React Redux

Concept, Workflow & Cheatsheet

Prof. Dr. Ulrich Anders, Cologne Business School

Version 1.2.1 · September, 09 2017

React and Redux really are some impressive developments and are certainly influencing how Frontend design is carried out now and in the foreseeable future. The principal concept of React is much easier to grasp than that of Redux. React deals with components that are much more imaginable. Redux on the other hand introduces a workflow that is much less naturally conceivable especially as it uses some vocabulary in its API that is not always intuitive.[1]

It is stated in the Redux documentation, that React does not need Redux and that it should not be used if it is not needed, but I think the opposite is true. Redux significantly reduces the complexity of an app, so in my opinion it really should be rather used than not used.[2] I actually love to think that the name Redux is derived from REDUce compleXity.

The price one has to pay for the reduction of complexity is, however, to learn the logic and vocabulary of an extra library. In addition to the official documentation, there are some really excellent tutorials out there about React with Redux. To name a few:

1. Frontend – Build your first real world React.js application. Post by Max Stoiber.
2. Leveling Up with React: Redux. Post by Brad Westfall
3. Three Rules For Structuring (Redux) Applications. Series of posts by Jack Hsu.
4. Getting Started with Redux. Video series by Dan Abramov.
5. Building React Applications with Idiomatic Redux. Video series by Dan Abramov.
6. Learn Redux. Video series by Wes Bos.
7. Learn React and Redux. Video series by Catalin Luntraru.

Most tutorials approach the topic of Redux by building an app. While this is done the tutorials introduce the important concepts and workflow around Redux step by step. The general challenge with this approach is that two types of information are competing with each other: the conceptual overview and the details of the coding.

So, in order to supplement existing tutorials, this article describes the Redux conceptual overview and its workflow in a React Redux app. The description is starting with the dominant player in Redux Applications which is the store. Once the workflow is understood, it will probably be much easier to follow all of the above tutorials.

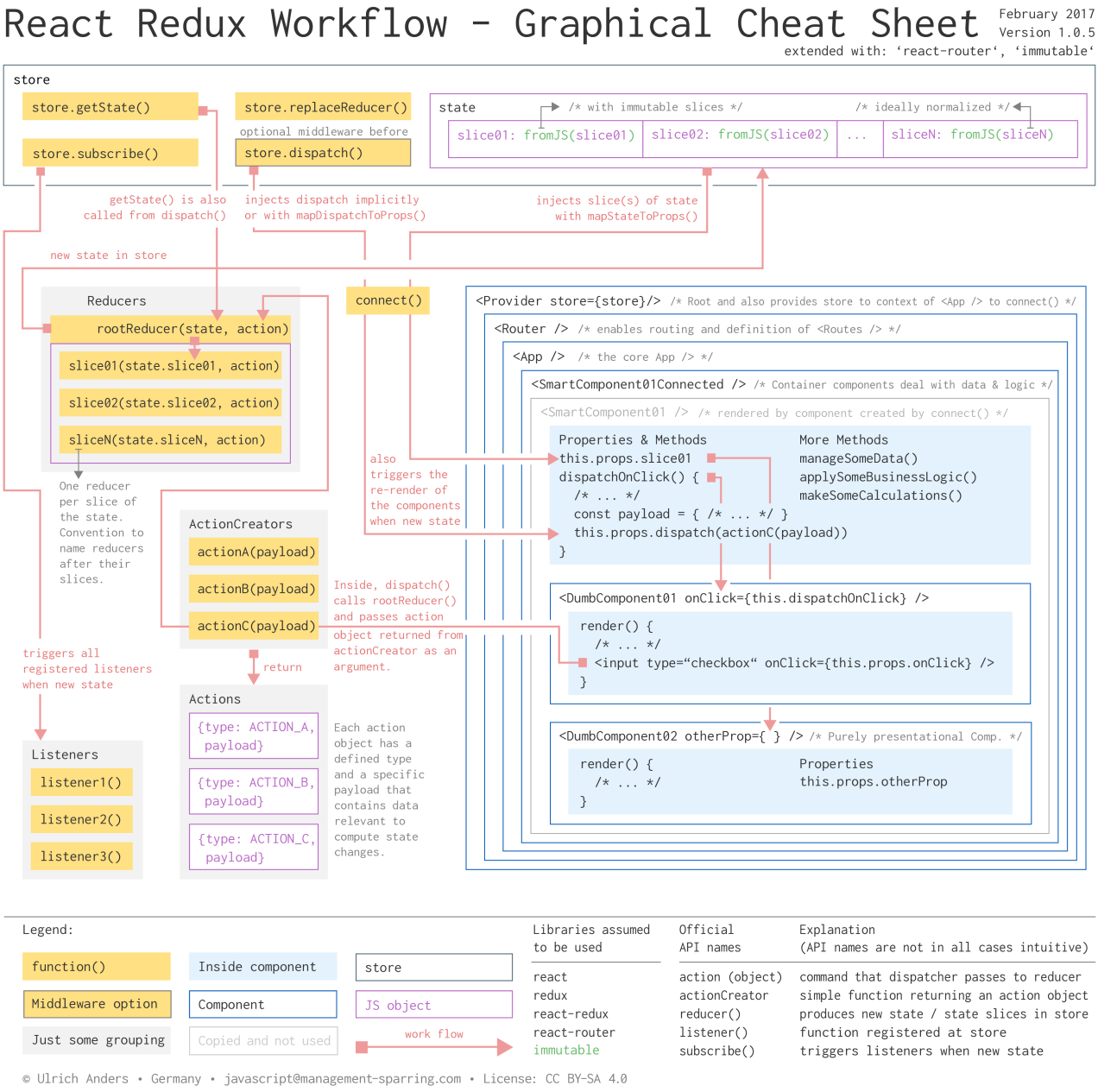
While going the full circle this article also points to some of the common external libraries and how they would come into play: 'immutable', 'normalizr', 'reselect', 'redux-thunk', 'redux-saga', 'redux-promise' and 'redux-persist'.

[1] Naming an API so that it is semantically intuitive is an art. At the time, the Redux API has been chosen to mainly stay close to the API of Flux which was already quite known by then.

[2] I do not mean to suggest to always use Redux over MobX or Flux. I mean to always use Redux as opposed to not using Redux or alternatives.

1 A Graphical Cheat Sheet

I'd like to start with a graphical cheat sheet explaining the workflow in a React Redux app. For those who already know React Redux it just serves as a reminder. For all others, once you have read this article you will understand the workflow pretty sure.



2 Store

1. The central idea of a Redux app is to separate the state of the app from the app itself.
2. The state of the app is stored in a store. I therefore distinguish between the state OF the app which is IN the store.
3. The store is THE dominante player in a React Redux app.

4. The state stored in the store is a simple JS object with slices. As the state thus hierarchically contains other structures it also called a state tree:

```
// this uses ES6 shorthand syntax
state = {
  slice01,
  slice02,
  /* ... */
  sliceN
}
```

5. Every slice can have its own data type: number, string, array and of course object. For the purpose of this tutorial I assume that all slices are of type object. For managing the state of your app you will find, that it is much easier to access parts of this objects if it is normalized. Read more about normalization in the library *normalizr*.

```
// bad:
"users": {
  {"id": "1", "name": "Adam"},
  {"id": "2", "name": "Eve"},
}

//good:
"users": {
  "1": { "id": "1", "name": "Adam" },
  "2": { "id": "2", "name": "Eve" }
},
```

6. Every slice should be immutable. That means it cannot be changed. If the state of the app changes, the old state object has to be updated into a new state object that contains everything from the current state object and the updated slice.

7. To help manage the immutability and also make the updates fast it is sensible to use for instance *immutable.js*. This library provides a function *fromJS()* that generates an immutable object from a JS object. I have assumed here that all slices are JS objects, which is most often the case in complex apps. But of course, other data structures are possible instead.

```
import { fromJS } from 'immutable'

state = {
  slice01: fromJS(slice01)
  slice02: fromJS(slice02),
  /* ... */
  sliceN: fromJS(SliceN)
}
```

8. There are some pros and cons using *immutable.js* which you can find in the Redux recipes for *immutable.js*. I agree with all of them but I differ with respect to putting the whole state into immutable.js. Instead, I recommend to use immutable.js only within a slice which gives you much more flexibility and preserves the logic of the state being split in slices, where each slice can be accessed and managed separately.

9. Every slice of the state is managed by its own function, that is responsible for updating exactly

this slice and copying all other slices into the new state object. This function is called a *reducer()*. The name *reducer* is semantically not very meaningful and has a purely technical origin.[3] If you struggle with the name translate it into *producer*, because its only purpose is to just produce a new state that is put in the store.

10. It is a popular convention (but not an obligation) to name a reducer function after the slice it manages. I suggest, that you really stick to this convention, because it is helpful in many respects when you go in. So a reducer for *state.slice01* would be called *slice01()*.

11. Since there usually is a reducer for each slice, we end up with a lot of individual slice reducers where each of them is only concerned with its slice. However, we need to take care of all the slices at the same time. The solution is to combine all slice reducers into one overall reducer that is often named *rootReducer()*. To achieve this there is a function called *combineReducers()* that takes an object containing the individual reducers:

```
import { combineReducers } from 'redux'

const rootReducer = function combineReducers({
  slice01,
  slice02,
  /* ... */
  sliceN
})
```

12. Note, since *combineReducers()* takes an object of reducers you can also build subsets of reducers with *combineReducers()* and then combine these subsets into the *rootReducer()*.

3 App & Components

1. A React Redux app contains components nested in components.

2. When a parent component has a child component the parent can pass down data to its child via key-value-pairs. These data become properties of the child component and are therefore accessible from within the child component via the property keyword *props*.

3. As the store is a component completely separate from the app, the store needs to become known to the app. This is achieved by passing the store from the Root component to the App component. Within the App component the store would now be accessible with *this.props.store*.

```
const Root = ( {store} ) => (
    <App store={store} />
)
```

4. The store could now be passed down from the App component to all its child components. But this is exactly not what we want, because not all components would need the store at all or all of it.

5. There are two types of components: smart (container) components and dumb (presentational) components. Smart components have state, dumb components don't. Of course, there are

[3] Quoting from the official documentation: "It's called a reducer because it's the type of function you would pass to Array.prototype.reduce(reducer, ?initialValue)."

exemptions to this rule. In the context of Redux you can interpret that smart components have outsourced the state they originally possessed to an external state manager which is the store.

6. Smart components are managing data, making calculations or handling events. They also embed dumb components and pass down data or functions needed by the dumb components.

```
class SmartComponent01 extends Component {
  manageSomeData () {
    /* ... */
  }
  makeSomeCalculations () {
    /* ... */
  }
  handleSomeEvent = (event) => {
    /* ... */
  }

  render() {
    return (
      <div>
        <DumbComponent01 data={/*...*/} />
        <DumbComponent02 func={/*...*/} />
      </div>
    )
  }
}
```

   Just as a side note, since this is the first time we are showing class methods: remember, that there are two types of notations for method declarations with a different effect to *autobinding* the methods to the *this* operator of the classes. Here, you would need to bind *manageSomeData()* and *makeSomeCalculations()* manually to the *this* operator, whereas *handleSomeEvent()* would have been autobound.

7. Since only smart container components manage data only they need to receive state data from the store. This works via a function called *connect()* which you need to import from 'react-redux'. The *connect()* function returns a higher order component, i.e. a component that expects a component as an argument. This argument is the smart component that you want to connect. The higher order component created by the *connect()* function renders the smart component passing the data from the store into its props.

```
import {connect} from 'react-redux'

/* ... */

const ConnectedSmartComponent01 = connect()(SmartComponent01)
export default ConnectedSmartComponent01
```

8. Even though your smart component is now connected, it does not yet receive any state data from the store. So you need to pass it the required state data. But where do you map it to? Well, remember components have properties and they are named *props*.

9. Thus, the mapping from a state slice to the properties of a smart component is done with a function that is conventionally named *mapStateToProps()*. Per connected component you'll

need to write such a function, that will return the state data of the slice(s) needed by the component. So, in the end you will have plenty of *mapStateToProps()* functions all with the same name, but a different content. The same name is not a problem, since none of the *mapStateToProps()* functions are ever exported. So, for the component to now actually receive the state data via the *connect()* function you need to pass *mapStateToProps()* to the *connect()* function. By doing this, the connected component is then subscribing to Redux state updates.

```
import {connect} from 'react-redux'

function mapStateToProps(state) {
  return {
    // component gets this.props.slice01
    slice01: state.slice01
  }
}

// export without a new name
export default connect(mapStateToProps)(SmartComponent01)
```

10. Whenever the state in the store is updated, all your *mapStateToProps()* mapping functions for connecting components will be called and the new state data mapped to the properties of the connected components.

11. If you only want to map a subset of your slice to certain properties in your component you can make a selection from the slice of your state tree. A function that makes a selection is called a *selector*. It is a convention to name selector functions with an initial *get...*. But remember each time the store gets an update all mapping functions are called independent of whether the state change is relevant for a component. That means that also all the selector functions are called. If a selector function now is computationally expensive, it might slow down your app. In this case you should optimize your selectors by help of 'reselect'.

```
function mapStateToProps(state) {
  return {
    // component gets this.props.selection1
    selection1: getSelection1(state.slice01),
    // component gets this.props.selection2
    selection2: getSelection2(state.slice01)
  }
}
```

12. Coming back to the *connect()* function. Even though you have imported the connect function properly from 'react-redux' you may wonder how the *connect()* function actually is able to access the store across the whole app even though the store is not explicitly passed down to all child components. The answer sits with a component called *Provider* that you must wrap around your App component. The sole purpose of the *Provider* is to add the store to the context of the App component, so that all child components can access it. *Context* is a special feature of React for such rare cases. With this said, the *connect()* function can now access the store methods. Thus, the Provider component replaces the root component that we introduced in the beginning:

```
import { Provider } from 'react-redux'
```

```
const Root = ( {store} ) => (
  <Provider store={store}>
    <App />
  </Provider>
)
```

4 Reducers & Actions

1. Now, say, the user clicks on a button or inputs some information that changes the state of your application. In this case you want to issue a command that the state of your store is to be updated. Well, for issuing such a command there is a function called *dispatch()*. This function is a method that belongs to the store, which means the store not only holds the state of the app it also operates as a dispatcher that dispatches commands.

2. But how then can you access *dispatch()* from within a component when it belongs to the store? The answer sits with the *connect()* function. Whenever you connect a component to the store, the *connect()* function takes the stores *dispatch()* method and implicitly injects it into the component by mapping it to its properties. So from within the connected component it is accessible like so:

   ```
   this.props.dispatch
   ```

3. If you do not want the *connect()* function to just inject the standard *dispatch()* function and map it to *this.props.dispatch* you can modify it according to your needs. This is achieved by help of a function, that is conventionally named *mapDispatchToProps()*. After writing it you need to hand it over to the *connect()* function as the second argument.

   ```
   function mapDispatchToProps(dispatch) {
     return {
       /* your own bindings for the dispatch() function */
     }
   }

   export default connect(mapStateToProps,
                          mapDispatchToProps)(SmartComponent01)
   ```

4. Your component now has a *dispatch()* function. But what is the *dispatch()* function dispatching? A dispatcher normally dispatches a command to someone. Think of 911, where the dispatcher commands a police officer to undertake something. Or in a shop, where the dispatcher commands a good to be send to a customer. In Redux you need different commands, but they all relate to updating the state in the store. Each command therefore needs to contain two kinds of information. A type to represent the command and the data relevant for the state change. This command is called an *action* in Redux. The name *action* may be confusing at first, since with this name one would rather expect a function than an object. However, with time you get used to it. The *action* is a plain JS object containing a type and ideally the minimal amount of *relevant* data required to perform the state change.

   ```
   const DO_SOMETHING = 'DO_SOMETHING'

   // an action object
   ```

```
{
  type: DO_SOMETHING,
  payload: relevantData
}
```

5. Here, I am explicitly saying *relevant* data. The *relevant* data are not necessarily the changing state data itself. The relevant data could also contain an ID, for instance, to find the corresponding state data in the state tree. It is a convention to specify the action types in capital letters. It is not a pre-requisite, but many people assign the relevant data to a key that is often named *payload* to have the same API for all action objects.

```
// relevant data
let payload =
  {
    id,
    something
  }
```

6. It is not so convenient to deal with action objects, so typically one uses a function, that returns the action object. This function has the name *actionCreator()*. The name is a little bit pompous, because the function is not really creating much, it is just returning an action object — that's it.

```
// this simple function is called an actionCreator
function do_something(payload) {
  return (
    {
      type: DO_SOMETHING,
      payload
    }
  )
}
```

7. If the *dispatch()* function is now dispatching a command only in form of an object, which function is then actually receiving this command and dealing with updating the store? We are coming back to our reducer functions that we introduced earlier. Remember, that the purpose of the reducer functions is to produce a new slice of the state, if there is a change within this slice. So this means the *dispatch()* function has to call the reducer function. And this is exactly what happens underneath in the Redux code. So within its function body the *dispatch()* function calls the *rootReducer()* function and passes over the command what to do in form of the *action* object as the second argument.

8. And the first argument? A *rootReducer()* obviously, needs two arguments in order to return a new state: *rootReducer = (state, action) => newState*. The first argument is the current state in the store and the second argument represents the changes to it. And where does the *rootReducer()* now get the current state from? Well, again from the *dispatch()* function. It first fetches the current state from the store with *store.getState()* method and then passes the current state to the *rootReducer()* as a first argument together with the *action* object as a second argument.

9. The *rootReducer()* takes the received action and hands it down to all its child reducers, however, not with the whole state but only with its corresponding slice of the state: *sliceReducer =*

(slice, action) => newSlice.[4] Each slice reducer compares the received *action.type* to the cases it has within its function body. If no match is found it returns the current state of the slice and nothing changes. If there is a match it computes the update of the according slice and returns it, so that an overall new state with an updated slice can be generated.

```
// reducer for state.slice01
// using setIn() from immutable.js
function slice01(slice01 = {}, action) {
  switch (action.type) {
    case DO_SOMETHING:
      return slice01.setIn(['somewhere', action.payload.id],
                           action.payload.something)
    default:
      return slice01
  }
}
```

10. In this way, the store receives a new state. When the store is updated with a new state a re-render of the components that have subscribed to the store is triggered.

11. Note, that action creators are most often used to create actions that lead to a change of the state in the store. But not all action creators are used to generate state changes that are stored. Some action objects are generated by action creators just to make temporary state changes to the app which are not stored.

12. An arbitrary number of functions can register at the store to be also triggered whenever the store got updated with a new state. The functions registered at the store to be triggered have been named listeners, probably because they are "listening" to updates of the state. But in fact, they are not actively listening themselves. Quite the opposite, they are not actively in control, they are just passively triggered. The triggering function is a method that belongs to the store named *store.subscribe()*. This name also takes a little bit getting used to as the store itself actually does not subscribe to anything. So, mentally you could think *trigger* when you use *subscribe()*.[5]

    ```
    store.subscribe(listener)
    ```

13. This completes the cycle.

5 Round up

1. Even though we started this article with the store, we have not spoken about how to create it. This was because we needed to introduce the concept of reducers first. Now, that we know what reducers are, we can set up the store with *createStore()*. At the minimum the store needs one argument which is the *rootReducer()*. Why? Remember, that later on the *store.dispatch()* function is calling the *rootReducer()*, so it needs to know it. As an optional second argument you can pre-populate the store with an initial state:

[4] Note, other than the *Redux Documentation* reads *(previousState, action) => newState* a reducer does not receive the whole state by default but only its slice of the state.

[5] Don't confuse a component actually *subscribing* to a state change by help of the *connect()* function with a function that can be registered with the *store.subscribe()* method in order to be triggered whenever there is a new state.

```
import { createStore } from 'redux'

let store = createStore(rootReducer, initialState)
```

2. If you want to save the store to local storage or re-hydrate the store when you restart your app it may be well worth to have a look at 'redux-persist'. This library also helps you with immutable transformations and with encrypting your data in the local storage.

3. Sometimes you want to include additional functionality whenever an action command is issued by the *dispatch()* function. This could for instance be a logger function or a timeout scheduler. For this purpose Redux offers an interface to so called *middleware*. The middleware is executed after the *dispatch()* and before the *rootReducer()* function. That means, each time when the *dispatch()* function is issuing an action command, the middleware is performed before the *rootReducer()* gets to work. To set this up, there is a third optional argument in *createStore()* called *enhancer*. This argument expects a function that contains the individual middleware functions. Redux provides for such a function called *applyMiddleware()*. You can now pass your enhancements as arguments to *applyMiddleware()* and they are registered so that they are executed each time the *dispatch()* function is called. Obviously, every middleware function has to comply to the middleware function specification, since the output of the first middleware function is becoming the input for the next. The sequence in which they are executed is from left to right. The last function that is executed is actually the *dispatch()* function itself.

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

let store = createStore(
  rootReducer,
  applyMiddleware(logger, timeoutScheduler)
)
```

4. Many extensions to Redux exploit this middleware functionality. If you want to stack further middleware on top of e.g. the *applyMiddleware()* you can use the *compose()* utility function.

```
let store = createStore(
  rootReducer,
  compose(
    applyMiddleware(logger, timeoutScheduler),
    middlewareFromAnExtension()
  )
)
```

5. React and Redux are synchronous by nature, that means they go about their things sequentially. But what happens, if the *dispatch()* is issuing an action command to the root reducer, but the relevant data are not immediately available and you need to wait for them to arrive from an external API? Or if you want to *dispatch()* an action command which data are based on a promise? For these cases Redux has additional libraries such as 'redux-thunk', 'redux-promise' or 'redux-saga'. They all come as middleware for the *dispatch()* function and help you deal with such cases.

6. As React apps are rendered on the client side, there are initially no URLs to get around in the app. Nevertheless it makes absolute sense to have URLs and to link to certain references in your app. This is achieved with a package called 'react-router'. I am referring to $\geq$v4 of 'react-router', since quite some API updates have taken place in this version compared to previous versions. Basically 'react-router' offers a Router component, that comes along with a

lot of features to manage the routing in your app. If you want to have routing functionality in your app, it is conceptually easiest to just wrap the Router component around you app.

```
import { Provider } from 'react-redux'
import { BrowserRouter as Router } from 'react-router-dom'

const Root = ( {store} ) => (
  <Provider store={store}>
    <Router>
      <App />
    </Router>
  </Provider>
)
```

7. In your App component you can then use linking and routing by help of NavLink and Route components:

```
import { Provider } from 'react-redux'
import { NavLink, Route } from 'react-router-dom'

const App = () => (
  <div>
    <NavLink to="/">Home</NavLink>
    <NavLink to="/about">About</NavLink>
    <NavLink to="/users">Users</NavLink>

    <Route exact path="/" component={Home} />
    <Route path="/about" component={About} />
    <Route path="/users" component={Users} />
  </div>
)
```

5.1 Wrapping up

1. With all this said you can now perfectly understand the workflow cycle of a React Redux app. We started the explanation with the store, but let's see what happens, when we actually start in the app. In the app a user activity generates an event. The event handler calls the *dispatch()* function that is sending the current state and an action (object) to the *rootReducer()*. The action object contains the relevant data for the requested change of state slice. The *rootReducer()* will interpret the *action.type*, process the data and generate a new state. After the store has received the new state, it triggers the re-render of the React Redux app. It also triggers the execution of all listener functions that are registered with the *subscribe()* method to the store. Furthermore, all components that are subscribed with *connect(mapStateToProps)* to the store now receive the new state data as defined in *mapStateToProps()*.
2. I hope this walk-through has added a little bit to understanding Redux and motivates you to now use it.