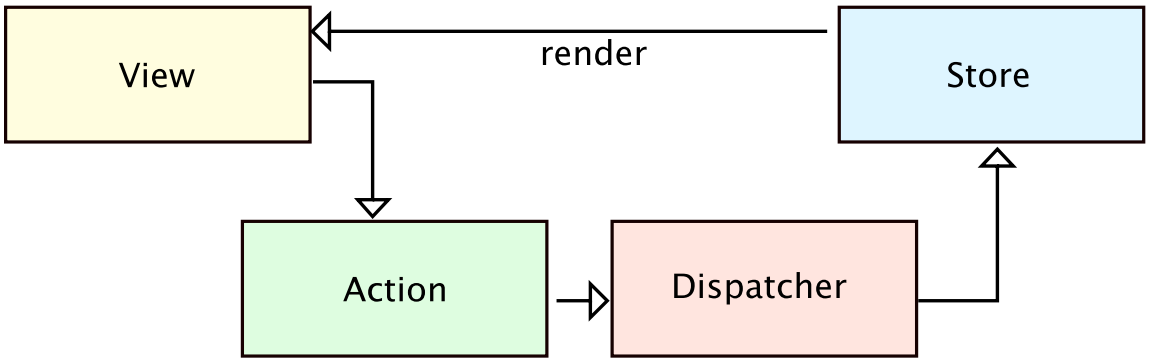
Redux's home page is at: https://redux.js.org.

Redux was created by Dan Abramov and Andrew Clark in 2015.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

What is Flux?

- Flux was created at Facebook as a **one-way data flow** architecture to make it easier to debug and understand highly stateful React applications



Flux is a response to how state is managed in other popular JavaScript libraries, especially Angular, that use a concept referred to as "two-way data binding". Facebook ran into issues with large scale applications in which converting to a one-way flow helped diagnose and debug issues.

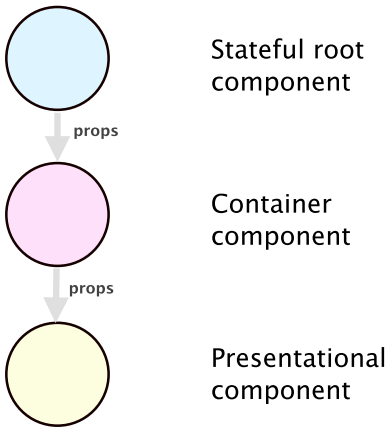
In addition to providing the architecture, Facebook also provides an implementation, which is also known as flux.

Redux is not the only third-party Flux implementation; others include Reflux, Alt and Fluxible.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Why Use Redux?

- Some applications need to pass **props** through a deep component tree, often passing props through components that don't use them - Redux lets you connect descendents directly to the data they need
- Redux makes state changes predictable since all mutations flow through a well known path
- Redux provides powerful debugging and visualization browser tools to make it easier to debug and reason about state changes, including undo/redo



Redux also can make writing React apps that use server-side rendering easier.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Why NOT Use Redux?

- Redux requires additional application complexity which is likely overkill for apps with minimal state requirements
- Redux has a non-trivial learning curve

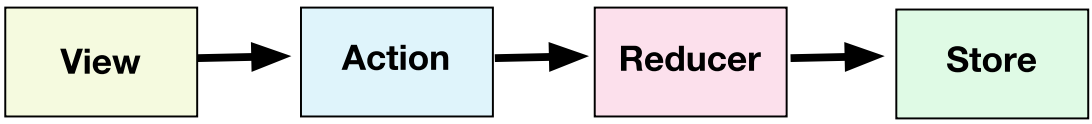
For an interesting take on "You Might Not Need Redux", see this blog post from Redux's creator:

https://medium.com/@dan_abramov/you-might-not-need-redux-be46360cf367

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Basic Redux Architecture

- Redux innovates on the Flux architecture:
  - Uses a single **store** to contain all application state
  - Uses **action** objects to encapsulate state changes
  - Uses **reducer functions** to apply actions to the store



Redux is not a completely faithful implmentation of the Flux architecture. In particular, it drops the centralized dispatcher component in favor of "reducers" and only allows a single state store.

Views display data to the user. If you're using Redux with React, you write views as React components.

Actions are simple JavaScript objects that encapsulate a state change.

Reducers are stateless functions that accept the current store state and an action - reducers return a new state object with specified changes.

The store is provided by Redux - your app will create one and then hook it up to the views and reducers.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Optional Redux Elements

- Redux allows for optional elements:
  - **Action creator** factory functions for centralizing action creation
  - **Selector** functions that centralize querying state from the store
  - Redux supports **middleware**, which provide extension points to the state-change flow, especially for cross-cutting concerns such as logging or Ajax request handling

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Redux Three Principles

- **Single Source of Truth** The state of your whole application is stored in an object tree within a single store
- **State is read-only** The only way to change the state is to emit an action, an object describing what happened
- **Changes are made with pure functions** To specify how the state tree is transformed by actions, you write pure reducers

See: https://redux.js.org/introduction/threeprinciples

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Introduction to Redux Stores

- A Redux **store** holds all of the state for a Redux application - it acts as the **single source of truth**
- Stores are read-only - the only way to change the store is via reducer functions that use actions to create updated copies of the store's state
- Redux provides the **createStore** API function that accepts a reducer function and returns a store reference

```
1    // stubbed out reducer accepts default state
2    // and action object
3    function myReducer(state='Hello', action) {}
4
5    const store = createStore(myReducer);
6
7    console.log("Initial state: " + store.getState());
```

You can read the API docs for createStore() at https://redux.js.org/api-reference/createstore.

Stores provide the getState() API to retrieve a reference to the current application state as well dispatch() to trigger a state change and subscribe() so that views can be notified of updates. You can read the full store API at: https://redux.js.org/api-reference/store.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Introduction to Actions

- Redux **actions** are simple objects that, by convention, contain:
    - A property named **type** whose values are upper-cased strings that indicate what state change is needed
    - Other properties as needed to contain the new state information

```
1    const myAction = {
2       type: "APPEND",
3       payload: {}
4    };
```

Redux apps can optionally consolidate the creation of actions into a centralized factory referred to as an "action creator".

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Introduction to Reducers

- Redux **reducers** are stateless functions that accept current store state and a Redux **action** object - reducers return a **copy** of the state, updated as specified by the action
- Reducer functions **must** be stateless with no side effects so that Redux can keep a log of updates and provide undo/redo
- It's common to write multiple reducers, each of which works on a slice of the total state in the store

```
1    function myReducer(state = 'Hello', action) {
2      if (action.type === 'APPEND') {
3        return state + ', world';
4      }
5      return state;
6    }
```

If you code multiple reducers, you combine them into a top-level reducer that delegates to the individual reducers - that's because the createStore() API accepts a reference to a single reducer. In that scenario, each reducer examines the action type and decide whether or not to do something or not.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The Most Basic Redux Hello, World

- This code uses Redux "standalone" without integrating with React in any way but shows Redux's basic flow

SimpleRedux.js

Note that in real-world React-Redux apps, you typically don't need to call subscribe() - the React-Redux integration library will do it behind the scenes.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

```
1    import React, { Component } from 'react';
2    import { createStore } from 'redux';
3    import logo from './logo.svg';
4    import './App.css';
5
6    class App extends Component {
7       constructor(props) {
8          super(props);
9
10         const store = createStore(this.myReducer);
11
12         console.log("Initial state: " + store.getState());
13
14         store.subscribe(() => {
15           console.log('Current state: ', store.getState());
16         });
17
18         store.dispatch({ type: 'APPEND', payload: {} });
19      }
20
21       myReducer(state = 'Hello', action) {
22         if (action.type === 'APPEND') {
23           return state + ', world';
24         }
25         return state;
26       }
27
28
29     render() {
30       return (
31         <div className="App">
32           <header className="App-header">
33             <img src={logo} className="App-logo" alt="logo" />
34             <h1 className="App-title">Welcome to React</h1>
35           </header>
36           <p className="App-intro">
37             To get started, edit <code>src/App.js</code> and save to reload.
38           </p>
39         </div>
40       );
41     }
42   }
43
44   export default App;
```

Redux and React

- Since Redux is not tied to React, you use the **react-redux** package to hook them together
- react-redux provides:
    - **Provider** that wraps a top-level React component and makes a Redux store available to nested React components
    - **connect** function that hooks up React components with the data they need from the store

If you are using npm, you can install react-redux:

```
npm install react-redux
```

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Presentational and Container Components with Redux

- React **presentational components** are stateless, accept only **props** and are not aware of Redux
- Without Redux, **container components** are often stateful and compose presentational components
- With Redux, container components can delegate state management to Redux, but still compose other components

If you use Redux, you can write most of your components as stateless, even containers. However, you don't need to use Redux for ALL components that need state. Consider a "controlled-component" user-input form component that maintains local state to keep track of the UI. It's simpler to write such components using normal React techniques and not introduce the extra complexity of Redux. In other words, most developers only use Redux to store so-called global app state.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The Redux Provider Component

- **Provider** wraps your top-level React component and works behind the scenes to allow connection between descendant React components and the store

```
1    import React from 'react';
2    import ReactDOM from 'react-dom';
3    import {createStore} from 'redux';
4    import {Provider} from 'react-redux';
5    import myReducer from './reducers/index';
6
7    const store = createStore(myReducer);
8
9    ReactDOM.render(
10       <Provider store={store}>
11           <App/>
12       </Provider>,
13       document.getElementById('root'));
```

Documentation for Provider is at:
https://github.com/reduxjs/react-redux/blob/master/docs/api.md#provider-store.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The connect() Function

- The **connect** function wraps a React component into a new *connected* component that receives its **props** from the Redux store
- You can then compose the connected component in your React app's component tree as normal

```
1    // MyConnectedComponent.js
2    import { connect } from 'react-redux';
3
4    const MyConnectedComponent = (props) => {
5    . . . .
6    }
7
8    export default
9      connect(mapStateToProps)(MyConnectedComponent);
```

You can use connect() on both presentational and container components. The idea is to connect() wherever in the component tree where you want to get data from the store and map it to a component's props. That means you can connect() a container and the container can pass props to its descendants in the normal React fashion. Or, if you have a hierarchy in non-Redux React in which you find yourself passing props through components that don't use them, just to get the data to descendants, you can use connect() to directly hook up the descendants to the store, avoiding the "passing unused props" issue.

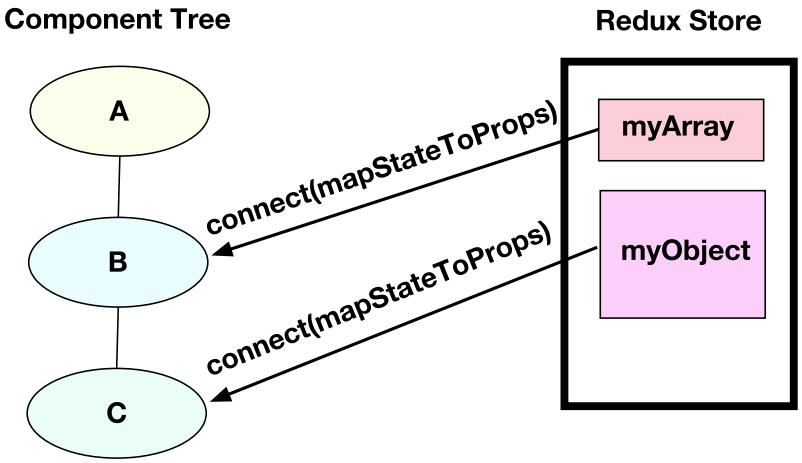
The docs for connect() are at: https://github.com/reduxjs/react-redux/blob/master/docs/api.md#provider-store Note that we pass a reference to a function, traditionally named mapStateToProps(), that specifies what slice of the entire store data that Redux should pass to the connected component. We will show an example of mapStateToProps() shortly.

The connect() function returns a so-called "higher order React component", which the docs describe as a function that accepts a reference to a React component, returning a new, "wrapped" component. A parent could then import and compose the connected component in the normal fashion.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

The connect() Function, cont'd

- You can connect as many components as you wish, even down in the component tree, to avoid passing props through components that don't actually use them



connect() optionally subscribes the wrapped component to changes in the store, so the component automatically re-renders when its slice of the store updates.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Mapping Store State to props

- When you call connect(), you can pass a function that maps data from the store to the connected component's **props**

```
// data in store
const data = {
   myArray: [123,456,947],
   myObject: {firstVal: "abc", secondVal: "def"}
}


// componentB.js

. . .
const mapStateToProps = (state) => {
  {theArray: state.myArray}
};
```

By providing this function, you can assign the portion of the store data that the component needs. In this case, ComponentB will get the "myArray" from the store passed to in props as "theArray".

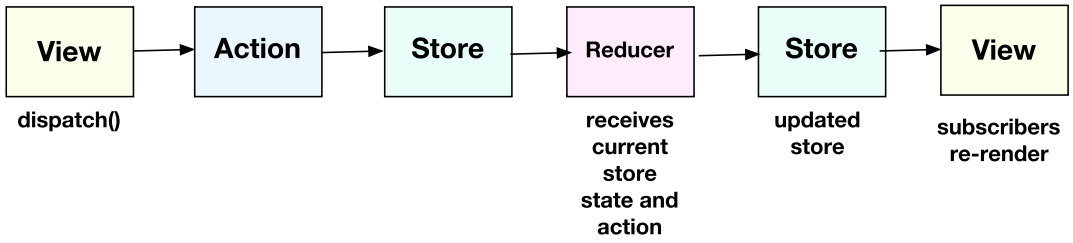
You can name the mapping function anything you want, but the conventional name is mapStateToProps().

If you pass such a mapping function to connect(), it will subscribe the connected component to updates and call the mapping function whenever the store is updated to remap the updated state.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Updating: Dispatching an Action

- It's common for view components to respond to user interaction, triggering an update to the application's state (the store)
- Redux passes **props** to connected components - the props include a **dispatch** function reference in addition to normal props
- You can use the **dispatch** function to invoke an **action** on the store, updating the store and re-rendering affected components



The dispatch() function is a method on the store. You can read the API docs at: https://redux.js.org/api/store#dispatch

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Dispatching an Action, cont'd

```
1    // MyConnectedComponent.js
2    <button
3      onClick={() => {updateStore(props.dispatch, myData)}}>
4        Click Me!
5    </button>
6
7    const updateStore = (dispatch, myData) => {
8       dispatch({type: "MY_ACTION",
9          payload: {myData: myData}});
10
11   }
```

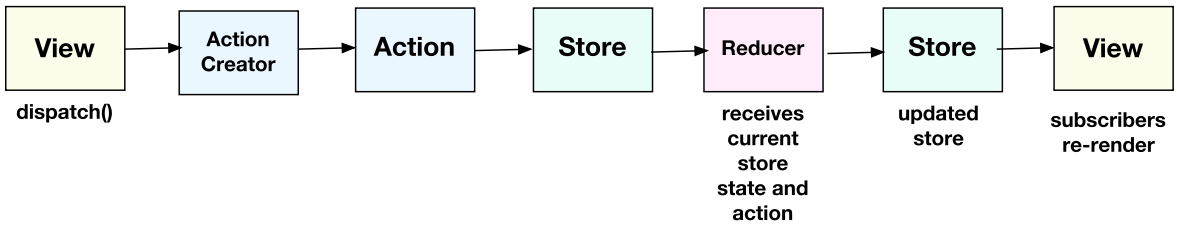

Here we show a fragment of a connected component, i.e. a view. Note it has an event handler so that when the user clicks the button, we end up calling dispatch(), which starts the process of updating the store.

Note the syntax for the onClick handler – writing inline arrow functions in this fashion is the "React way".

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Using an Action Creator Function

- An **action creator** function creates an action object and returns it
- Using action creators is optional, but lets you centralize creating actions to make maintenance easier
- For simple cases, you can avoid writing an action creator by coding the view to directly create and dispatch the action as shown previously



Essentially, action-creator functions add boilerplate to your application and a level of indirection. For simple scenarios, you might not need that extra code, but for more complex applications, it makes maintenance easier - if you decide to change the format of an action object, for example, you don't need to hunt around in your code to find where it's created.

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Using an Action Creator Function, cont'd

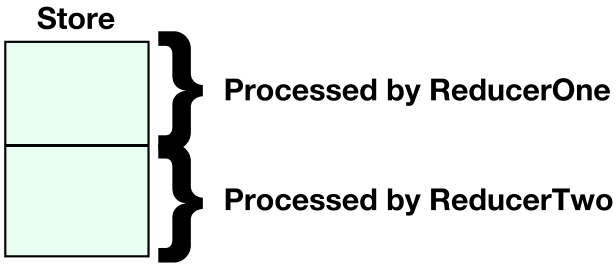
```
1    // MyConnectedComponent.js
2    import {myActionCreator} from '../actions';
3    . . . .
4    const updateStore = (dispatch, myData) => {
5       dispatch(myActionCreator(myData);
6    }
```

```
1    // actions/index.js
2    export function myActionCreator(myData) {
3        return {
4            type: "MY_ACTION",
5            payload: {myData: myData}
6        }
7    }
```


Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Combining Reducers

- In more complex Redux applications, it's a good idea to refactor a single, complex reducer into individual reducer functions, each of which work on a slice of total store state
- Redux provides the **combineReducers** function to which you pass references to individual reducers - it returns a root reducer that delegates to child reducers
- You can then pass the root reducer to the **createStore()** API



You can read about the combineReducer() API at: https://redux.js.org/api/combinereducers.

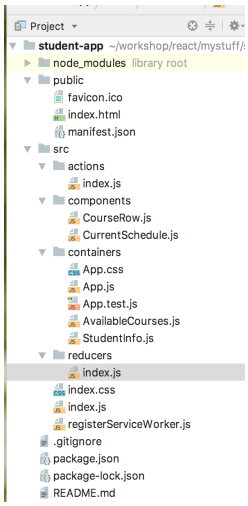
Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

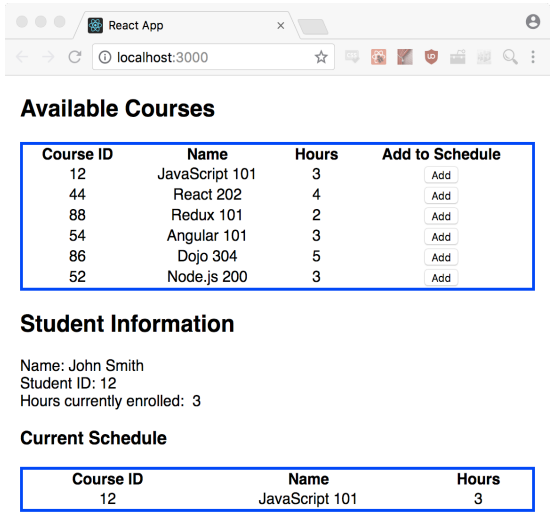
Combining Reducers, cont'd

```
1    // reducers/index.js
2
3    import {combineReducers} from "redux";
4
5    const reducerOne = (state = initialData, action) => {
6      if(action.type === "ACTION_ONE") {....}
7    }
8
9    const reducerTwo = (state = initialData, action) => {
10     if(action.type === "ACTION_TWO") {....}
11   }
12
13   const rootReducer = combineReducers({
14      reducerOne, reducerTwo});
15
16   export default rootReducer;
```


Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Complete React-Redux Example





index.js

App.js

CurrentSchedule.js

reducers-index.js

AvailableCourses.js

StudentInfo.js

actions-index.js

CourseRow.js

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

```
1    import React from 'react';
2    import ReactDOM from 'react-dom';
3    import {createStore} from 'redux';
4    import {Provider} from 'react-redux';
5    import rootReducer from './reducers/index';
6
7    import './index.css';
8    import App from './containers/App';
9    import registerServiceWorker from './registerServiceWorker';
10
11   const store = createStore(rootReducer);
12
13   // wrap root component in <Provider> so it and descendants can
14   // access Redux store
15   ReactDOM.render(
16       <Provider store={store}>
17           <App/>
18       </Provider>,
19       document.getElementById('root'));
20
21   registerServiceWorker();
```

```
const data = {
    courseList: [
        {courseID: 12, name: "JavaScript 101", hours: 3},
        {courseID: 44, name: "React 202", hours: 4},
        {courseID: 88, name: "Redux 101", hours: 2},
        {courseID: 54, name: "Angular 101", hours: 3},
        {courseID: 86, name: "Dojo 304", hours: 5},
        {courseID: 52, name: "Node.js 200", hours: 3}
    ],
    currentStudent: {
        studentID: 12,
        fullName: "John Smith",
        currentSchedule: []
    }
};

const rootReducer = (state = data, action) => {
    console.log("rootReducer, state: %o, action: %o ", state, action);

    if (action.type === "ADD_TO_SCHEDULE") {

        // first ensure that specified course not already on schedule
        for (let i = 0;
             i < state.currentStudent.currentSchedule.length;
             i++) {
            if (state.currentStudent.currentSchedule[i].courseID ===
                action.payload.courseID) {
                    return state;
            }
        }

        // need to find the course in the courseList, then
        // clone the currentSchedule and append new course
        // then return cloned total state including the
        // updated currentSchedule array

        // find the course
        let course = {};

        for (let i = 0; i < state.courseList.length; i++) {
            if (action.payload.courseID === state.courseList[i].courseID) {
                course = state.courseList[i];
                break;
            }
        }
```

```
46
47          // first clone the current state, doing a deep copy
48          let newState = JSON.parse(JSON.stringify(state));
49
50          // add new course to schedule
51          newState.currentStudent.currentSchedule.push(course);
52
53          return newState;
54      }
55
56      return state;
57  }
58
59  export default rootReducer;
```

```
1
2    export function addCourseToSchedule(courseID) {
3        return {
4            type: "ADD_TO_SCHEDULE",
5            payload: {courseID: courseID}
6        }
7    }
```

```
1    import React, {Component} from 'react';
2    import './App.css';
3
4    import AvailableCourses from "./AvailableCourses";
5    import StudentInfo from "./StudentInfo";
6
7    class App extends Component {
8        render() {
9            return (
10               <div className="App">
11                   <AvailableCourses/>
12                   <StudentInfo/>
13               </div>
14           );
15       }
16   }
17
18   export default App;
```

```
1    import React from 'react';
2    import {connect} from 'react-redux';
3    import {addCourseToSchedule} from '../actions';
4
5    // this component uses Redux to connect to global app
6    // state and also maintains local state so it can track
7    // UI changes so it can disable the "add" button for
8    // adding a course to the current schedule
9    class AvailableCourses extends React.Component {
10     constructor(props) {
11       super(props);
12
13       // keep track of which courses the user's added
14       // in this array
15       this.state = {
16         addedCourses: []
17       }
18
19       this.handleAdd = this.handleAdd.bind(this);
20     }
21
22     // append newly added course to state array so we can
23     // disable the UI "Add" button
24     handleAdd(courseID) {
25       this.setState({
26         addedCourses: [...this.state.addedCourses, courseID]
27       });
28     }
29
30     render() {
31       return (<section>
32           <h1>Available Courses</h1>
33           <table>
34             <thead>
35             <tr>
36               <th>Course ID</th>
37               <th>Name</th>
38               <th>Hours</th>
39               <th>Add to Schedule</th>
40             </tr>
41             </thead>
42             <tbody>
43             {
44               // iterate courseList provided by Redux
45               this.props.courseList.map((course) => {
```

```
46                  return (
47                    <tr key={course.courseID}>
48                      <td>{course.courseID}</td>
49                      <td>{course.name}</td>
50                      <td>{course.hours}</td>
51                      <td>
52                        <button
53                          disabled={
54                            this.state.addedCourses.indexOf(course.courseID) !== -1}
55                          onClick=
56                            {() => {
57                              this.handleAdd(course.courseID);
58                              updateStore(this.props.dispatch, course.courseID);
59                            }}>
60                          Add
61                        </button>
62                      </td>
63                    </tr>
64                  )
65                }
66              )
67            }
68            </tbody>
69          </table>
70        </section>
71      );
72    }
73  }
74
75  // dispatch an action on the store to update current schedule
76  const updateStore = (dispatch, courseID) => {
77
78    // dispatch({type: "ADD_TO_SCHEDULE",
79    //     payload: {courseID: courseID}});
80
81    // invoke action-creator function
82    dispatch(addCourseToSchedule(courseID));
83  }
84
85  // AvailableCourses component needs courseList provided by Redux
86  // note how we use parenthesis in the arrow function - that's
87  // special syntax to return object-literal
88  const mapStateToProps = (state) => (
89    {courseList: state.courseList}
90  );
```

```
91
92   // need to connect() to get data from Redux store
93   export default connect(mapStateToProps)(AvailableCourses);
```

```
1    import React from 'react';
2
3    const CourseRow = ({courseID, name, hours}) => {
4        return (
5            <tr key={courseID}>
6                <td>{courseID}</td>
7                <td>{name}</td>
8                <td>{hours}</td>
9            </tr>
10       );
11   };
12
13   export default CourseRow;
```

```
1    import React from 'react';
2
3    const CurrentSchedule = ({currentSchedule}) => {
4        return(
5            <section>
6                <h1>Current Schedule</h1>
7                <table>
8                    <thead>
9                    <tr>
10                       <th>Course ID</th>
11                       <th>Name</th>
12                       <th>Hours</th>
13                   </tr>
14                   </thead>
15                   <tbody>
16                   {
17                       currentSchedule.map((course) => {
18                               return (
19                                   <tr key={course.courseID}>
20                                       <td>{course.courseID}</td>
21                                       <td>{course.name}</td>
22                                       <td>{course.hours}</td>
23                                   </tr>
24                               )
25                           }
26                       )
27                   }
28                   </tbody>
29               </table>
30
31           </section>
32       );
33   };
34
35   export default CurrentSchedule;
```

```
1    import React from 'react';
2    import CurrentSchedule from "../components/CurrentSchedule"
3    import {connect} from "react-redux";
4
5    const StudentInfo = ({currentStudent}) =>
6        <section>
7            <h1>Student Information</h1>
8            <div>Name: {currentStudent.fullName}</div>
9            <div>Student ID: {currentStudent.studentID}</div>
10           <div>
11               Hours currently enrolled:  
12               {calcHours(currentStudent.currentSchedule)}
13           </div>
14           <CurrentSchedule
15               currentSchedule={currentStudent.currentSchedule}/>
16       </section>
17
18   const calcHours = (currentSchedule) => {
19       let hours = 0;
20
21       for(let i = 0; i < currentSchedule.length; i++) {
22           hours += currentSchedule[i].hours;
23       }
24
25       return hours;
26   }
27
28   // StudentInfo needs currentStudent from the store
29   const mapStateToProps = (state) => (
30       {currentStudent: state.currentStudent}
31   );
32
33   // need to connect() to get access to redux store
34   export default connect(mapStateToProps)(StudentInfo);
```

Chapter Summary

In this chapter, you learned:

- The basics of how Redux lets you separate application state from the app itself and how to connect the state each nested component needs
- How to architect and refactor Redux-based apps using reducers and action-creator functions

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum

Copyright © Descriptor Systems, 2001-2016. Course materials may not be reproduced in whole or in part without prior written consent of Joel Barnum