# Redux and React: An Introduction

## **What is redux?**

Redux is a [flux](https://facebook.github.io/flux/docs/in-depth-overview.html) based state container for handling javascript application state. It is a popular choice for storing application state mainly due to its three defining principles:

* A single object tree stores all of your application state
* State is readonly and changes are triggered by actions
* The state can only be manipulated by pure functions that are triggered by your actions

You can read a bit more about redux [here](https://redux.js.org/docs/introduction/).

## **Actions**

An action is simply an object that describes a change you want to make to your state. These are somewhat similar to event objects.

A standard pattern for actions is the following structure:

**const** action **=** {

type: 'ACTION\_TYPE',

payload: 'Some data'

};

This is the structure we’ll use to describe all of our actions in this post as it keeps our actions very consistent.

The type is similar to an event type and is required for all actions, and the payload is the data that will be used to transform our state.

Not all actions need a payload though, as some actions like incrementing a number do not require any additional data e.g.

**const** increment **=** {

type: 'INCREMENT'

};

## **Action Creators**

Action creators are simply a function that allow us to abstract away the creation of actions, allowing us to easily dispatch an action without having to define all of its properties.

You may often hear action creators being referring to as actions, but for the purposes of this post I’ll refer to them as separate entities.

Here’s an example of a simple action creator:

**export** **const** ADD\_NUMBER **=** 'ADD\_NUMBER';

**export** **const** addNumber **=** (number) **=>** ({

type: ADD\_NUMBER,

payload: number

});

Now we can use this later to quickly create an action with some additional data attached to it e.g.

**const** action **=** addNumber(7);

## **Reducers**

A reducer is the pure function that we will use to transform our store state. Reducers are triggered whenever an action is dispatched and receive both the current state of that reducer (which will be undefined to begin with) and the action that was dispatched.

Here’s a simple example of a reducer that keeps track of a number and handles the “add number” action that we defined above.

**import** { ADD\_NUMBER } **from** './our-actions-file';

**export** **const** count **=** (state **=** 0, action) **=>** {

**switch** (action.type) {

**case** ADD\_NUMBER:

**return** state **+** action.payload;

**default**:

**return** state;

}

};

There are several things are important to understand when defining a reducer:

* Our state will be undefined to begin with, so we’ll want to give this a sensible default value (0 in this case)
* Our reducer cannot return an undefined value
* Our reducer will be triggered by any action that is dispatched, so we should return the existing state if the action is not relevant to this reducer (that’s what the default case is for).

## **Creating A Store**

Now that you understand the basics of actions and reducers we can actually put them to use and create a store.

In this case we are going to create a simple store that only contains our single “count” reducer.

**import** { createStore } **from** 'redux';

**import** { count } **from** './our-reducers-file';

**export** **const** store **=** createStore(count);

With this example our “count” reducer will make up the entirety of our store state, so calling the method store.getState() will simply return a number. Let’s talk a bit about some of the available store methods…

## **Store Methods**

We wont actually need to call any of these methods ourselves, (and I’d actually avoid this at all costs), as the tools we’ll cover shortly will handle this for us, but for the purposes of describing how the store composes our state, and how actions are dispatched it’s important to cover briefly.

### **getState**

store.getState() is pretty self explanatory - it simply returns the current state of the store.

### **dispatch**

store.dispatch() is the method that is used to dispatch an action and subsequently trigger our reducers.

If we were to manually dispatch our “add number” action we would do so in the following way:

store.dispatch(addNumber(7));

This would cause our “count” reducer to then be called with the current store state and our “add number” action.

Note that we are not passing the action creator itself to the dispatch function, but instead the action that is returned by it.

## **Combining Reducers**

For most applications we are going to want to store more than a single number, which we can then access from an object tree. In order to save us a lot of hassle handling all of the store state in a single reducer we can use a function provided by redux to combine our reducers into an object tree.

**import** { combineReducers, createStore } **from** 'redux';

**import** { count, someOtherReducer } **from** './our-reducers-file';

**export** **const** store **=** createStore(combineReducers({

count,

someOtherReducer

}));

What this is actually doing behind the scenes is creating another function that calls all our our reducers with the state that is relevant to them. It’s basically like a magical parent reducer.

If we were to over simplify how this works it’d look something like the following:

**const** combineReducers **=** (reducers) **=>** {

**return** (state **=** {}, action) **=>** {

**const** newState **=** {};

**for** (**let** key **in** reducers) {

**const** reducer **=** reducers[key];

newState[key] **=** reducer(state[key], action);

}

**return** newState;

};

}

Note how all of the reducers are called with the same action.

After we combine our reducers, calling store.getState() would return something like this:

**const** state **=** {

count: 0,

someOtherReducer: 'Some value'

};

This allows use to access each of our reducers state individually e.g.

**const** count **=** state.count;

But this isn’t exactly how we’ll be doing things. As I mentioned before, we wont be manually calling getState or dispatch.

## **Provider**

Here’s where we start to integrate redux with our react application.

To do so we’ll also need to install a module called react-redux.

React redux provides several tools that allow us to easily access store state and dispatch actions from our react components.

The provider is a react component that sits at the root level of your app, and allows any of its children access to the store (which we supply to it as a prop) via [context](https://reactjs.org/docs/context.html) and the connect function (which we’ll get to in a second).

**import** React **from** 'react'

**import** ReactDOM **from** 'react-dom';

**import** { Provider } **from** 'react-redux';

**import** { store } **from** './our-store-file';

**import** Counter **from** './somewhere-else';

**const** App **=** () **=>** (

**<**Provider store**=**{store}**>**

**<**Counter **/>**

**<**/Provider>

);

ReactDOM.render(**<**App **/>**, document.getElementById('app'));

You can also manually supply the store to a connected component (which we’ll cover in a second), which is useful when testing components. I would not, however, recommend giving any of your components direct access to the store in your application.

## **Connecting A Component**

The final piece of the puzzle is the connect function provided by react-redux. This allows us to map parts of the store state to a component, and at the same time, automatically wrap our actions with the dispatch method so that we don’t have to worry about calling it ourselves.

The main benefits of using a provider with this connect function are that an application can be provided an entirely different store when needed, which is very useful for server side rendering, but I wont be covering that today.

Here’s a simple component that allows us to display a number and add to it:

**import** React, { Component } **from** 'react';

**import** { addNumber } **from** './our-actions-file';

**class** Counter **extends** Component {

onAddClick **=** () **=>** {

**this**.props.addNumber(7);

}

render () {

**return** (

**<**div**>**

Count: {**this**.props.count}

**<**button onClick**=**{**this**.onAddClick}**>**

Add 7**!**

**<**/button>

**<**/div>

);

}

}

Right now, if this component was rendered like in the above provider example, it wont have access to either of the props count or addNumber as we are not providing them, but after we connect the component, these state values and actions will be mapped to its props.

Here we’re going to connect our component:

**const** mapStateToProps **=** ({count}) **=>** ({

count

});

**export** **default** connect(mapStateToProps, { addNumber })(Counter);

Let’s break this down a bit…

Firstly, connect is a function that returns another function. Connect takes 2 optional arguments; mapStateToProps, and mapDispatchToProps; and returns a function that takes our component as an argument.

So what are mapStateToProps and mapDispatchToProps?

### **mapStateToProps**

mapStateToProps is a function that will be called when our component mounts, updates, or our store state is changed. All this does is extract the state that we want from the store and return it as an object. The connectfunction then provides these values as props to our component so that we can access them with this.props.count for example.

### **mapDispatchToProps**

mapDispatchToProps, which in this case is simply an object containing our action creator (but can also be a function that allows you to do some more complex stuff), wraps each of our actions with dispatch so that when called with this.props.addNumber(7), for example, automatically dispatches our action. Similarly to mapStateToProps, the connect function provides these values to our component so they can be accessed as props.

An over simplified example of what happens to our mapDispatchToPropsbehind the scenes would look something like this:

**const** mapDispatchToProps **=** (actionCreators) **=>** {

**const** dispatchedActionCreators **=** {};

**for** (**let** key **in** actionCreators) {

**const** actionCreator **=** actionCreators[key];

dispatchedActionCreators[key] **=** (...args) **=>** {

*// For the purposes of this example `dispatch` magically comes out of nowhere*

dispatch(actionCreator(...args));

};

}

**return** dispatchedActionCreators;

};

An alternative to using a mapDispatchToProps function or object, is to provide nothing e.g.

**export** **default** connect(mapStateToProps)(Counter);

This might seem like an odd thing to do at first, but if we do not provide dispatch props then the store’s dispatch method is automatically provided as a prop, so we can manually dispatch actions.

I know I said that we shouldn’t need to call dispatch directly on the store ourselves, but in this case it is fine because we are not accessing the store directly. It’s being provided by the connect function.

Some may prefer this approach as you can avoid shadowing variable names when destructuring actions from props that are also imported, which can occasionally result in calling the wrong function, like in the following example:

**import** React, { Component } **from** 'react';

**import** { addNumber } **from** './our-actions-file';

**class** Counter **extends** Component {

onAddClick **=** () **=>** {

**const** { addNumber } **=** **this**.props;

*// This shares a variable name with the imported action creator*

*// Sometimes the action creator may be accidentally called instead of the dispatch version*

addNumber(7);

}

render () {

**return** (

**<**div**>**

Count: {**this**.props.count}

**<**button onClick**=**{**this**.onAddClick}**>**

Add 7**!**

**<**/button>

**<**/div>

);

}

}

**export** **default** connect(mapStateToProps, { addNumber })(Counter);

An example that uses the dispatch prop rather than mapDispatchToProps:

**import** React, { Component } **from** 'react';

**import** { addNumber } **from** './our-actions-file';

**class** Counter **extends** Component {

onAddClick **=** () **=>** {

*// Now we are directly referencing our imported action creator*

*// And manually dispatching it with the dispatch prop provided by connect*

**this**.props.dispatch(addNumber(7));

}

render () {

**return** (

**<**div**>**

Count: {**this**.props.count}

**<**button onClick**=**{**this**.onAddClick}**>**

Add 7**!**

**<**/button>

**<**/div>

);

}

}

**export** **default** connect(mapStateToProps)(Counter);

## **We’re Done!**

That’s pretty much all you need to know to get started using redux with react, but there are plenty of other complexities to learn to really master redux, and more tools that can be used with redux and react to allow some other functionality e.g.

* Custom mapStateToProps function
* Middleware (which allows applying additional effects to actions)
* Asynchronous actions using something like [redux-thunk](https://github.com/gaearon/redux-thunk) or [redux-saga](https://github.com/redux-saga/redux-saga)(which are middleware)

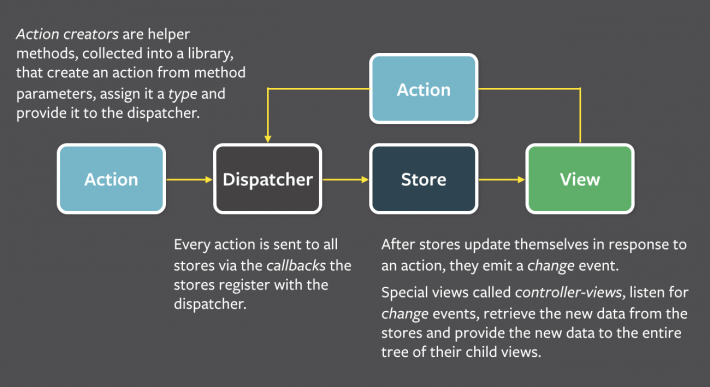
I hope you’ve learnt something useful today. Now go make something awesome!

# Introduction to Redux and React-Redux

## **Flux**

*Flux is the application architecture that Facebook uses for building client-side web applications*

[Flux](http://facebook.github.io/flux/) is a pattern for a unidirectional data flow that inspired Redux so it is interesting to learn about it as well. With Flux the view propagates an **action** through a central **dispatcher** to the various **stores** (holding the application’s data and business logic). As a result the views affected by those changes are updated.

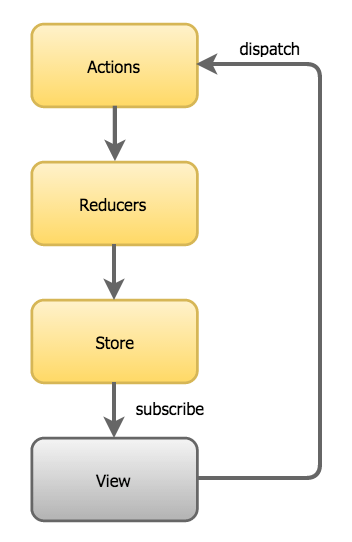


Flux was introduced two years ago and was a big success. Tones of librairies based on the Flux pattern emerged ([Flummox](https://github.com/acdlite/flummox/), [Alt](https://github.com/goatslacker/alt), [Fluxxor](https://github.com/BinaryMuse/fluxxor), etc..), selecting one or the other was a real dilemma at the time (famous JS fatigue). Out of this chaos arose the Redux revolution.

## **Redux**

*Redux is a predictable state container for JavaScript apps.*

Redux evolves from the Flux pattern, but reduces its complexity by removing the **Dispatcher** and by using only one **Store**. Exit the **Dispatcher**, Redux is all about **Store**, **Actions** and **Reducers**.



*As your app grows, instead of adding stores, you split the root reducer into smaller reducers independently operating on the different parts of the state tree.*

You can use Redux together with React, or with any other view library.

### **Store**

With Redux your application has **only one store**. This store is the “single source of truth” that represents the state of your entire application.

### **Actions**

Now that we have our initial state tree, how to modify it? To modify it we need to create actions. An action is an object describing the changes we want to make to our state tree.

Here is an action that will open our menu:

|  |
| --- |
| const action = {    type: 'MENU\_OPEN'  } |

Here is another on that will change the firstname of our user:

|  |
| --- |
| const action = {    type: 'USER\_UPDATE\_FIRSTNAME',    newFirstName: 'Marie'  } |

We are now ready to make changes to our state tree! To do so we will use Reducers.

### **Reducers**

Reducers are pure functions that take the previous state and an action, and return the next state.

Pure functions can be described as the following:

* Predictable
* Testable
* Declarative
* Return a new reference
* Do not mutate arguments

Here is our reducer that handle the menu state:

|  |
| --- |
| // Initial state to false meaning the menu is closed by default  function menuReducer(state = false, action) {    switch (action.type) {    case 'MENU\_OPEN':      return true    case 'MENU\_CLOSE':      return false    case 'MENU\_TOGGLE':      return !state    default:      return state    }  } |

Several things to notice here:

* We always return the previous state as default. This is in case no actions match, we do not want anything to change so we return the old state.
* We always return a new reference. That’s what pure functions do. Immutability has also a performance benefit as your app can know if your state changed or not simply by === it.
* We always have an initial state. That will be the value of your state when the app starts.

Now that we know our primitive concepts: Store, Actions and Reducers we need to learn how everything work together.

### **Creating a store**

To create a store you need to register all the reducers of your app using combineReducersmethod:

|  |
| --- |
| import { createStore, combineReducers } from 'redux'  import reducerIsMenuOpened from './reducers/isMenuOpened'  import reducerUser from './reducers/user'    // Create a store with several reducers  let store = createStore(combineReducers({    isMenuOpened: reducerIsMenuOpened,    user: reducerUser  }) |

Now we can use the store reference to subscribe, dispatch, or getState. These are the only methods you will need.

getState method returns the state tree at the moment you ask:

|  |
| --- |
| store.getState() |

subscribe is called whenever the state tree changes:

|  |
| --- |
| store.subscribe(() => console.log(store.getState()) |

The only way to change the state tree is to emit an action through a dispatch method:

|  |
| --- |
| store.dispatch(action) |

Let’s try it out!

|  |
| --- |
| // Subscribe to any change and dump in the console  store.subscribe(() => console.log(store.getState())    // Dump the initial state:  console.log(store.getState())  /\*  {    "isMenuOpened": false,    "user" : {      "firstname": "Julien"    }  }  \*/  const actionMenuOpen = {    type: 'MENU\_OPEN'  }  store.dispatch(actionMenuOpen)  // Dumps  /\*  {    "isMenuOpened": true,    "user" : {      "firstname": "Julien"    }  }  \*/  const actionChangeFirstName = {    type: 'USER\_UPDATE\_FIRSTNAME',    newFirstName: 'Marie'  }  store.dispatch(actionChangeFirstName)  // Dumps  /\*  {    "isMenuOpened": true,    "user" : {      "firstname": "Marie"    }  }  \*/ |

Now that we learnt Redux basics (Store, Actions and Reducers), let see how to use it with React.

## **React-redux**

[React-redux](https://github.com/reactjs/react-redux) is the official library that supports React bindings for Redux.

### **Installation**

|  |
| --- |
| npm install --save redux react-redux |

### **Creating a store**

With **React-redux** you do not need to subscribe to store changes in your components.

To create the store we only need redux and it is exactly the same as if you were using vanilla ES6 JavaScript.

|  |
| --- |
| import { combineReducers } from 'redux'  import reducerIsMenuOpened from './reducers/isMenuOpened'  import reducerUser from './reducers/user'    export default createStore(combineReducers({    isMenuOpened: reducerIsMenuOpened,    user: reducerUser  }) |

Our store is now created, all we need to do is to make it available to every component of our application. To do so we need to wrap our RootComponent (it could be anything) with **React-redux’s** Provider component.

|  |
| --- |
| import React from 'react'  import ReactDOM from 'react-dom'  import { Provider } from 'react-redux'  import store from './pathToPreviousFile'    ReactDOM.render(    <Provider store={store}>      <RootComponent />    </Provider>,    document.getElementById('root')  ) |

The store is now available to any component of your application through context but you will not use it yourself. Interacting with the store is the role of Containers.

### **Containers**

Containers are high order components that aim to communicate with the store for you. The container gives you access to your state tree and to the dispatch method.

Let’s create a simple menu component to explain how containers work.

|  |
| --- |
| const Menu = ({    isMenuOpened,    toggleMenu  }) => (    <div>      <button onClick={toggleMenu}>Toggle menu</button>      <ul className={isMenuOpened ? 'menu-opened' : 'menu-closed'}>        <li>Menu 1</li>        <li>Menu 2</li>      </ul>    </div>  )  export default Menu |

As you can see our component is stateless and only has two props:

* isMenuOpened: set a specific CSS class to display or hide the menu
* toggleMenu: Dispatch an action to toggle the menu isMenuOpened flag

The Menu component does not have any clue if your app uses Redux or not. It is what we call a presentational component, it only communicate through props.

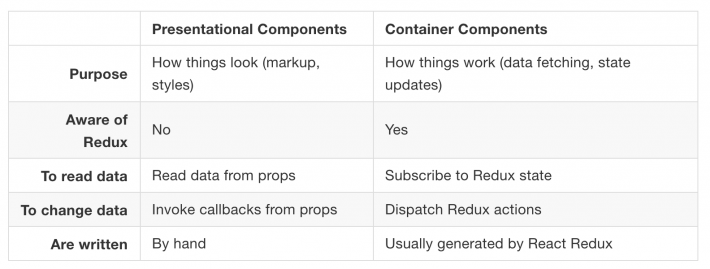
Now let’s create a container to wrap our Menu presentational component:

|  |
| --- |
| import { connect } from 'react-redux'  import Menu from './Menu'  import {toggleMenu} from '../actions/menu'    const mapStateToProps = (state) => {    return {      isMenuOpened: state.isMenuOpened    }  }    const mapDispatchToProps = (dispatch) => {    return {      toggleMenu: () => dispatch(toggleMenu())    }  }    export default connect(    mapStateToProps,    mapDispatchToProps  )(Menu) |

The connect method creates a new component from mapStateToProps (allow you to select the data necessary to your component and insert them as props), mapDispatchToProps(allow you to dispatch actions to modify the state tree) and the Menu component.

Instead of using the Menu component now you can use the MenuContainer that is aware of Redux and is meant to interact with the app store.

Here is a little recap of the differences between a presentational component and a container the taken from [Redux](http://redux.js.org/docs/basics/UsageWithReact.html) documentation.



Now you have all the knowledge necessary to create your own app using React and

# How Redux Works: A Counter-Example

OCTOBER 29, 2017[Follow @dceddia](https://twitter.com/intent/follow?screen_name=dceddia)

After learning a bit about React and getting into Redux, it’s really confusing how it all works.

Actions, reducers, action creators, middleware, pure functions, immutability…

Most of these terms seem totally foreign.

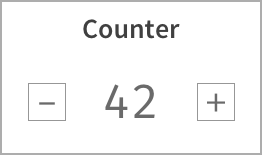
So in this post we’re going to demystify how Redux works with a backwards approach that I think will help your understanding. As in the [last post](https://daveceddia.com/what-does-redux-do/), I’ll try to explain Redux in simple terms before tackling the terminology.

If you’re not yet sure what Redux is for or why you should use it, read [this post first](https://daveceddia.com/what-does-redux-do/)and then come back here.

## **First: Plain React State**

We’ll start with an example of plain old React state, and then add Redux piece-by-piece.

Here is a counter:



And here’s the code (I left out the CSS to keep this simple, so it won’t be as pretty as the image):

import React from 'react';

class Counter extends React.Component {

state = { count: 0 }

increment = () => {

this.setState({

count: this.state.count + 1

});

}

decrement = () => {

this.setState({

count: this.state.count - 1

});

}

render() {

return (

<div>

<h2>Counter</h2>

<div>

<button onClick={this.decrement}>-</button>

<span>{this.state.count}</span>

<button onClick={this.increment}>+</button>

</div>

</div>

)

}

}

export default Counter;

As a quick review, here’s how this works:

* The count state is stored in the top level Counter component
* When the user clicks “+”, the button’s onClick handler is called, which is bound to the increment function in the Counter component.
* The increment function updates the state with the new count.
* Because state was changed, React re-renders the Counter component (and its children), and the new counter value is displayed.

If you need more detail about how state changes work, go read [A Visual Guide to State in React](https://daveceddia.com/visual-guide-to-state-in-react/) and then come back here. Seriously: if the above was not review for you, you need to learn how React state works before you learn Redux.

#### **Quick Setup**

If you’d like to follow along with the code, create a project now:

* Install create-react-app if you don’t have it (npm install -g create-react-app)
* Create a project: create-react-app redux-intro
* Open src/index.js and replace it with this:

import React from 'react';

import { render } from 'react-dom';

import Counter from './Counter';

const App = () => (

<div>

<Counter />

</div>

);

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

* Create a src/Counter.js with the code from the Counter example above.

## **Now: Add Redux**

As [discussed in Part 1](https://daveceddia.com/what-does-redux-do/), Redux keeps the **state** of your app in a single **store**. Then, you can extract parts of that state and plug it into your components as props. This lets you keep data in one global place (the store) and feed it directly to anycomponent in the app, without the gymnastics of passing props down multiple levels.

Side note: you’ll often see the words “state” and “store” used interchangably. Technically, the **state** is the data, and the **store** is where it’s kept.

As we go through the steps below, follow along in your editor! It will help you understand how this works (and we’ll get to work through some errors together).

Add Redux to the project:

$ yarn add redux react-redux

#### **redux vs react-redux**

Wait – 2 libraries? “What’s react-redux,” you say? Well, I’ve kinda been lying to you (sorry).

See, redux gives you a store, and lets you keep state in it, and get state out, and respond when the state changes. But that’s all it does. It’s actually react-reduxthat lets you connect pieces of the state to React components. That’s right: reduxknows nothing about React at all.

These libraries are like two peas in a pod. 99.999% of the time, when anyone mentions “Redux” in the context of React, they are referring to both of these libraries in tandem. So keep that in mind when you see Redux mentioned on StackOverflow, or Reddit, or [elsewhere](https://daveceddia.com/keeping-up-with-javascript/).

## **Last Things First**

Most tutorials start by creating a store, setting up Redux, writing a reducer, and so on. Lots must happen before anything appears on screen.

I’m going to take a backwards approach, and it will take just as much code to make things appear on screen, but hopefully the motivation behind each step will be clearer.

Here’s a video walkthrough of converting the Counter to use Redux (or if video is not your thing, keep on reading!).

Back to the Counter app, let’s just imagine for a second that we moved the component’s state into Redux.

We’ll remove the state from the component, since we’ll be getting that from Redux soon:

import React from 'react';

class Counter extends React.Component {

increment = () => {

// fill in later

}

decrement = () => {

// fill in later

}

render() {

return (

<div>

<h2>Counter</h2>

<div>

<button onClick={this.decrement}>-</button>

<span>{this.props.count}</span>

<button onClick={this.increment}>+</button>

</div>

</div>

)

}

}

export default Counter;

## **Wiring Up The Counter**

Notice that {this.state.count} changed to {this.props.count}. This won’t work yet, of course, because the Counter is not receiving a count prop. We’re gonna use Redux to inject that.

To get the count out of Redux, we first need to import the connect function at the top:

import { connect } from 'react-redux';

Then we need to “connect” the Counter component to Redux at the bottom:

// Add this function:

function mapStateToProps(state) {

return {

count: state.count

};

}

// Then replace this:

// export default Counter;

// With this:

export default connect(mapStateToProps)(Counter);

This will fail with an error (more on that in a second).

Where previously we were exporting the component itself, now we’re wrapping it with this connect function call.

#### **What’s connect?**

You might notice the call looks little… weird. Why connect(mapStateToProps)(Counter) and not connect(mapStateToProps, Counter) or connect(Counter, mapStateToProps)? What’s that doing?

It’s written this way because connect is a higher-order function, which is a fancy way of saying it returns a function when you call it. And then calling that function with a component returns a new (wrapped) component.

Another name for this is a [higher-order component](https://daveceddia.com/extract-state-with-higher-order-components/) (aka “HOC”). HOCs have gotten some bad press lately, but they’re still quite useful, and connect is a good example of a useful one.

What connect does is hook into Redux, pull out the entire state, and pass it through the mapStateToProps function that you provide. This needs to be a custom function because only you will know the “shape” of the state in Redux.

connect passes the entire state as if to say, “Hey, tell me what you need out of this jumbled mess.”

The object you return from mapStateToProps gets fed into your component as props. The example above will pass state.count as the value of the count prop: the keys in the object become prop names, and their corresponding values become the props’ values. So you see, this function literally defines a mapping from state into props.

## **Errors Mean Progress!**

If you’re following along, you will see an error like this in the console:

*Could not find “store” in either the context or props of “Connect(Counter)”. Either wrap the root component in a , or explicitly pass "store" as a prop to "Connect(Counter)".*

Since connect pulls data from the Redux store, and we haven’t set up a store or told the app how to find it, this error is pretty logical. Redux has no dang idea what’s going on right now.

## **Provide a Store**

Redux holds the global state for the entire app, and by wrapping the entire app with the Provider component from react-redux, every component in the app tree will be able to use connect to access the Redux store if it wants to.

This means App, and children of App (like Counter), and children of their children, and so on – all of them can now access the Redux store, but only if they are explicitly wrapped by a call to connect.

I’m not saying to actually do that – connecting every single component would be a bad idea (messy design, and slow too).

This Provider thing might seem like total magic right now. It is a little bit; it actually uses React’s “context” feature under the hood.

It’s like a secret passageway connected to every component, and using connectopens the door to the passageway.

Imagine pouring syrup on a pile of pancakes, and how it manages to make its way into ALL the pancakes even though you just poured it on the top one. Providerdoes that for Redux.

In src/index.js, import the Provider and wrap the contents of App with it.

import { Provider } from 'react-redux';

...

const App = () => (

<Provider>

<Counter/>

</Provider>

);

We’re still getting that error though – that’s because Provider needs a store to work with. It’ll take the store as a prop, but we need to create one first.

## **Create the Store**

Redux comes with a handy function that creates stores, and it’s called createStore. Yep. Let’s make a store and pass it to Provider:

import { createStore } from 'redux';

const store = createStore();

const App = () => (

<Provider store={store}>

<Counter/>

</Provider>

);

Another error, but different this time:

*Expected the reducer to be a function.*

So, here’s the thing about Redux: it’s not very smart. You might expect that by creating a store, it would give you a nice default value for the state inside that store. Maybe an empty object?

But no: Redux makes zero assumptions about the shape of your state. It’s up to you! It could be an object, or a number, or a string, or whatever you need. So we have to provide a function that will return the state. That function is called a **reducer** (we’ll see why in a minute). So let’s make the simplest one possible, pass it into createStore, and see what happens:

function reducer() {

// just gonna leave this blank for now

// which is the same as `return undefined;`

}

const store = createStore(reducer);

## **The Reducer Should Always Return Something**

The error is different now:

*Cannot read property ‘count’ of undefined*

It’s breaking because we’re trying to access state.count, but state is undefined. Redux expected our reducer function to return a value for state, except that it (implicitly) returned undefined. Things are rightfully broken.

The reducer is expected to return the state. It’s actually supposed to take the current state and return the new state, but nevermind; we’ll come back to that.

Let’s make the reducer return something that matches the shape we need: an object with a count property.

function reducer() {

return {

count: 42

};

}

Hey! It works! The count now appears as “42”. Awesome.

Just one thing though: the count is forever stuck at 42.

## **The Story So Far**

Before we get into how to actually update the counter, let’s look at what we’ve done up til now:

* We wrote a mapStateToProps function that does what the name says: transforms the Redux state into an object containing props.
* We connected the Redux store to our Counter component with the connectfunction from react-redux, using the mapStateToProps function to configure how the connection works.
* We created a reducer function to tell Redux what our state should look like.
* We used the ingeniously-named createStore function to create a store, and passed it the reducer.
* We wrapped our whole app in the Provider component that comes with react-redux, and passed it our store as a prop.
* The app works flawlessly, except the fact that the counter is stuck at 42.

With me so far?

## **Interactivity (Making It Work)**

So far this is pretty lame, I know. You could’ve written a static HTML page with the number “42” and 2 broken buttons in 60 seconds flat, yet here you are, reading how to overcomplicate that very same thing with React and Redux and who knows what else.

I promise this next section will make it all worthwhile.

Actually, no. I take that back. A simple Counter app is a great teaching tool, but Redux is absolutely overkill for something like this. React state is perfectly fine for something so simple. Heck, even plain JS would work great. Pick the right tool for the job. Redux is not always that tool. But I digress.

## **Initial State**

So we need a way to tell Redux to change the counter.

Remember the reducer function we wrote? (of course you do, it was 2 minutes ago)

Remember how I mentioned it takes the current state and returns the new state? Well, I lied again. It actually takes the current state and an action, and then it returns the new state. We should have written it like this:

function reducer(state, action) {

return {

count: 42

};

}

The very first time Redux calls this function, it will pass undefined as the state. That is your cue to return the initial state. For us, that’s probably an object with a count of 0.

It’s common to write the initial state above the reducer, and use ES6’s default argument feature to provide a value for the state argument when it’s undefined.

const initialState = {

count: 0

};

function reducer(state = initialState, action) {

return state;

}

Try this out. It should still work, except now the counter is stuck at 0 instead of 42. Awesome.

## **Action**

We’re finally ready to talk about the action parameter. What is it? Where does it come from? How can we use it to change the damn counter?

An “action” is a JS object that describes a change that we want to make. The only requirement is that the object needs to have a type property, and its value should be a string. Here’s an example of an action:

{

type: "INCREMENT"

}

Here’s another one:

{

type: "DECREMENT"

}

Are the gears turning in your head? Do you know what we’re gonna do next?

## **Respond to Actions**

Remember the reducer’s job is to take the current state and an action and figure out the new state. So if the reducer received an action like { type: "INCREMENT" }, what might you want to return as the new state?

If you answered something like this, you’re on the right track:

function reducer(state = initialState, action) {

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

return {

count: state.count + 1

};

}

return state;

}

It’s common to use a switch statement with cases for each action you want to handle. Change your reducer to look like this:

function reducer(state = initialState, action) {

switch(action.type) {

case 'INCREMENT':

return {

count: state.count + 1

};

case 'DECREMENT':

return {

count: state.count - 1

};

default:

return state;

}

}

#### **Always Return a State**

You’ll notice that there’s always the fallback case where all it does is return state. This is important, because Redux can (will) call your reducer with actions that it doesn’t know what to do with. In fact, the very first action you’ll receive is { type: "@@redux/INIT" }. Try putting a console.log(action) above the switchand see.

Remember that the reducer’s job is to return a new state, even if that state is unchanged from the current one. You never want to go from “having a state” to “state = undefined”, right? That’s what would happen if you left off the defaultcase. Don’t do that.

#### **Never Change State**

One more thing to never do: do not mutate the state. State is immutable. You must never change it. That means you can’t do this:

function brokenReducer(state = initialState, action) {

switch(action.type) {

case 'INCREMENT':

// NO! BAD: this is changing state!

state.count++;

return state;

case 'DECREMENT':

// NO! BAD: this is changing state too!

state.count--;

return state;

default:

// this is fine.

return state;

}

}

You also can’t do things like state.foo = 7, or state.items.push(newItem), or delete state.something.

Think of it like a game where the only thing you can do is return { ... }. It’s a fun game. Maddening at first. But you’ll get better at it with practice.

I put together a short guide on how to do immutable updates, showing 7 common patterns for updating state within objects and arrays.

[Get the Guide: Immutable Updates in Redux](https://daveceddia.com/how-does-redux-work/#ck_modal)

#### **All These Rules…**

Always return a state, never change state, don’t connect every component, eat your broccoli, don’t stay out past 11… it’s exhausting. It’s like a rules factory, and I don’t even know what that is.

Yeah, Redux can be like an overbearing parent. But it comes from a place of love. Functional programming love.

Redux is built on the idea of immutability, because mutating global state is the road to ruin.

Have you ever kept a global object and used it to pass state around an app? It works great at first. Nice and easy. And then the state starts changing in unpredictable ways and it becomes impossible to find the code that’s changing it.

Redux avoids these problems with some simple rules. State is read-only, and actions are the only way to modify it. Changes happen one way, and one way only: action -> reducer -> new state. The reducer function must be “pure” – it cannot modify its arguments.

There are even addon packages that let you log every action that comes through, rewind and replay them, and anything else you could imagine. Time-travel debugging was one of the original motivations for creating Redux.

## **Where Do Actions Come From?**

One piece of this puzzle remains: we need a way to feed an action into our reducer function so that we can increment and decrement the counter.

Actions are not born, but they are **dispatched**, with a handy function called dispatch.

The dispatch function is provided by the instance of the Redux store. That is to say, you can’t just import { dispatch } and be on your way. You can call store.dispatch(someAction), but that’s not very convenient since the storeinstance is only available in one file.

As luck would have it, the connect function has our back. In addition to injecting the result of mapStateToProps as props, connect also injects the dispatchfunction as a prop. And with that bit of knowledge, we can finally get the counter working again.

Here is the final component in all its glory. If you’ve been following along, the only things that changed are the implementations of increment and decrement: they now call the dispatch prop, passing it an action.

import React from 'react';

import { connect } from 'react-redux';

class Counter extends React.Component {

increment = () => {

this.props.dispatch({ type: 'INCREMENT' });

}

decrement = () => {

this.props.dispatch({ type: 'DECREMENT' });

}

render() {

return (

<div>

<h2>Counter</h2>

<div>

<button onClick={this.decrement}>-</button>

<span>{this.props.count}</span>

<button onClick={this.increment}>+</button>

</div>

</div>

)

}

}

function mapStateToProps(state) {

return {

count: state.count

};

}

export default connect(mapStateToProps)(Counter);

The code for the entire project (all two files of it) can be found [on Github](https://github.com/dceddia/redux-intro).

## **What Now?**

With the Counter app under your belt, you are well-equipped to learn more about Redux.

*“What?! There’s more?!”*

There is much I haven’t covered here, in hopes of making this guide easily digestible – action constants, action creators, middleware, thunks and asynchronous calls, selectors, and on and on. There’s a lot. The [Redux docs](https://redux.js.org/) are well-written and cover all that and more.

But you’ve got the basic idea now. Hopefully you understand how data flows in Redux (dispatch(action) -> reducer -> new state -> re-render), and what a reducer does, and what an action is, and how that all fits together.

I’m putting together a new course that will cover all of this and more! [Sign up here](https://daveceddia.com/how-does-redux-work/#ck_modal)to be notified.

## 1.1   What is state?

React components have the concept of local, or component state.

Within any given component, you can keep track of the value of an input field or whether a button has been toggled, for example. Local state makes easy work of managing a single component’s behavior.

However, today’s single page applications often require synchronizing a complex web of state. Nested levels of components may render a different user experience based on the pages a user has already visited, the status of an AJAX request, or whether a user is logged in.

Let’s consider a use case involving the authentication status of a user. Your product manager tells you that when a user is logged into an ecommerce store, the navigation bar should display the user’s avatar image, the store should display items nearest to the user’s zip code first, and the newsletter signup form should be hidden.

Within a vanilla React architecture, your options are limited for syncing state across each of the components. In the end, you’ll likely end up passing the authentication status and additional user data from one top-level component down to each of these nested components.

This architecture has several disadvantages. Along the way, data may filter through components that have no use for it other than to pass the data on to their children.

In a large application, this can result in tons of data moving through unrelated components, passed down via props or passed up via callbacks.

It’s likely that a small number of components at the top of the application end up with an awareness of most of the state used throughout the entire application. At a certain scale, maintaining and testing this code becomes untenable. Because React was not intended to solve the same breadth of problems that other MVC frameworks attempted to address, an opportunity existed to bridge those gaps.

With React in mind, Facebook eventually introduced Flux, an architecture pattern for web applications. Flux became tremendously influential in the world of front-end development and began a shift in how we thought about state management in client-side applications. Facebook offered its own implementation of this pattern, but soon more than a dozen “Flux-inspired” state management libraries emerged and competed for React developers’ attention.

This was a tumultuous time for React developers looking to scale an application. We saw the light with Flux, but continued to experiment to find more elegant ways to manage complex state in applications. For a time, newcomers encountered a paradox of choice; a divided community effort had produced so many options, it was anxiety-inducing. To our surprise and delight though, the dust is already settling and Redux has emerged as a clear winner.

Redux took the React world by storm with a simple premise, a big payoff, and a memorable introduction. The premise is to store your entire application state in a single object using pure functions. The payoff is totally predictable application state. The introduction, for most early users, came in Dan Abramov’s 2015 React Europe conference talk, titled “Live React: Hot Reloading with Time Travel.” Dan wowed attendees by demonstrating a Redux developer experience that blew established workflows out of the water. A technique called “hot loading” made live application updates while maintaining existing state, and his nascent Redux developer tools enabled you to “time travel” through application state — rewinding and replaying user actions with a single click.

To understand Redux, we’d first like to properly introduce you to Flux, the architecture pattern developed at Facebook and credited to Jing Chen. Redux and many of its alternatives are variations of this Flux architecture.

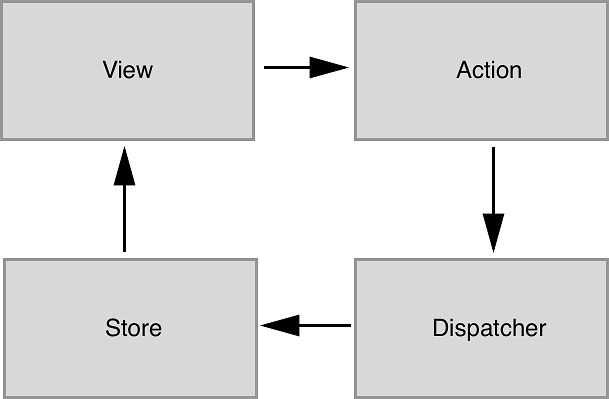
## 1.2   What is Flux?

Flux is foremost an architecture pattern. It was developed as an alternative to the prevailing Model-View-Controller (MVC) JavaScript patterns popularized by incumbent frameworks, such as Backbone, Angular, or Ember. Although each framework puts its own spin on the MVC pattern, many share similar frustrations: generally, the flow of data between models, views and controllers can be difficult to follow.

Many of these frameworks use two-way data binding, in which changes to the views update corresponding models, and changes in the models update corresponding views. When any given view can update one or more models, which in turn can update more models, you can’t be blamed for losing track of the expected outcome at a certain scale. Chen contested that although MVC frameworks work well for smaller applications, the two-way data-binding models that many of them employ don’t scale well enough for the size of Facebook’s application. Developers at the company became apprehensive of making changes, for fear of the tangled web of dependencies producing unintended consequences.

Flux sought to address the unpredictability of state and the fragility of a tightly coupled model and view architecture. Chen went about scrapping the two-way data binding model in favor of a unidirectional data flow. Instead of permitting each view to interact with their corresponding models, Flux requires all changes to state to follow a single path. When a user clicks a Submit button on a form, for example, an action is sent to the application’s one and only dispatcher. The dispatcher will then send the data through to the appropriate data stores for updating. Once updated, the views will become aware of the new data to render. Figure 1.1 illustrates this unidirectional data flow.

##### Figure 1.1 Flux specifies that data must flow in a single direction.



### 1.2.1   Actions

Every change to state starts with an action (figure 1.1). An action is a JavaScript object describing an event in your application. They are typically generated by either a user interaction or by a server event, like an HTTP response.

### 1.2.2   Dispatcher

All data flow in a Flux application is funneled through a single dispatcher. The dispatcher itself has very little functionality, because its purpose is to receive all actions and send them to each store that has been registered. Every action will be sent to every store.

### 1.2.3   Stores

Each store manages the state of one domain within an application. In an ecommerce site, you may expect to find a shopping cart store and a product store, for example. Once a store is registered with the dispatcher, it will begin to receive actions. When it receives an action type that it cares about, the store will update accordingly. Once a change to the store is made, an event is broadcast to let the views know to update using the new state.

### 1.2.4   Views

Flux may have been designed with React in mind, but the views aren’t required to be React components. For their part, the views need only subscribe to the stores they wish to display data from. The Flux documentation encourages the use of the controller-view pattern, whereby a top-level component handles communication with the stores and passes data to child components. Having both a parent and a nested child component communicating with stores can lead to extra renders and unintended side-effects.

Again, Flux is an architecture pattern first. The Facebook team maintains one simple implementation of this pattern, aptly (or confusingly, depending on your perspective) named Flux. Many alternative implementations have emerged since 2014, including Alt, Reflux, and Redux. A more comprehensive list of those alternative implementations can be found in section 1.4.

## 1.3   What is Redux?

We can’t put it much better than the official docs: “Redux is a predictable state container for JavaScript applications.” It’s a standalone library, but is used most often as a state management layer with React. Like Flux, its major goal is to bring consistency and predictability to the data in our applications. Redux divides the responsibilities of state management into a few separate units:

* The store holds all of your application state in a single object. (We’ll commonly refer to this object as the state tree.)
* The store can be updated only via actions, an object describing an event.
* Functions known as reducers specify how to transform application state. Reducers are functions that take the current state in the store and an action, and return the next state after applying any updates.

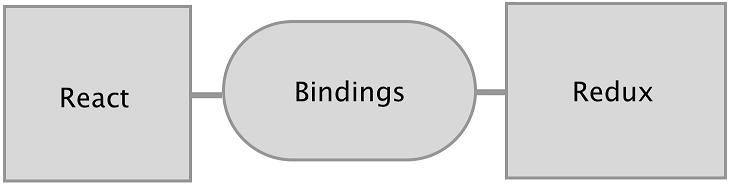
Technically speaking, Redux may not qualify as a Flux implementation. It nontrivially deviates from some of the components of the prescribed Flux architecture, such as the removal of the dispatcher altogether. Ultimately though, Redux is very Flux-like and the distinction is a matter of semantics.

Redux enjoys the benefits of a predictable data flow from the Flux architecture, but also found ways to alleviate the uncertainty of store callback registrations. As alluded to in the previous section, it can be a pain to reconcile the state of multiple Flux stores. Redux, instead, prescribes a single store to manage the state of an entire application. We’ll learn more about how this works and what the implications are in the coming sections.

### 1.3.1   React and Redux

Although Redux was designed and developed in the context of React, the two libraries are completely decoupled. React and Redux are connected via bindings, shown simply in figure 1.2.

##### Figure 1.2 Redux isn’t part of any existing framework or library, but additional tools called **bindings** connect Redux with React. Over the course of the book we’ll use the react-redux package for this.



It turns out that the Redux paradigm for state management can be implemented alongside most JavaScript frameworks. Bindings exist for Angular 1 and 2, Backbone, Ember, and many more technologies.

Although this book is fundamentally about Redux, our treatment of it is closely tied with React. Redux is a small standalone library, but it fits particularly well with React components. Redux will help you define what your application does; React will handle how your application looks.

Most of the code we’ll write over the course of the book, not to mention most of the React/Redux code you’ll write period, will fall into a few categories:

* The application’s state and behavior, handled by Redux.
* Bindings, provided by the react-redux package, that connect the data in the Redux store with the view (React components).
* Stateless components that comprise much of your view layer.

You’ll find that React is a very natural ecosystem for Redux. While React has mechanisms to manage state directly in components, the door is wide open for Redux to come in and manage the greater application state. If you’re interested in an alternative ecosystem, chapter 12 explores the relationship between Redux and some of the other JavaScript frameworks.

### 1.3.2   The Three Principles

You will have covered substantial ground by grokking that state in Redux is represented by a single source of truth, is read-only, and changes to it must be made with pure functions.

#### Single source of truth

Unlike the various domain stores prescribed by the Flux architecture, Redux manages an entire application’s state in one object, inside of one store. The use of a single store has important implications. The ability to represent the entire application state in a single object simplifies the developer experience; it becomes dramatically easier to think through the application flow, predict the outcome of new actions, and debug issues produced by any given action. The potential for time travel debugging, or the ability flip back and forth through snapshots of application state, is what inspired the creation of Redux in the first place.

#### State is read-only

Just like Flux, actions are the only way to initiate changes in application state. No stray AJAX call can produce a change in state without being communicated via an action. Redux differs from many Flux implementations though, in that these actions do not result in a mutation of the data in the store. Instead, each action results in a shiny new instance of the state to replace the current one. More on that subject in the next bullet point.

#### Changes are made with pure functions

Actions are received by reducers. It’s important that these reducers be pure functions. Pure functions are deterministic; they always produce the same output given the same inputs, and they don’t mutate any data in the process. If a reducer were to mutate the existing state while producing the new one, we may end up with erroneous new state, but we also lose the predictable transaction log that each new action is intended to provide. The Redux developer tools and other features, such as undo and redo functionality, rely on application state being computed by pure functions.

### 1.3.3   The Workflow

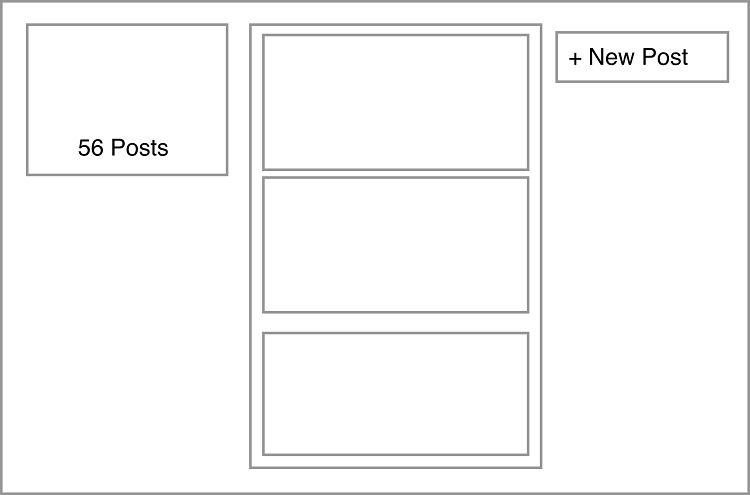
So far, we’ve touch briefly upon things like actions, reducers, and the store, but in this section, we’ll cover each in more depth. What’s important to take away here is the role that each element plays, and how they work together to produce a desired result. For now, don’t worry about finer implementation details, because we’ll have plenty of time in later chapters to apply the concepts we’re about to explore.

Modern web applications are ultimately about handling events. They could be initiated by a user, like navigating to a new page or submitting a form. Or they could be initiated by another external source, like a server response. Responding to events usually involves updating state and re-rendering with that updated state. The more that your application does, the more state you need to track and update. Combine this with the fact that most of these events occur asynchronously, and you suddenly have some real obstacles to maintaining an application at scale.

Redux exists to create structure around how you handle events and manage state in your application, hopefully making you a more productive and happy human in the process.

Let’s look at how we might handle a single event in an application using Redux and React. Say you were tasked with implementing one of the core features of a social network, adding a post to your activity feed. Here’s a quick mockup of a user profile page, which may or may not take its inspiration from Twitter.

##### Figure 1.3 A simple mockup of a profile page. This page is backed by two main pieces of data: the total post count, and the list of post objects in the user’s activity feed.

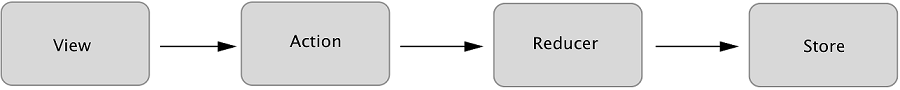


The following distinct steps are involved in handling an event like a new post:

* From the view, indicate that an event has occurred (a post submission) and pass along the necessary data (the content of the post to be created).
* Update state based on the type of event — add an item to the user’s activity feed and increment the post count.
* Re-render the view to reflect the updated state.

Sounds reasonable, right? If you’ve used React before, you’ve likely implemented features like this directly in components. Redux takes a different approach. Code to satisfy the three tasks is moved out of React components into a few separate entities. We’re already familiar with the View in figure 1.4, but we’re excited to introduce a new cast of characters you’ll hopefully learn to love.

##### Figure 1.4 A look at how data flows through a React/Redux application. We’ve omitted a few common pieces like middleware and selectors, which we’ll cover in-depth in later chapters.



#### Actions

We want to do two things in response to the user submitting a new post: add the post to the user’s activity feed and increment their total post count. After the user submits, we’ll kick off the process by dispatching an action. Actions are plain old JavaScript objects that represent an event in your application, as follows:

1

2

3

4

5

6

{

**type**: 'CREATE\_POST',

payload: {

body: 'All that is gold does not glitter'

}

}

[copy](javascript:void(0))

Let’s break that down. We have an object with two properties:

* type — a string that represents the category of action being performed. By convention, this property is capitalized and uses underscores as delimiters.
* payload — an object that provides the data necessary to perform the action. In our case, we only need one field: the contents of the message we want to post. The name “payload” is just a popular convention.

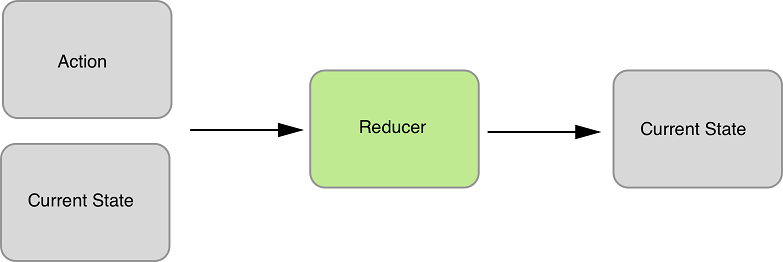
Actions have the advantage of serving as audits. There’s a historical record of everything happening in your application, including any data needed to complete a transaction. It’s hard to understate how valuable this is in maintaining a grasp on a complex application. Once you get used to having a highly readable stream describing the behavior of your application in real-time, you’ll find it hard to live without.

Throughout the book, we’ll frequently come back to this idea of whatversus how. You can think of Redux as decoupling what happens in an application from how we respond to an event. Actions handle the what in this equation. They simply describe an event, they don’t know and don’t care what happens downstream. Somewhere down the road we’ll eventually have to specify how to handle an action. Sounds like a job fit for a reducer!

#### Reducers

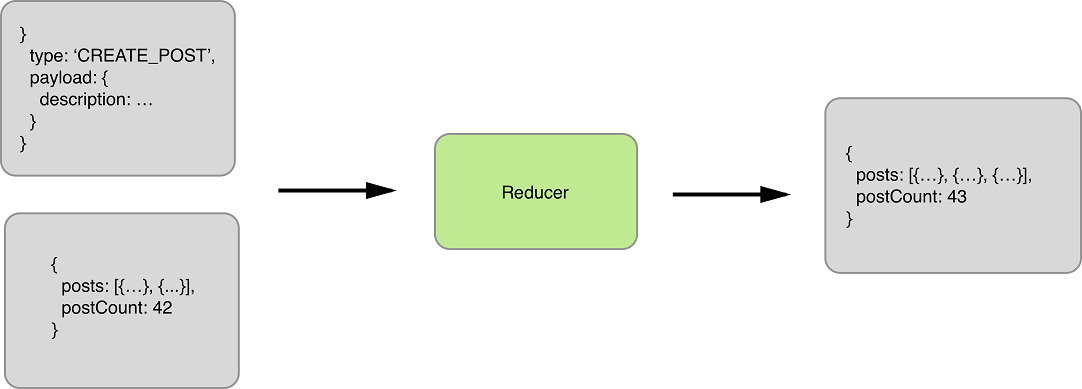
Reducers are functions responsible for updating your state in response to actions. They are simple functions that take your current state and an action as arguments, and return the next state.

##### Figure 1.5 An abstract representation of a reducer’s function signature. If this diagram looks simple, that’s because it is! Reducers are meant to be simple functions that compute a result, making them easy to work with and test.



Reducers are typically very easy to work with. Just like all pure functions, they produce no side-effects. They don’t affect the outside world in any way, and they’re referentially transparent. The same inputs will always yield the same return value. This makes them particularly easy to test. Given certain inputs, you can verify that you receive the expected result. Here’s how our reducer might update the list of posts and the total post count:

##### Figure 1.6 Visualizing a reducer hard at work. It accepts as input an action and the current state. The reducer’s only responsibility is to calculate the next state based on these arguments. No mutations, no side-effects, no funny business. Data in, data out.



We’re focusing on a single event in this example, which means we need only one reducer. However, you certainly aren’t limited to only one. In fact, more sizable applications frequently implement several reducer functions, each concerned with a different slice of the state tree. These reducers are combined, or composed, into a single “root reducer.”

#### Store

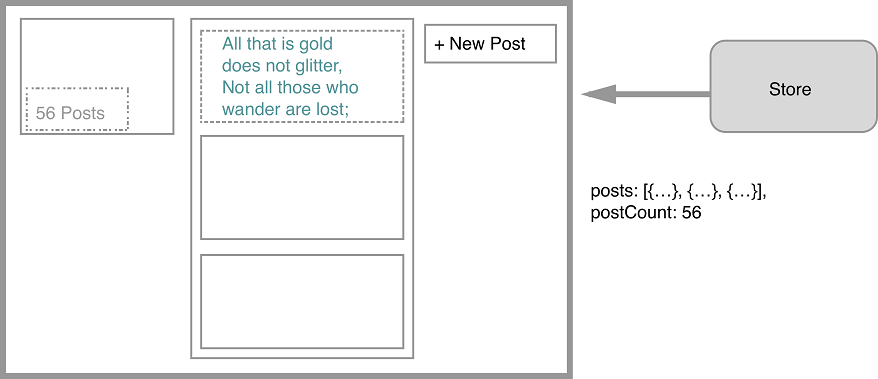
Reducers describe how to update state in response to an action, but can’t modify state directly. That privilege rests solely with the store.

In Redux, application state is stored in a single object. The store has a few main roles, which follow:

* Hold application state.
* Provide a way to access state.
* Provide a way to specify updates to state. The store requires an action be dispatched to modify state.
* Allow other entities to subscribe to updates (React components in our case). View bindings provided by react-redux will allow us to receive updates from the store and respond to them in our components.

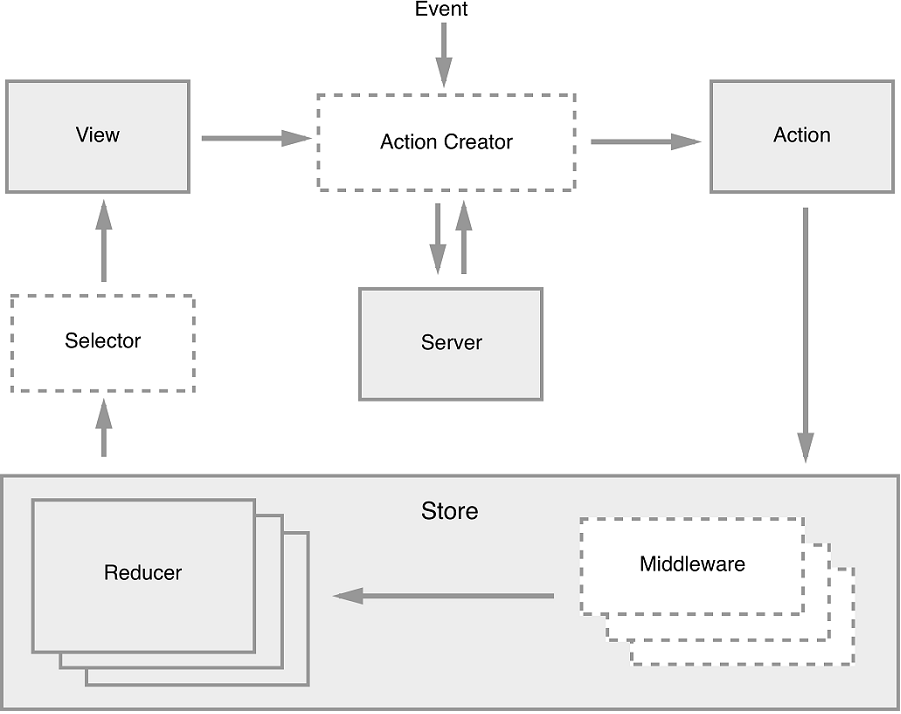
The reducer processed the action and computed the next state. Now it’s time for the store to update itself and broadcast the new state to all registered listeners (we care specifically about the components that make up our profile page).

##### Figure 1.7 The store now completes the loop by providing the new state to our profile page. Notice that the post count has incremented, and the new post has been added to the activity feed. If our user adds another post, we’d follow the same exact flow. The view would dispatch an action, reducers specify how to update state, and the store broadcasts the new state back to the view.



Now that you’re familiar with some of the most important building blocks, let’s take a look at a more comprehensive diagram of the Redux architecture. Some of pieces will be unfamiliar now, but we will revisit this diagram repeatedly throughout this book, and over time, we’ll fill in each of those gaps.

##### Figure 1.8 This diagram will serve to anchor our understanding of the elements of Redux as we move forward. So far, we’ve talked about actions, reducers, the store, and views.



To review, an interaction with a view may produce an action. That action will filter through one or more reducers and produce a new state tree within the store. Once the state has updated, the views will be made aware that there is new data to render. That’s the whole cycle! Items in this diagram with a dashed border (action creators, middleware, and selectors) are optional, but powerful tools in a Redux architecture. We will cover each of these topics in future chapters.

If this feels like a lot, don’t fret. If you’re new to the kind of one-directional architecture that we’re beginning to explore, it can be initially overwhelming (we certainly thought so at first). It takes time to let these concepts sink in. Developing a sense for what role they play and what type of code belongs where is as much art as it is science, and it’s a skill you’ll develop over time as you continue get your hands dirty.

## 1.4   Why Should I Use Redux?

By this point, you’ve been exposed to many of the Redux talking points. Just in case you have to pitch your boss on Redux by the time you finish the first chapter, let’s consolidate those ideas into a highlight reel. In short, Redux is a small, easy-to-learn state management library that results in a highly predictable, testable, and debuggable application.

### 1.4.1   Predictability

The biggest selling point for Redux is the sanity it provides to applications juggling complex state. The Redux architecture offers a straightforward way to conceptualize and manage state, one action at a time. Regardless of application size, actions within the unidirectional data flow result in predictable changes to a single store.

### 1.4.2   Developer Experience

Predictability enables some world-class debugging tools. Hot-loading and time-travel debugging provide developers with wildly faster development cycles, whether building new features or hunting down bugs. Your boss will like that you’re a happier developer, but she’ll love that you’re a faster one.

### 1.4.3   Testability

The Redux implementation code you’ll write is primarily functions, many of them pure. Each piece of the puzzle can be broken out and unit-tested in isolation with ease. Official documentation makes use of Jest and Enzyme, but whichever JavaScript testing libraries your organization prefers will do the trick.

### 1.4.4   Learning Curve

Redux is a pretty natural step up from vanilla React. The library has a remarkably small footprint, exposing only a handful of APIs to get the job done. You can become familiar with all of it in a day. Writing Redux code also requires your team to become familiar with some functional programming patterns. This will be new territory for some developers, but the concepts are straightforward. Once you understand that changes to state can be produced only by pure functions, you’re most of the way there.

### 1.4.5   Size

If your boss is doing her job, one of the items on her checklist is dependency size. Redux is a tiny library, just under 7KB when minified. Checkmate.

## 1.5   When Should I Use Redux?

There’s no question we’ve been hitting you over the head with how great Redux is, but it’s certainly no cure-all. We’ve argued in favor of why you should use Redux, but as we all know, nothing in life is free and no software pattern exists without tradeoffs.

The cost of Redux is a fair amount of boilerplate code and the added complexity of something more than React’s local component state. It’s important to realize that Redux, and the usage patterns you establish while using it, is one more thing for a new developer on your team to learn before they can contribute.

Redux co-creator Dan Abramov weighed in here, even publishing a blog post entitled “You Might Not Need Redux.” He recommends starting without Redux and introducing the library only after you’ve reached enough state management pain points to justify including it. The recommendation is intentionally vague, because that turning point will be slightly different for every team. Smaller applications without complex data requirements are the most common scenario where it might be more appropriate to not use Redux in favor of plain React.

What might those pain points look like? There are a few common scenarios that teams use to justify bringing in Redux. The first is the passing of data through several layers of components that don’t have any use for it. The second scenario deals with sharing and syncing data between unrelated parts of the application. We all have a tolerance for doing some of the above in React, but eventually there comes a breaking point.

Redux is likely a good fit out of the gate if you know you’ll want to build a specific feature that it excels at. If you know your application will have complex state and require undo and redo functionality, cut to the chase and pull in Redux. If server-side rendering is a requirement, consider Redux up front.

## 1.6   Alternatives to Redux

As mentioned already, Redux entered a crowded state-management market and more options have appeared since. Let’s run through the most popular alternatives for managing state in React applications.

### 1.6.1   Flux Implementations

While researching, we stopped counting Flux implementation libraries somewhere in the low 20s. Astoundingly, at least eight of them have received more than 1000 stars on GitHub. This highlights an important era in React’s history; the Flux architecture was a groundbreaking idea that spurred a lot of excitement in the community and, as a result, a great deal of experimentation and growth. During this period, libraries came and went at such an exhausting rate that the term JavaScript Fatigue was coined. With hindsight, it’s clear that each of those experiments was an important stepping stone along the way. Over time, many of the alternative Flux implementation maintainers have graciously bowed out of the race in favor of Redux or one of the other popular options, but there are still several well-maintained options out there.

#### Flux

Flux, of course, is the one that started it all. In the maintainers’ own words, “Flux is more of a pattern than a framework.” A lot of great documentation about the Flux architecture pattern lives in this repository, but a small API is exposed to facilitate building applications with the architecture. The Dispatcher is at the core of that API, and in fact, several other Flux implementations have incorporated that Dispatcher into their libraries. Measured in GitHub stars, this library is about half as popular as Redux and continues to be actively maintained by the Facebook team.

#### Reflux

Reflux was a fast follow to the original Flux library. The library introduces some functional reactive programming ideas to the Flux architecture by ripping out the single Dispatcher in favor of giving each action the ability to dispatch themselves. Callbacks can be registered with actions to update stores. Reflux is still maintained and about one sixth as popular as Redux, measured by GitHub stars.

#### Alt

Unlike Reflux, Alt stays true to the original Flux ideas and makes use of the Flux Dispatcher. Alt’s selling points are its adherence to the Flux architecture and a reduction in boilerplate code. Although it once enjoyed an enthusiastic community, at the time of writing, there have been no commits to the project in more than six months.

#### Honorable Mentions

To round out the bunch with greater than 1000 GitHub stars, we also have Fluxible, Fluxxor, NuclearJS, and Flummox. Fluxible continues to be well-maintained by the Yahoo team. Fluxxor, NuclearJS and Flummox may be maintained, but are no longer active. To underscore the idea that these projects were important stepping stones, Flummox was created by Andrew Clark, who went on to co-create Redux with Dan Abramov.

### 1.6.2   MobX

MobX offers a functional reactive solution to state management. Like Flux, MobX uses actions to modify state, but components react to that mutated, or observable, state. Although some of the terminology in functional reactive programming can be intimidating, the features are pretty approachable in practice. MobX also requires less boilerplate code than Redux, but does more for you under the hood and is therefore less explicit. The first commits for MobX predate those of Redux by only a couple of months, in early 2015.

### 1.6.3   GraphQL Clients

GraphQL is an exciting new technology, also being developed by the Facebook team. It’s a query language that allows you to specify and receive exactly the data that is required by a component. This paradigm fits well with the intended modularity of React components; any data fetching that is required by the component is encapsulated within it. Queries to the API are optimized for the data needs of parent and children components.

Typically, GraphQL is used with a GraphQL client. The two most popular clients today are Relay and Apollo Client. Relay is another project developed and maintained by the Facebook team (and open-source community). Apollo was originally implemented with Redux under the hood, but now offers additional configurability.

While it’s possible to bring in both Redux and a GraphQL client to manage the same application’s state, you may find the combination to be overly complex and unnecessary. Although GraphQL clients handle data fetching from a server and Redux is more general- purpose, there’s a lot of overlap in usage between the packages.

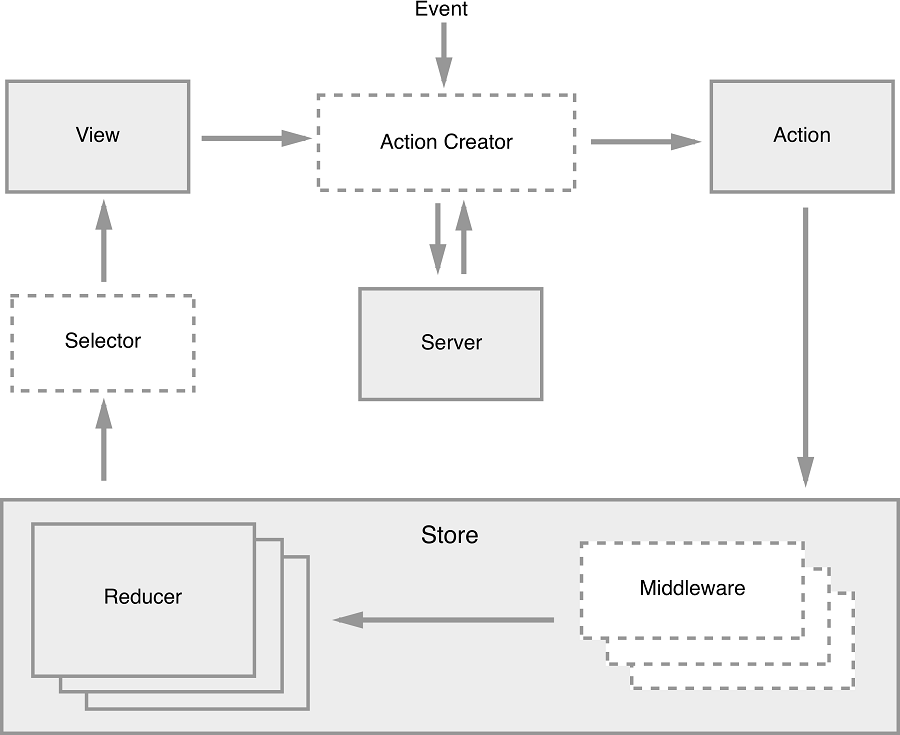
## 1.7   Summary

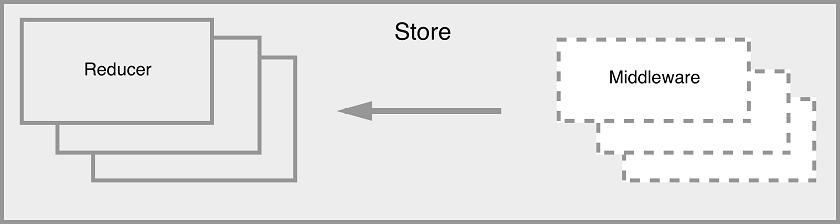
This chapter introduced the Flux architecture pattern and where Redux ran with those ideas. You learned several practical details about the library, including the following:

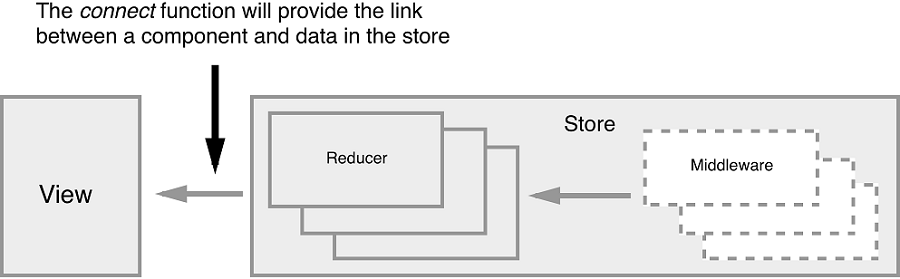
* Redux state is stored in a single object and is the product of pure functions.
* For the price of some boilerplate code, Redux can introduce predictability, testability, and debuggability to your complex application.
* If you’re experiencing pain points while syncing state across your application or passing data through multiple component layers, consider introducing Redux.

Now you’re ready to put the basic building blocks together and see a functioning Redux application, end to end. In the next chapter, we’ll build a task management application with React and Redux.

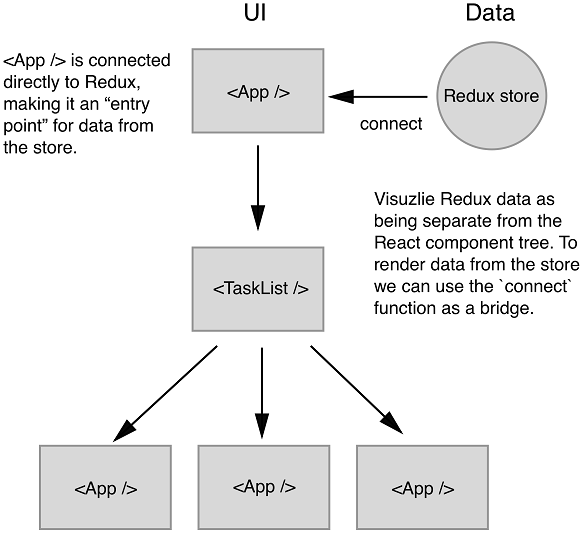
##### Figure 2.3 The Redux architecture.







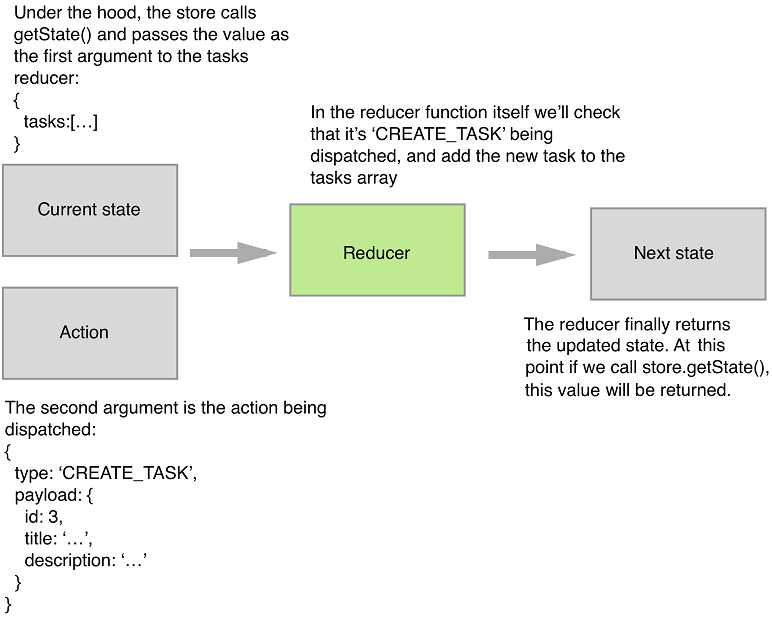
##### Figure 2.6 A visualization of how React and Redux work together.



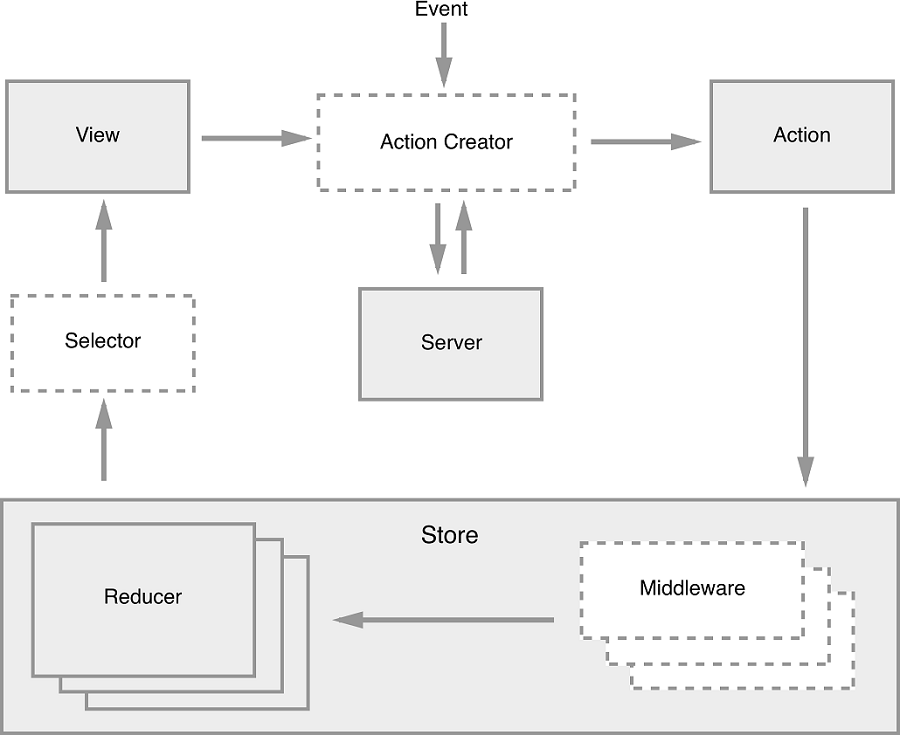
##### Figure 2.9 Although views can dispatch actions, they will often invoke action creators instead — functions that return actions.

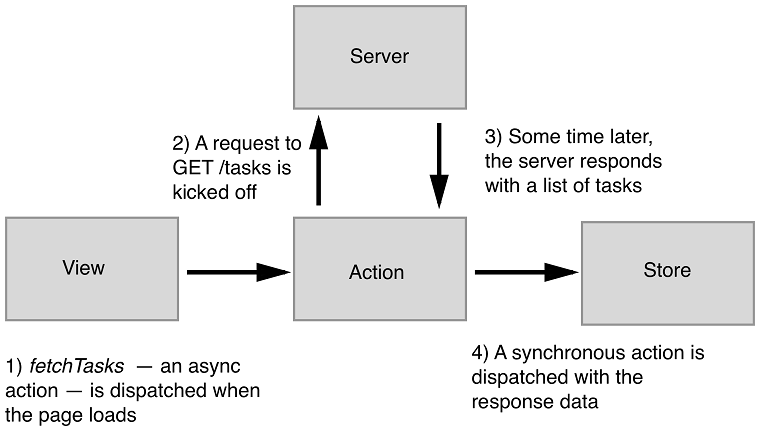


##### Figure 2.10 A look at our reducer in action. It takes two arguments, the current state of the store and the CREATE\_TASK action, and returns the next state.

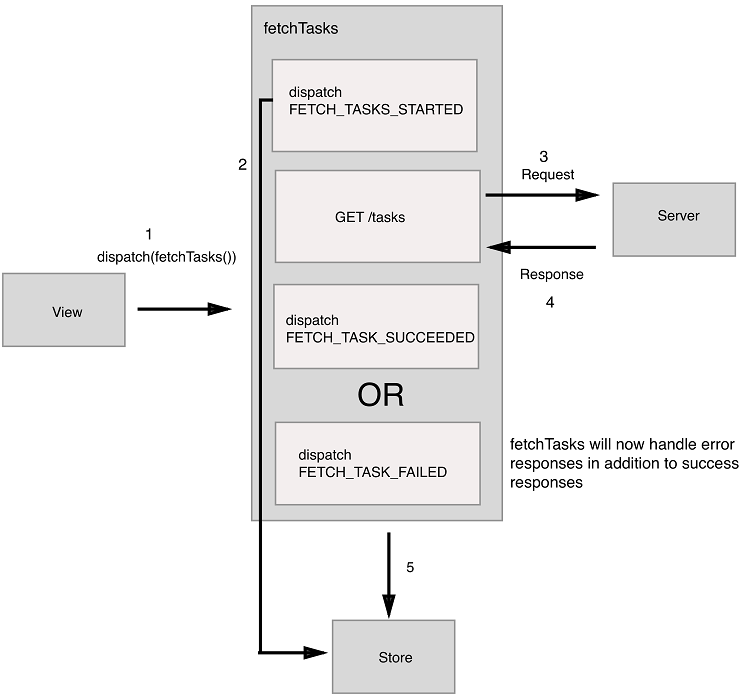


##### Figure 2.11 The Redux architecture





export function fetchTasks() {  return dispatch => {    api.fetchTasks().then(resp => {      dispatch(fetchTasksSucceeded(resp.data));    });  };}



# React Redux --- Concept, Workflow & Cheat Sheet

Ulrich Anders

Version 1.2.0, September 2017

## Preface

This article contains a graphical cheat sheet for the workflow and concept of Redux. In line with this cheat sheet the purpose of this article is to help new and existing Redux users to understand how the general mechanism of Redux works in detail and where to hook into this process.

This article is not an easy read. It is really for understanding the workflow of React Redux in detail. I recommend that you look at the graphical cheatsheet in parallel (print out, split screen, second monitor) while reading the article to always keep the overview.

I am trying to keep the article current as much as possible, however, the most recent version will always be on [GitHub](https://github.com/uanders/react-redux-cheatsheet)together with a Changelog.

## Introduction

[React](https://facebook.github.io/react/) and [Redux](http://redux.js.org/) 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.^[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.]

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.^[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.] 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.](http://academy.plot.ly/react/1-introduction/)
2. [Leveling Up with React: Redux. Post by Brad Westfall](https://css-tricks.com/learning-react-redux/)
3. [Three Rules For Structuring (Redux) Applications. Series of posts by Jack Hsu.](https://jaysoo.ca/2016/02/28/organizing-redux-application/)
4. [Getting Started with Redux. Video series by Dan Abramov.](https://egghead.io/courses/getting-started-with-redux)
5. [Building React Applications with Idiomatic Redux. Video series by Dan Abramov.](https://egghead.io/courses/building-react-applications-with-idiomatic-redux)
6. [Learn Redux. Video series by Wes Bos.](https://learnredux.com/)
7. [Learn React and Redux. Video series by Catalin Luntraru.](https://www.youtube.com/watch?v=d0oUGmSE6IY&list=PLJBrYU54JD2pTblB20OmV7GL6H5J-p2g8)

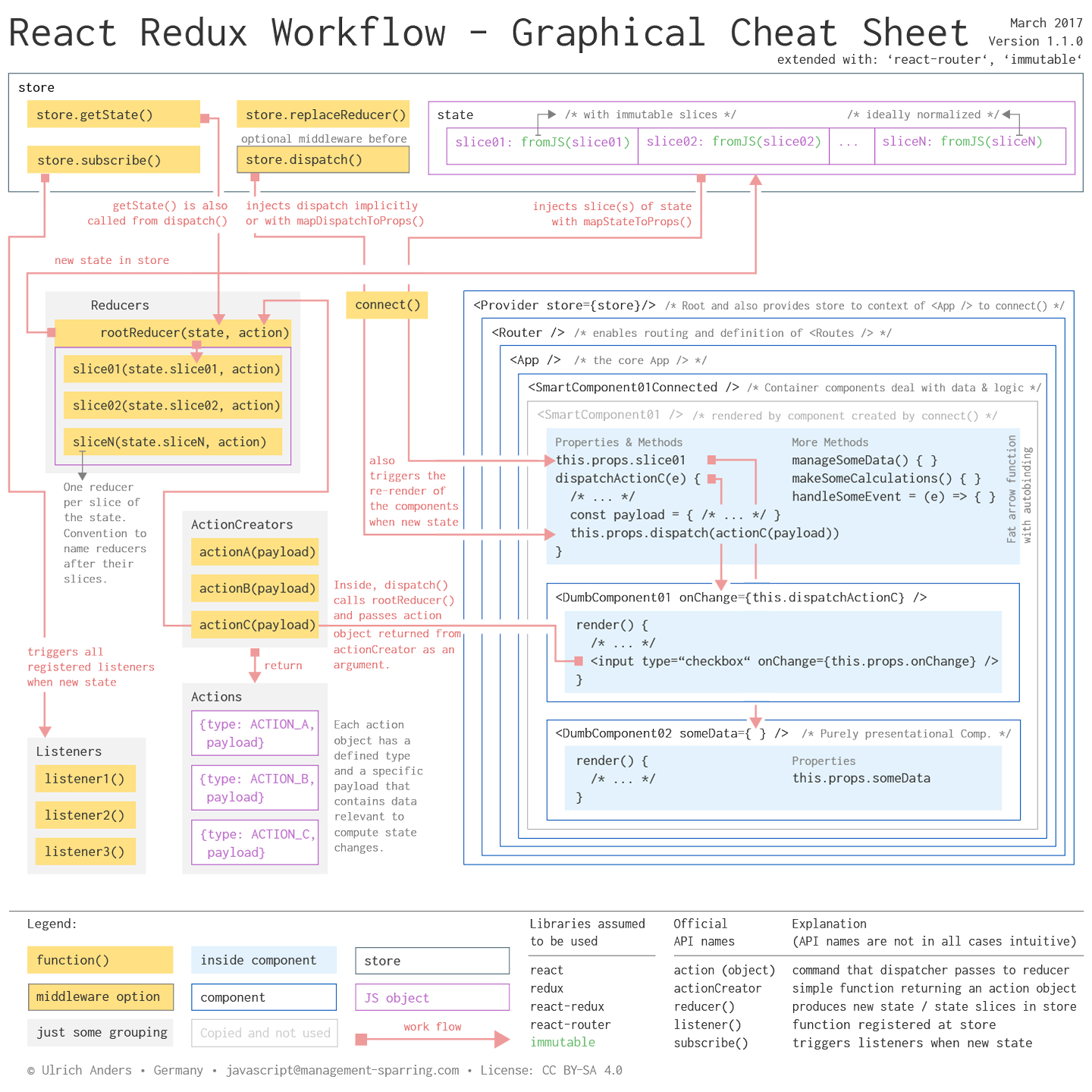
Most tutorials approach the topic of Redux by building an app. While this is done the tutorials introduce the important concepts and workflow around Redux step by step. The general challenge with this approach is that two types of information are competing with each other: the conceptual overview and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

## 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.

[](https://github.com/uanders/react-redux-cheatsheet/blob/master/1440/react-redux-workflow-graphical-cheat-sheet-extended_v110.png)

## 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](http://redux.js.org/docs/basics/Store.html). 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:
5. // this uses ES6 shorthand syntax
6. state = {
7. slice01,
8. slice02,
9. /\* ... \*/
10. sliceN

}

1. 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](https://github.com/paularmstrong/normalizr).
2. // bad:
3. "users": {
4. {"id": "1", "name": "Adam"},
5. {"id": "2", "name": "Eve"},
6. }
7. //good:
8. "users": {
9. "1": { "id": "1", "name": "Adam" },
10. "2": { "id": "2", "name": "Eve" }

},

1. 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.
2. To help manage the immutability and also make the updates fast it is sensible to use for instance [immutable.js](https://facebook.github.io/immutable-js/docs/#/). This library provides a function [fromJS()](https://facebook.github.io/immutable-js/docs/#/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.
3. import { fromJS } from 'immutable'
4. state = {
5. slice01: fromJS(slice01)
6. slice02: fromJS(slice02),
7. /\* ... \*/
8. sliceN: fromJS(SliceN)

}

1. There are some pros and cons using immutable.js which you can find in the [Redux recipes for immutable.js](http://redux.js.org/docs/recipes/UsingImmutableJS.html). 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.
2. 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()](http://redux.js.org/docs/basics/Reducers.html). The name reducer is semantically not very meaningful and has a purely technical origin.^[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)."] 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.
3. It is a [popular convention](http://redux.js.org/docs/api/combineReducers.html) (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.slice01would be called slice01().
4. 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()](http://redux.js.org/docs/api/combineReducers.html) that takes an object containing the individal reducers:
5. import { combineReducers } from 'redux'
6. const rootReducer = function combineReducers({
7. slice01,
8. slice02,
9. /\* ... \*/
10. sliceN

})

1. 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().

## 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](https://facebook.github.io/react-native/docs/props.html).
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.
4. const Root = ( {store} ) => (
5. <App store={store} />

)

1. 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.
2. There are two types of components: [smart (container) components and dumb (presentational) components](https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0#.3upgdk21p). Smart components have state, dumb components don't. Of course, there are 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.
3. 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.
4. class SmartComponent01 extends Component {
5. manageSomeData () {
6. /\* ... \*/
7. }
8. makeSomeCalculations () {
9. /\* ... \*/
10. }
11. handleSomeEvent = (event) => {
12. /\* ... \*/
13. }
14. render() {
15. return (
16. <div>
17. <DumbComponent01 data={/\*...\*/} />
18. <DumbComponent02 func={/\*...\*/} />
19. </div>
20. )
21. }

}

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](https://facebook.github.io/react/docs/react-without-es6.html#autobinding) the methods to the [this](http://exploringjs.com/es6/ch_arrow-functions.html#sec_traditional-functions-bad-non-methods) operator of the classes. Here, you would need to bind manageSomeData() and makeSomeCalculations() manually to the this operator, whereas handleSomeEvent() would have been autobound.

1. Since only smart container components manage data only they need to receive state data from the store. This works via a function called [connect()](https://github.com/reactjs/react-redux/blob/master/docs/api.md#connectmapstatetoprops-mapdispatchtoprops-mergeprops-options) 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.
2. import {connect} from 'react-redux'
3. /\* ... \*/
4. const ConnectedSmartComponent01 = connect()(SmartComponent01)
5. export default ConnectedSmartComponent01
6. 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.
7. Thus, the mapping from a state slice to the properties of a smart component is done with a function that is conventionally named [mapStateToProps()](http://redux.js.org/docs/basics/UsageWithReact.html). 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.
8. import {connect} from 'react-redux'
9. function mapStateToProps(state) {
10. return {
11. // component gets this.props.slice01
12. slice01: state.slice01
13. }
14. }
15. // export without a new name

export default connect(mapStateToProps)(SmartComponent01)

1. 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.
2. 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'](https://github.com/reactjs/reselect).
3. function mapStateToProps(state) {
4. return {
5. // component gets this.props.selection1
6. selection1: getSelection1(state.slice01),
7. // component gets this.props.selection2
8. selection2: getSelection2(state.slice01)
9. }

}

1. 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](https://github.com/reactjs/react-redux/blob/master/docs/api.md#provider-store) 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](https://facebook.github.io/react/docs/context.html) 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:
2. import { Provider } from 'react-redux'
3. const Root = ( {store} ) => (
4. <Provider store={store}>
5. <App />
6. </Provider>

)

## 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()](http://redux.js.org/docs/api/Store.html#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

1. If you do not want the connect() function to just inject the standard dispatch() function and map it to this.props.dispatchyou 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.
2. function mapDispatchToProps(dispatch) {
3. return {
4. /\* your own bindings for the dispatch() function \*/
5. }
6. }
7. export default connect(mapStateToProps,

mapDispatchToProps)(SmartComponent01)

1. 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](http://redux.js.org/docs/basics/Actions.html) 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.
2. const DO\_SOMETHING = 'DO\_SOMETHING'
3. // an action object
4. {
5. type: DO\_SOMETHING,
6. payload: relevantData

}

1. 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](https://github.com/acdlite/flux-standard-action) to have the same API for all action objects.
2. // relevant data
3. let payload =
4. {
5. id,
6. something

}

1. 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()](http://redux.js.org/docs/basics/Actions.html#action-creators). 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.
2. // this simple function is called an actionCreator
3. function do\_something(payload) {
4. return (
5. {
6. type: DO\_SOMETHING,
7. payload
8. }
9. )

}

1. 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.
2. 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.
3. 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.^[Note, other than the [Redux Documentation](http://redux.js.org/docs/basics/Reducers.html#handling-actions) reads (previousState, action) => newState a reducer does not receive the whole state by default but only its slice of the state.] 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.
4. // reducer for state.slice01
5. // using setIn() from immutable.js
6. function slice01(slice01 = {}, action) {
7. switch (action.type) {
8. case DO\_SOMETHING:
9. return slice01.setIn(['somewhere', action.payload.id],
10. action.payload.something)
11. default:
12. return slice01
13. }

}

1. 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.
2. 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.
3. 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()](http://redux.js.org/docs/api/Store.html#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().^[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.]

store.subscribe(listener)

1. This completes the cycle.

## 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()](http://redux.js.org/docs/api/createStore.html). 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:
2. import { createStore } from 'redux'

let store = createStore(rootReducer, initialState)

1. 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'](https://github.com/rt2zz/redux-persist). This library also helps you with immutable transformations and with encrypting your data in the local storage.
2. 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](http://redux.js.org/docs/advanced/Middleware.html). 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()](http://redux.js.org/docs/api/applyMiddleware.html). 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.
3. import { createStore, applyMiddleware } from 'redux'
4. let store = createStore(
5. rootReducer,
6. applyMiddleware(logger, timeoutScheduler)

)

1. 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()](http://redux.js.org/docs/api/compose.html) utility function.
2. let store = createStore(
3. rootReducer,
4. compose(
5. applyMiddleware(logger, timeoutScheduler),
6. middlewareFromAnExtension()
7. )

)

1. 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'](https://github.com/gaearon/redux-thunk), ['redux-promise'](https://www.npmjs.com/package/redux-promise) or ['redux-saga'](https://github.com/redux-saga/redux-saga). They all come as middleware for the dispatch() function and help you deal with such cases.
2. 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'](https://github.com/ReactTraining/react-router). I am referring to ≥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.
3. import { Provider } from 'react-redux'
4. import { BrowserRouter as Router } from 'react-router-dom'
5. const Root = ( {store} ) => (
6. <Provider store={store}>
7. <Router>
8. <App />
9. </Router>
10. </Provider>

)

1. In your App component you can then use linking and routing by help of NavLink and Route components:
2. import { Provider } from 'react-redux'
3. import { NavLink, Route } from 'react-router-dom'
4. const App = () => (
5. <div>
6. <NavLink to="/">Home</NavLink>
7. <NavLink to="/about">About</NavLink>
8. <NavLink to="/users">Users</NavLink>
9. <Route exact path="/" component={Home} />
10. <Route path="/about" component={About} />
11. <Route path="/users" component={Users} />
12. </div>

)

## 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.

[Deals](https://www.sitepoint.com/premium/deals/?ref_source=sitepoint&ref_medium=topnav)

**[12](https://www.sitepoint.com/premium/deals/?ref_source=sitepoint&ref_medium=topnav)**

[Learning Paths](https://www.sitepoint.com/premium/paths?ref_source=sitepoint&ref_medium=topnav)[Books & Courses](https://www.sitepoint.com/premium/topics/all?q=&limit=400&offset=0&page1&ref_source=sitepoint&ref_medium=topnav)

[**Login**](https://www.sitepoint.com/premium/sign-in)[**Sign Up**](https://www.sitepoint.com/premium/l/join?ref_source=sitepoint&ref_medium=topnav)

**OFFER ENDS IN**

**02**

DAYS

**05**

HRS

**53**

MINS

**56**

SECS

[**Get All Our Books And Courses For $9**](https://www.sitepoint.com/premium/l/join?ref_source=sitepoint&ref_medium=noticebar&ref_campaign=getting-started-redux&ref_content=javascript)

[**JavaScript**](https://www.sitepoint.com/javascript/) - September 22, 2017 - By [Vasu K](https://www.sitepoint.com/author/vasudevank/)

# To Redux or Not: the Art of Structuring State in React Apps

**Related Topics:**

[npm](https://www.sitepoint.com/javascript/npm/)

[Node.js](https://www.sitepoint.com/javascript/nodejs-javascript/)

[React](https://www.sitepoint.com/javascript/react/)

[AngularJS](https://www.sitepoint.com/javascript/angularjs/)

[Ajax](https://www.sitepoint.com/javascript/ajax-javascript/)

[More...](https://www.sitepoint.com/javascript/)

[[SitePoint Javascript - Proudly supported by Wix Code, Creation without limits](https://www.wix.com/eteamhtml/wixcode5?utm_campaign=ma_sitepoint.com_wixcode&experiment_id=sitepoint.com%5ebar%5esitepoint.com_barsponsorship) Build websites with built-in DB, backend, IDE & one-click deploys](https://www.wix.com/eteamhtml/wixcode5?utm_campaign=ma_sitepoint.com_wixcode&experiment_id=sitepoint.com%5ebar%5esitepoint.com_barsponsorship" \o "WixCode)

For a high-quality, in-depth introduction to React, you can’t go past Canadian full-stack developer Wes Bos. [Try his course here](https://reactforbeginners.com/friend/SITEPOINT), and use the code**SITEPOINT** to get **25% off**and to help support SitePoint.

**One common trend I find among most Redux developers is a hatred towards**setState()**. A lot of us (yes, I’ve fallen into this trap many times before) flinch at the sight of**setState()**and try to keep all the data in our Redux store. But, as the complexity of your application grows, this poses several challenges.**

In this post, I’ll walk you through various strategies to model your state, and dive into when each of them can be used.

## Getting Started

Redux works with the principle of being the single source of truth for your application state. A new Game of Thrones season is airing now, and I’m sure everyone’s excited to know how this is going to unfold. Let’s build a fun Game of Thrones fan listing page, to understand these concepts in detail.

Note: I’ll be using *yarn* to run the app. If you don’t have [*yarn*](https://www.sitepoint.com/yarn-vs-npm/) set up, replace yarn with *npm*.

Before we dive in, download the basic skeleton from [the repo](https://github.com/skmvasu/redux-state-vs-local-state-sp) and run:

yarn install

yarn run start

You should see a basic list page with some of your favorite GoT characters listed.

Note: We’ll be using the [*ducks pattern*](https://github.com/erikras/ducks-modular-redux) to write our application. It reduces unnecessary module imports and cuts down on a lot of boilerplate.

## Intro to Redux

The scope of this article is to help you structure your Redux apps. It assumes a basic knowledge of the library. I’ll give a brief overview of Redux concepts that will help you follow the rest of the article better. If you’re familiar with how these works, feel free to skip this section.

All Redux apps make use of four important constructs: actions, reducers, a store, and containers.

### Actions

An **action** is an intent to update the state. It could be triggered by a network call, or a user clicking a button. Actions have two parts:

1. **Action type**. A unique identifier representing an action.
2. **Payload**. Any metadata that’s associated with the action. For instance, if we make a network request to fetch a list of movies, the response from the server is the payload.

For this example, we’ll be using a library called redux-actions to create actions.

### Reducers

A **reducer** is a function that listens for an action and returns a new state representation.

### Store

An application can be divided into many reducers, representing various parts of the page. A **store** brings all these together and keeps the app state intact.

### Containers

**Containers** connect your app state and actions with the component, passing them down as props.

To get a deep understanding of how this works, I’d encourage you to first look at the [free introduction series by Dan Abramov](https://egghead.io/courses/getting-started-with-redux).

## Split App Data and UI State

The list page is nice, but the names don’t give any context to people who are new to the GoT universe. Let’s extend the component to render the character description as well:

//GoTCharacter.js

export const CharacterRow = ({character}) => (

<div className="row">

<div className="name">{character.name}</div>

<div className="description">{character.description}</div>

</div>

);

While this solves the problem, our designers feel that the page looks clumsy, and it’s a better idea to collapse this information till users want it. There are three different approaches we can take to solve this problem.

### The **setState** approach

The simplest way to achieve this in React is using setState() to store the data within the component itself:

//GoTCharacter.js

export class StatefulCharacterRow extends Component {

constructor() {

super();

this.state = {

show\_description: false

}

}

render() {

const {character} = this.props;

return (<div className="row">

<div className="name">{character.name}</div>

<a href="#" onClick={() => this.setState({

show\_description: !this.state.show\_description})} >

{this.state.show\_description ? 'collapse' : 'expand'}

</a>

{this.state.show\_description &&

<div className="description">{character.description}</div>}

</div>);

}

};

### The Redux approach

Using setState() is fine as long as the state we’re dealing with is only local to the component. If, for instance, we want to put in place an “expand all” function, it will be difficult to handle this with just React.

Let’s see how we can move this to Redux:

// FlickDuck.js

// …

export const toggleCharacterDescription = createAction(

FlixActions.TOGGLE\_CHARACTER\_DESCRIPTION, (character) => ({character})

);

export default (current\_state, action) => {

const state = current\_state || default\_state;

switch (action.type) {

case FlixActions.TOGGLE\_CHARACTER\_DESCRIPTION:

return {...state, characters: state.characters.map(char => {

if (char.id === action.payload.character.id) {

return {...char,show\_description: !char.show\_description};

}

return char;

})}

default:

return state

}

}

// GoTCharactersContainer.js

import { connect } from 'react-redux';

import GoTCharacters from './GoTCharacters';

import {toggleCharacterDescription} from './FlickDuck';

const mapStateToProps = (state) => ({

...state.flick

});

const mapDispatchToProps = (dispatch) => ({

toggleCharacterDescription : (data) => dispatch(toggleCharacterDescription(data))

});

export default connect(mapStateToProps, mapDispatchToProps)(GoTCharacters);

// GoTCharacters.js

const GoTCharacters = ({characters,toggleCharacterDescription}) => {

return (

<div className="characters-list">

{characters.map(char => (

<CharacterRow

character={char}

toggleCharacterDescription={toggleCharacterDescription}

key={char.id}/>

))}

</div>

);

};

export const CharacterRow = ({character, toggleCharacterDescription}) => (

<div className="row">

<div className="name">{character.name}</div>

<a href="#" onClick={toggleCharacterDescription.bind(null, character)} >

{character.show\_description ? 'collapse' : 'expand'}

</a>

{character.show\_description &&

<div className="description">{character.description}</div>}

</div>

);

We’re storing the state of the description field inside the character object. Our state will look like this now:

state = {

characters: [{

id: 1,

name: "Eddard Ned Stark",

house: "stark",

description: "Lord of Winterfell - Warden of the North - Hand of the King - Married to Catelyn (Tully) Stark",

imageSuffix: "eddard-stark",

wikiSuffix: "Eddard\_Stark",

show\_description: true

},

{

id: 2,

name: "Benjen Stark",

house: "stark",

description: "Brother of Eddard Stark - First ranger of the Night's Watch",

imageSuffix: "benjen-stark",

wikiSuffix: "Benjen\_Stark",

show\_description: false

}]

}

This is a general pattern a lot of developers follow when they’re starting out with Redux. There’s nothing wrong with this approach, and it works great for smaller apps.

So far, we’ve been dealing with the characters from the first chapter of GoT, and the universe is about to get a whole lot bigger. When it does, our app will become slow. Imagine looping through 1000 characters to update one row.

Let’s see how to scale this for a larger dataset:

// FlickDuck.js

// …

case FlixActions.TOGGLE\_CHARACTER\_DESCRIPTION:

const {character} = action.payload;

return {

...state,

character\_show\_description: {

...state.character\_show\_description,

[character.id]: !state.character\_show\_description[character.id]

}

}

// …

And in GoTCharacters.js:

export const CharacterRow = ({character, character\_show\_description, toggleCharacterDescription}) => (

<div className="row">

<div className="name">{character.name}</div>

<a href="#" onClick={toggleCharacterDescription.bind(null, character)} >

{character\_show\_description[character.id] ? 'collapse' : 'expand'}

</a>

{character\_show\_description[character.id] &&

<div className="description">{character.description}</div>}

</div>

);

When the user clicks on the **expand** link, we update the character\_show\_description with the current character id. The state looks like this now:

state = {

characters: [...],

character\_show\_description: {

1: true,

2: false

}

}

Now we can update the UI state without looping over all the characters.

## Managing Form State in Redux

Managing form state is a tricky business. In a typical application, we’ll serialize the form data once during submit and, if it’s valid, submit it. Otherwise, we’ll show an error message. Easy-peasy, right?

But, in the real world, we’ll have some complex interactions involving forms. When there is a validation error on a form, we may have to show the errors at top of the page. We may even need to disable some elements in the other part of the page, depending on the UX. This is usually achieved by passing random callbacks from your parents’ parents’ parent, or even manipulating the DOM with every validation.

Let’s see how we can implement this with Redux:

// FlickDuck.js

// ============

const FlixActions = km({

FETCH\_CHARACTERS: null,

TOGGLE\_CHARACTER\_DESCRIPTION: null,

TOGGLE\_CHARACTER\_EDIT: null,

SYNC\_CHARACTER\_EDIT\_DATA: null,

SAVE\_CHARACTER\_EDIT: null

});

const default\_state = {

characters: characters,

character\_show\_description: {},

show\_character\_edit: {},

character\_edit\_form\_data: {}

};

export const toggleEdit = createAction(

FlixActions.TOGGLE\_CHARACTER\_EDIT, (character) => ({character})

);

export const syncCharacterEditData = createAction(

FlixActions.SYNC\_CHARACTER\_EDIT\_DATA, (character, form\_data) => ({character, form\_data})

);

export const editCharacterDetails = createAction(

FlixActions.SAVE\_CHARACTER\_EDIT, (character) => ({character})

);

export default (current\_state, action) => {

// …

switch (action.type) {

// …

case FlixActions.TOGGLE\_CHARACTER\_EDIT:

character = action.payload.character;

const show\_character\_edit = !state.show\_character\_edit[character.id];

return {

...state,

show\_character\_edit: {

...state.show\_character\_edit,

[character.id]: show\_character\_edit

}, character\_edit\_form\_data : {

...state.character\_edit\_form\_data,

[character.id]: show\_character\_edit ? {...character} : {}

}

}

case FlixActions.SYNC\_CHARACTER\_EDIT\_DATA:

character = action.payload.character;

const {form\_data} = action.payload;

return {

...state,

character\_edit\_form\_data: {

...state.character\_edit\_form\_data,

[character.id]: {...form\_data}

}

}

case FlixActions.SAVE\_CHARACTER\_EDIT:

character = action.payload.character;

const edit\_form\_data = state.character\_edit\_form\_data[character.id];

const characters = state.characters.map(char => {

if (char.id === character.id) return {...char, name:edit\_form\_data.name, description: edit\_form\_data.description}

return char;

});

return {

...state,

characters,

show\_character\_edit: {

...state.show\_character\_edit,

[character.id]: false

}

}

// …

}

}

// GotCharacters.js

export const CharacterRow = ({character, character\_show\_description, character\_edit\_form\_data, show\_character\_edit, toggleCharacterDescription, toggleEdit, syncCharacterEditData, editCharacterDetails}) => {

const toggleEditPartial = toggleEdit.bind(null, character);

return (<div className="row">

<div className="name">{character.name}</div>

<a href="#" onClick={toggleCharacterDescription.bind(null, character)} >

{character\_show\_description[character.id] ? 'collapse' : 'expand'}

</a>

{!character\_show\_description[character.id] && <a href="#" onClick={toggleEditPartial} >

edit

</a>}

{character\_show\_description[character.id] &&

<div className="description">{character.description}</div>}

{show\_character\_edit[character.id] &&

<EditCharacterDetails character={character}

cancelEdit={toggleEditPartial}

syncCharacterEditData={syncCharacterEditData}

editCharacterDetails={editCharacterDetails}

edit\_data={character\_edit\_form\_data[character.id]}/>

}

</div>);

}

export const EditCharacterDetails = ({character, edit\_data, syncCharacterEditData, editCharacterDetails, cancelEdit}) => {

const syncFormData = (key, e) => {

const {value} = e.currentTarget;

syncCharacterEditData(character, {

...edit\_data,

[key]: value

});

};

const saveForm = (e) => {

e.preventDefault();

editCharacterDetails(character);

};

return (

<form onSubmit={saveForm}>

<label>Name: </label>

<input name='name' value={edit\_data.name} onChange={syncFormData.bind(null, 'name')}/>

<label>Description:</label>

<textarea name='description' value={edit\_data.description} onChange={syncFormData.bind(null, 'description')}/>

<button type="reset" onClick={cancelEdit}> Cancel </button>

<button type="submit"> Submit </button>

</form>

);

};

Let’s extend this to handle validations:

// FlickDuck.js

// ============

export const editCharacterDetails = createAction(

FlixActions.VALIDATE\_AND\_SAVE\_CHARACTER\_EDIT, (dispatch, character, edit\_form\_data) => {

const errors = validateCharacterForm(edit\_form\_data);

if (Object.keys(errors).length) {

return dispatch(showErrorMessage(character, errors));

}

return dispatch(saveCharacterEdit(character));

}

);

export const showErrorMessage = createAction(

FlixActions.VALIDATE\_CHARACTER\_EDIT, (character, errors) => ({character, errors, hasError: true})

);

export const saveCharacterEdit = createAction(

FlixActions.SAVE\_CHARACTER\_EDIT, (character) => ({character})

);

switch (action.type) {

// …

case FlixActions.VALIDATE\_CHARACTER\_EDIT:

character = action.payload.character;

const {errors, hasError} = action.payload;

return {

...state,

character\_edit\_form\_errors: {

...state.character\_edit\_form\_errors,

[character.id]: {errors, hasError}

}

}

// …

}

Isn’t this very similar to the example we saw in the previous section? What’s so special about it in forms?

Before jumping into this, it’s important to understand how Redux internals work. When your state changes, you don’t update a single point in the tree. Instead, the entire state tree is replaced by a new one. This tree is passed to your React component and React reconciles all the components to see if the DOM needs updating.

Form state is special, because the state tree changes very quickly. Depending on the users’ typing speed, that could be a problem. Since state changes trigger reconciliation of all nodes, there might be a small lag when users type. It can become very noticeable when dealing with a big page with a few hundred components.

Let’s see how we can remodel this without making a big change:

export class StatefulCharacterRow extends Component {

constructor() {

super();

this.toggleEditForm = this.toggleEditForm.bind(this);

this.syncCharacterEditData = this.syncCharacterEditData.bind(this);

this.state = {

show\_description: false,

show\_edit\_form: false,

edit\_data: {}

}

}

toggleEditForm() {

const {name, description} = this.props.character;

const show\_edit\_form = !this.state.show\_edit\_form;

const edit\_data = show\_edit\_form ? {name, description} : {};

this.setState({show\_edit\_form, edit\_data});

}

syncCharacterEditData(character, form\_data) {

this.setState({

edit\_data: {...this.state.edit\_data, ...form\_data}

});

}

render() {

const {character} = this.props;

return (<div className="row">

<div className="name">{character.name}</div>

<a href="#" onClick={() => this.setState({

show\_description: !this.state.show\_description})} >

{this.state.show\_description ? 'collapse' : 'expand'}

</a>

{!this.state.show\_edit\_form && <a href="#" onClick={this.toggleEditForm} >

edit

</a>}

{this.state.show\_description &&

<div className="description">{character.description}</div>}

{this.state.show\_edit\_form &&

<EditCharacterDetails character={character}

cancelEdit={this.toggleEditForm}

syncCharacterEditData={this.syncCharacterEditData}

editCharacterDetails={this.props.editCharacterDetails}

edit\_data={this.state.edit\_data}/> }

</div>);

}

};

The simplest way to handle this is to create a wrapper component around your form (think of it like a container) and store the state there. So, when users input changes, only this node gets updated without shaking the entire tree.

Notice that we’ve only moved the form state inside React, but the error state still remains outside. This will help reduce unnecessary clutter if we ever want to handle these errors outside of the form scope.

### Recommended Courses



[**The Best Way to Learn React for Beginners**](https://reactforbeginners.com/friend/SITEPOINT)

Wes Bos

A step-by-step training course to get you building real world React.js + Firebase apps and website components in a couple of afternoons. Use coupon code **'SITEPOINT'** at checkout to get **25% off**.

## Wrapping Up

Before deciding on where to store state when using Redux, it would be helpful to understand the following scenarios:

### 1. Is this UI state, or application state?

Character name is application state, whereas tracking whether an action is in progress is UI state. While it’s tempting to couple them, in the long run it’ll pay off to keep them separate.

state = {

characters: [{

id: 1,

name: Jon Snow,

…

}],

ui\_state: {

1: {

is\_edit\_in\_progress: true,

show\_description: false

}

}

}

### 2. How to decide what goes in component state and what goes in Redux

Usually, app data can be rendered many times on a page. For instance, we can render a list of all characters and show a count of characters grouped by the house they belong to. It makes sense to manage them in Redux.

Store UI state in Redux if there is a global dependency. Otherwise, you’re better off handling it with React’s local component state.

Redux has helped me structure my thoughts better. With jQuery/Backbone, my focus was around how to manipulate the DOM to achieve the intended effect. With Redux, it’s about getting your application state right. Once you nail that, the complexity of your front-end codebase comes down significantly.

[](https://www.sitepoint.com/author/vasudevank/)

Meet the author

[**Vasu K**](https://www.sitepoint.com/author/vasudevank/)

Hola! I'm a Fullstack developer and a strong advocate of Mobile first design. I'm running a digital children's startup for kids and I lead the engineering efforts there. In my free time I ramble about technology, and consult startups.

SPONSORS

[[](https://srv.buysellads.com/ads/click/x/GTND42QJF6SDC2J7CTYLYKQMC6AIP2QMCWSDVZ3JCWYIL23NC6SDP2JKC6BI45QNCVBDEK3EHJNCLSIZ)](https://srv.buysellads.com/ads/click/x/GTND42QJF6SDC2J7CTYLYKQMC6AIP2QMCWSDVZ3JCWYIL23NC6SDP2JKC6BI45QNCVBDEK3EHJNCLSIZ)

**[HelloSign](https://srv.buysellads.com/ads/click/x/GTND42QJF6SDC2J7CTYLYKQMC6AIP2QMCWSDVZ3JCWYIL23NC6SDP2JKC6BI45QNCVBDEK3EHJNCLSIZ)**[HelloSign API: Everything IT requires and Developers love.](https://srv.buysellads.com/ads/click/x/GTND42QJF6SDC2J7CTYLYKQMC6AIP2QMCWSDVZ3JCWYIL23NC6SDP2JKC6BI45QNCVBDEK3EHJNCLSIZ)

[[](https://srv.buysellads.com/ads/click/x/GTND42QJF6SDC2J7CTYLYKQMC6AIP2QICYAILZ3JCWYIL23NC6YDV5QKC6BI45QNCVBDEK3EHJNCLSIZ)](https://srv.buysellads.com/ads/click/x/GTND42QJF6SDC2J7CTYLYKQMC6AIP2QICYAILZ3JCWYIL23NC6YDV5QKC6BI45QNCVBDEK3EHJNCLSIZ)

**[MailChimp](https://srv.buysellads.com/ads/click/x/GTND42QJF6SDC2J7CTYLYKQMC6AIP2QICYAILZ3JCWYIL23NC6YDV5QKC6BI45QNCVBDEK3EHJNCLSIZ)**[The brains behind your business just got another brain.](https://srv.buysellads.com/ads/click/x/GTND42QJF6SDC2J7CTYLYKQMC6AIP2QICYAILZ3JCWYIL23NC6YDV5QKC6BI45QNCVBDEK3EHJNCLSIZ)

* **Stuff We Do**
* [Premium](https://www.sitepoint.com/premium/)
* [Versioning](https://www.sitepoint.com/versioning/)
* [Themes](https://www.sitepoint.com/themes/)
* [Forums](https://www.sitepoint.com/community/)
* [References](https://www.sitepoint.com/html-css/css/)
* **About**
* [Our Story](https://www.sitepoint.com/about-us/)
* [Press Room](https://www.sitepoint.com/press/)
* **Contact**
* [Contact Us](https://www.sitepoint.com/contact-us/)
* [FAQ](https://sitepoint.zendesk.com/hc/en-us)
* [Write for Us](https://www.sitepoint.com/write-for-us/)
* [Advertise](https://www.sitepoint.com/advertise/)
* **Legals**
* [Terms of Use](https://www.sitepoint.com/legals/)
* [Privacy Policy](https://www.sitepoint.com/legals/#privacy)
* **Connect**

© 2000 – 2018 SitePoint Pty. Ltd.

Focus Retriever

For a high-quality, in-depth introduction to React, you can’t go past Canadian full-stack developer Wes Bos. [*Try his course here*](https://reactforbeginners.com/friend/SITEPOINT), and use the code***SITEPOINT*** to get ***25% off***and to help support SitePoint.

**A typical web application is usually composed of several UI components that share data. Often, multiple components are tasked with the responsibility of displaying different properties of the same object. This object represents state which can change at any time. Keeping state consistent among multiple components can be a nightmare, especially if there are multiple channels being used to update the same object.**

Take, for example, a site with a shopping cart. At the top we have a UI component showing the number of items in the cart. We could also have another UI component that displays the total cost of items in the cart. If a user clicks the **Add to Cart** button, both of these components should update immediately with the correct figures. If the user decides to remove an item from the cart, change quantity, add a protection plan, use a coupon or change shipping location, then the relevant UI components should update to display the correct information. As you can see, a simple shopping cart can quickly become difficult to keep in sync as the scope of its features grows.

In this guide, I’ll introduce you to a framework known as [Redux](https://github.com/reactjs/redux), which can help you build complex projects in way that’s easy to scale and maintain. To make learning easier, we’ll use a simplified **shopping cart project** to learn how Redux works. You’ll need to be at least familiar with the [React](https://www.sitepoint.com/getting-started-react-beginners-guide/) library, as you’ll later need to integrate it with Redux.

## Prerequisites

Before we get started, make sure you’re familiar with the following topics:

* [Functional JavaScript](https://www.sitepoint.com/introduction-functional-javascript/)
* [Object-oriented JavaScript](https://www.sitepoint.com/oriented-programming-1/)
* [ES6 JavaScript Syntax](https://www.sitepoint.com/shorthand-javascript-techniques/)

Also, ensure you have the following setup on your machine:

* [a NodeJS environment](https://www.sitepoint.com/beginners-guide-node-package-manager/)
* [a Yarn setup](https://www.sitepoint.com/yarn-vs-npm/) (recommended)

You can access the entire code used in this tutorial on [GitHub](https://github.com/brandiqa/redux-shopping-cart).

## What is Redux

Redux is a popular JavaScript framework that provides a predictable state container for applications. Redux is based on a simplified version of Flux, a framework developed by Facebook. Unlike standard MVC frameworks, where data can flow between UI components and storage in both directions, Redux strictly allows data to flow in one direction only. See the below illustration:

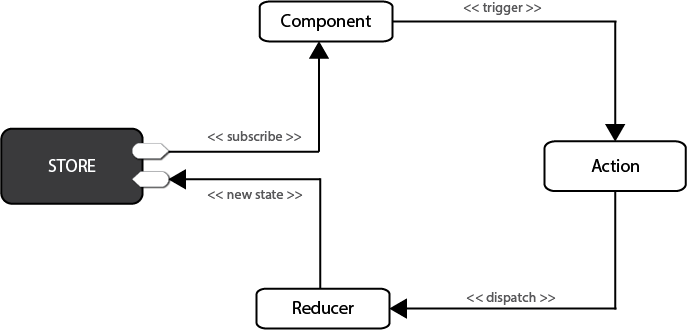


Figure 1: Redux Flow Chart

In Redux, all data — i.e. **state** — is held in a container known as the [store](http://redux.js.org/docs/basics/Store.html). There can only be one of these within an application. The store is essentially a state tree where states for all objects are kept. Any UI component can access the state of a particular object directly from the store. To change a state from a local or remote component, an [action](http://redux.js.org/docs/basics/Actions.html) needs to be dispatched. **Dispatch** in this context means sending actionable information to the store. When a store receives an action, it delegates it to the relevant [reducer](http://redux.js.org/docs/basics/Reducers.html). A reducer is simply a pure function that looks at the previous state, performs an action and returns a new state. To see all this in action, we need to start coding.

## Understand Immutability First

Before we start, I need you to first understand what **immutability** means in JavaScript. According to the Oxford English Dictionary, immutability means being **unchangeable**. In programming, we write code that changes the values of variables all the time. This is referred to as **mutability**. The way we do this can often cause unexpected bugs in our projects. If your code only deals with primitive data types (numbers, strings, booleans), then you don’t need to worry. However, if you’re working with Arrays and Objects, performing **mutable** operations on them can create unexpected bugs. To demonstrate this, open your terminal and launch the Node interactive shell:

node

Next, let’s create an array, then later assign it to another variable:

> let a = [1,2,3]

> let b = a

> b.push(9)

> console.log(b)

[ 1, 2, 3, 9 ] // b output

> console.log(a)

[ 1, 2, 3, 9 ] // a output

As you can see, updating array b caused array a to change as well. This happens because Objects and Arrays are known **referential data types** — meaning that such data types don’t actually hold values themselves, but are pointers to a memory location where the values are stored. By assigning a to b, we merely created a second pointer that references the same location. To fix this, we need to copy the referenced values to a new location. In JavaScript, there are three different ways of achieving this:

1. using immutable data structures created by [Immutable.js](https://facebook.github.io/immutable-js/)
2. using JavaScript libraries such as [Underscore](http://underscorejs.org/) and [Lodash](https://lodash.com/) to execute immutable operations
3. using native **ES6** functions to execute immutable operations.

For this article, we’ll use the **ES6** way, since it’s already available in the NodeJS environment. Inside your NodeJS terminal, execute the following:

> a = [1,2,3] // reset a

[ 1, 2, 3 ]

> b = Object.assign([],a) // copy array a to b

[ 1, 2, 3 ]

> b.push(8)

> console.log(b)

[ 1, 2, 3, 8 ] // b output

> console.log(a)

[ 1, 2, 3 ] // a output

In the above code example, array b can now be modified without affecting array a. We’ve used [Object.assign()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/assign) to create a new copy of values that variable b will now point to. We can also use the rest operator(...) to perform an immutable operation like this:

> a = [1,2,3]

[ 1, 2, 3 ]

> b = [...a, 4, 5, 6]

[ 1, 2, 3, 4, 5, 6 ]

> a

[ 1, 2, 3 ]

The rest operator works with object literals too! I won’t go deep into this subject, but here are some additional ES6 functions that we’ll use to perform immutable operations:

* [spread syntax](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_operator) — useful in append operations
* [map function](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map) — useful in an update operation
* [filter function](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/filter) — useful in a delete operation

In case the documentation I’ve linked isn’t useful, don’t worry, as you’ll see how they’re used in practice. Let’s start coding!

## Setting up Redux

The fastest way to set up a Redux development environment is to use the create-react-app tool. Before we begin, make sure you’ve installed and updated nodejs, npm and yarn. Let’s set up a Redux project by generating a redux-shopping-cartproject and installing the [Redux](https://www.npmjs.com/package/redux) package:

create-react-app redux-shopping-cart

cd redux-shopping-cart

yarn add redux # or npm install redux

Delete all files inside the src folder except index.js. Open the file and clear out all existing code. Type the following:

import { createStore } from "redux";

const reducer = function(state, action) {

return state;

}

const store = createStore(reducer);

Let me explain what the above piece of code does:

* **1st statement**. We import a createStore() function from the Redux package.
* **2nd statement**. We create an empty function known as a **reducer**. The first argument, state, is current data held in the store. The second argument, action, is a container for:
  + **type** — a simple string constant e.g. ADD, UPDATE, DELETE etc.
  + **payload** — data for updating state
* **3rd statement**. We create a Redux store, which can only be constructed using a reducer as a parameter. The data kept in the Redux store can be accessed directly, but can only be updated via the supplied reducer.

You may have noticed I mentioned current data as if it already exists. Currently, our state is undefined or null. To remedy this, just assign a default value to state like this to make it an empty array:

const reducer = function(state=[], action) {

return state;

}

Now, let’s get practical. The reducer we created is generic. Its name doesn’t describe what it’s for. Then there’s the issue of how we work with multiple reducers. The answer is to use a combineReducers function that’s supplied by the Redux package. Update your code as follows:

// src/index.js

…

import { combineReducers } from 'redux';

const productsReducer = function(state=[], action) {

return state;

}

const cartReducer = function(state=[], action) {

return state;

}

const allReducers = {

products: productsReducer,

shoppingCart: cartReducer

}

const rootReducer = combineReducers(allReducers);

let store = createStore(rootReducer);

In the code above, we’ve renamed the generic reducer to cartReducer. There’s also a new empty reducer named productsReducer that I’ve created just to show you how to combine multiple reducers within a single store using the combineReducersfunction.

Next, we’ll look at how we can define some test data for our reducers. Update the code as follows:

// src/index.js

…

const initialState = {

cart: [

{

product: 'bread 700g',

quantity: 2,

unitCost: 90

},

{

product: 'milk 500ml',

quantity: 1,

unitCost: 47

}

]

}

const cartReducer = function(state=initialState, action) {

return state;

}

…

let store = createStore(rootReducer);

console.log("initial state: ", store.getState());

Just to confirm that the store has some initial data, we use store.getState() to print out the current state in the console. You can run the dev server by executing npm start or yarn start in the console. Then press Ctrl+Shift+I to open the inspector tab in Chrome in order to view the console tab.

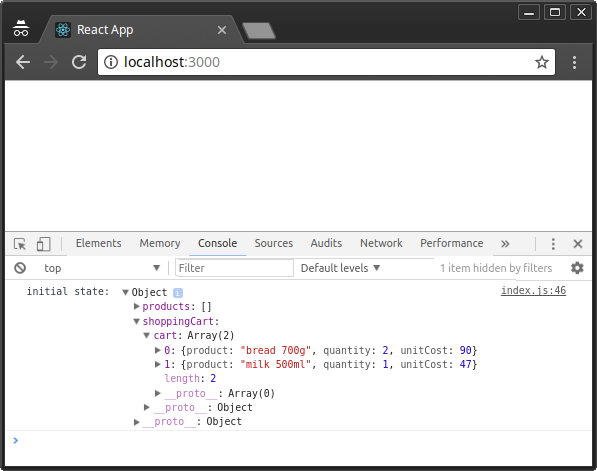


Figure 2: Redux Initial State

Currently, our cartReducer does nothing, yet it’s supposed to manage the state of our shopping cart items within the Redux store. We need to define actions for adding, updating and deleting shopping cart items. Let’s start by defining logic for a ADD\_TO\_CART action:

// src/index.js

…

const ADD\_TO\_CART = 'ADD\_TO\_CART';

const cartReducer = function(state=initialState, action) {

switch (action.type) {

case ADD\_TO\_CART: {

return {

...state,

cart: [...state.cart, action.payload]

}

}

default:

return state;

}

}

…

Take your time to analyze and understand the code. A reducer is expected to handle different action types, hence the need for a SWITCH statement. When an action of type ADD\_TO\_CART is dispatched anywhere in the application, the code defined here will handle it. As you can see, we’re using the information provided in action.payload to combine to an existing state in order to create a new state.

Next, we’ll define an action, which is needed as a parameter for store.dispatch(). **Actions** are simply JavaScript objects that must have type and an optional payload. Let’s go ahead and define one right after the cartReducer function:

…

function addToCart(product, quantity, unitCost) {

return {

type: ADD\_TO\_CART,

payload: { product, quantity, unitCost }

}

}

…

Here, we’ve defined a function that returns a plain JavaScript object. Nothing fancy. Before we dispatch, let’s add some code that will allow us to listen to store event changes. Place this code right after the console.log() statement:

…

let unsubscribe = store.subscribe(() =>

console.log(store.getState())

);

unsubscribe();

Next, let’s add several items to the cart by dispatching actions to the store. Place this code before unsubscribe():

…

store.dispatch(addToCart('Coffee 500gm', 1, 250));

store.dispatch(addToCart('Flour 1kg', 2, 110));

store.dispatch(addToCart('Juice 2L', 1, 250));

For clarification purposes, I’ll illustrate below how the entire code should look after making all the above changes:

// src/index.js

import { createStore } from "redux";

import { combineReducers } from 'redux';

const productsReducer = function(state=[], action) {

return state;

}

const initialState = {

cart: [

{

product: 'bread 700g',

quantity: 2,

unitCost: 90

},

{

product: 'milk 500ml',

quantity: 1,

unitCost: 47

}

]

}

const ADD\_TO\_CART = 'ADD\_TO\_CART';

const cartReducer = function(state=initialState, action) {

switch (action.type) {

case ADD\_TO\_CART: {

return {

...state,

cart: [...state.cart, action.payload]

}

}

default:

return state;

}

}

function addToCart(product, quantity, unitCost) {

return {

type: ADD\_TO\_CART,

payload: {

product,

quantity,

unitCost

}

}

}

const allReducers = {

products: productsReducer,

shoppingCart: cartReducer

}

const rootReducer = combineReducers(allReducers);

let store = createStore(rootReducer);

console.log("initial state: ", store.getState());

let unsubscribe = store.subscribe(() =>

console.log(store.getState())

);

store.dispatch(addToCart('Coffee 500gm', 1, 250));

store.dispatch(addToCart('Flour 1kg', 2, 110));

store.dispatch(addToCart('Juice 2L', 1, 250));

unsubscribe();

After you’ve saved your code, Chrome should automatically refresh. Check the console tab to confirm that the new items have been added:

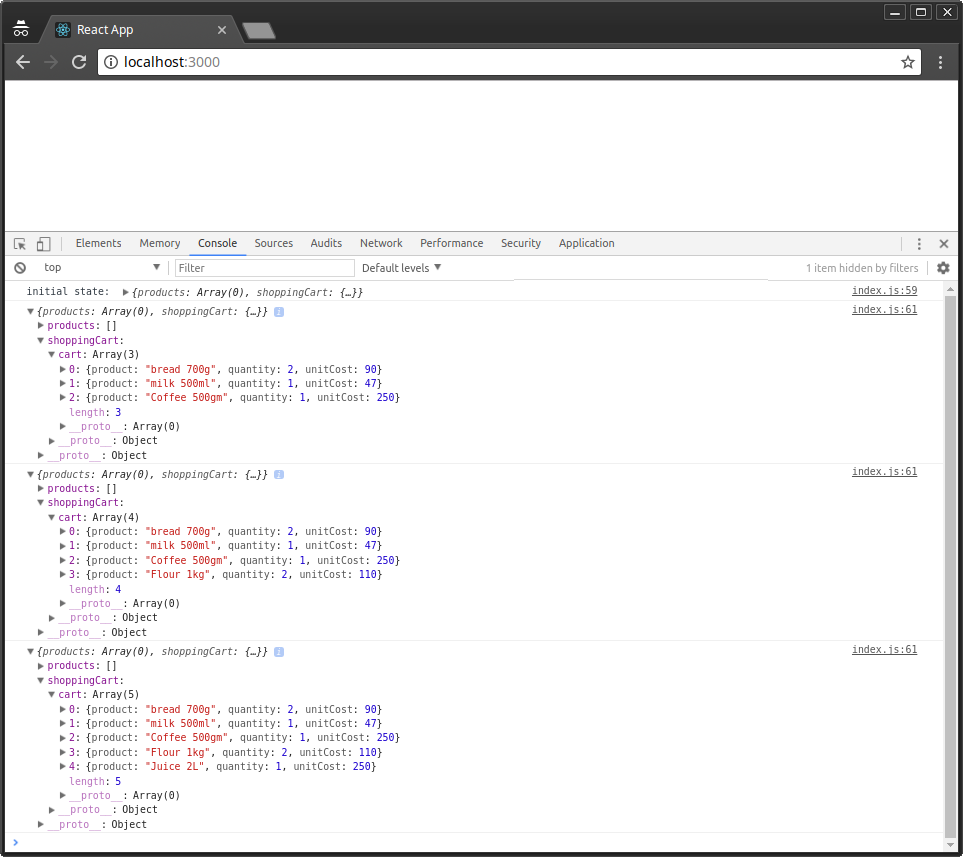


Figure 3: Redux Actions Dispatched

## Organizing Redux Code

The index.js file has quickly grown large. This is not how Redux code is written. I’ve only done this to show you how simple Redux is. Let’s look at how a Redux project should be organized. First, create the following folders and files within the src folder, as illustrated below:

src/

├── actions

│ └── cart-actions.js

├── index.js

├── reducers

│ ├── cart-reducer.js

│ ├── index.js

│ └── products-reducer.js

└── store.js

Next, let’s start moving code from index.js to the relevant files:

// src/actions/cart-actions.js

export const ADD\_TO\_CART = 'ADD\_TO\_CART';

export function addToCart(product, quantity, unitCost) {

return {

type: ADD\_TO\_CART,

payload: { product, quantity, unitCost }

}

}

// src/reducers/products-reducer.js

export default function(state=[], action) {

return state;

}

// src/reducers/cart-reducer.js

import { ADD\_TO\_CART } from '../actions/cart-actions';

const initialState = {

cart: [

{

product: 'bread 700g',

quantity: 2,

unitCost: 90

},

{

product: 'milk 500ml',

quantity: 1,

unitCost: 47

}

]

}

export default function(state=initialState, action) {

switch (action.type) {

case ADD\_TO\_CART: {

return {

...state,

cart: [...state.cart, action.payload]

}

}

default:

return state;

}

}

// src/reducers/index.js

import { combineReducers } from 'redux';

import productsReducer from './products-reducer';

import cartReducer from './cart-reducer';

const allReducers = {

products: productsReducer,

shoppingCart: cartReducer

}

const rootReducer = combineReducers(allReducers);

export default rootReducer;

// src/store.js

import { createStore } from "redux";

import rootReducer from './reducers';

let store = createStore(rootReducer);

export default store;

// src/index.js

import store from './store.js';

import { addToCart } from './actions/cart-actions';

console.log("initial state: ", store.getState());

let unsubscribe = store.subscribe(() =>

console.log(store.getState())

);

store.dispatch(addToCart('Coffee 500gm', 1, 250));

store.dispatch(addToCart('Flour 1kg', 2, 110));

store.dispatch(addToCart('Juice 2L', 1, 250));

unsubscribe();

After you’ve finished updating the code, the application should run as before now that it’s better organized. Let’s now look at how we can update and delete items from the shopping cart. Open cart-reducer.js and update the code as follows:

// src/reducers/cart-actions.js

…

export const UPDATE\_CART = 'UPDATE\_CART';

export const DELETE\_FROM\_CART = 'DELETE\_FROM\_CART';

…

export function updateCart(product, quantity, unitCost) {

return {

type: UPDATE\_CART,

payload: {

product,

quantity,

unitCost

}

}

}

export function deleteFromCart(product) {

return {

type: DELETE\_FROM\_CART,

payload: {

product

}

}

}

Next, update cart-reducer.js as follows:

// src/reducers/cart-reducer.js

…

export default function(state=initialState, action) {

switch (action.type) {

case ADD\_TO\_CART: {

return {

...state,

cart: [...state.cart, action.payload]

}

}

case UPDATE\_CART: {

return {

...state,

cart: state.cart.map(item => item.product === action.payload.product ? action.payload : item)

}

}

case DELETE\_FROM\_CART: {

return {

...state,

cart: state.cart.filter(item => item.product !== action.payload.product)

}

}

default:

return state;

}

}

Finally, let’s dispatch the UPDATE\_CART and DELETE\_FROM\_CART actions in index.js:

// src/index.js

…

// Update Cart

store.dispatch(updateCart('Flour 1kg', 5, 110));

// Delete from Cart

store.dispatch(deleteFromCart('Coffee 500gm'));

…

Your browser should automatically refresh once you’ve saved all the changes. Check the console tab to confirm the results:

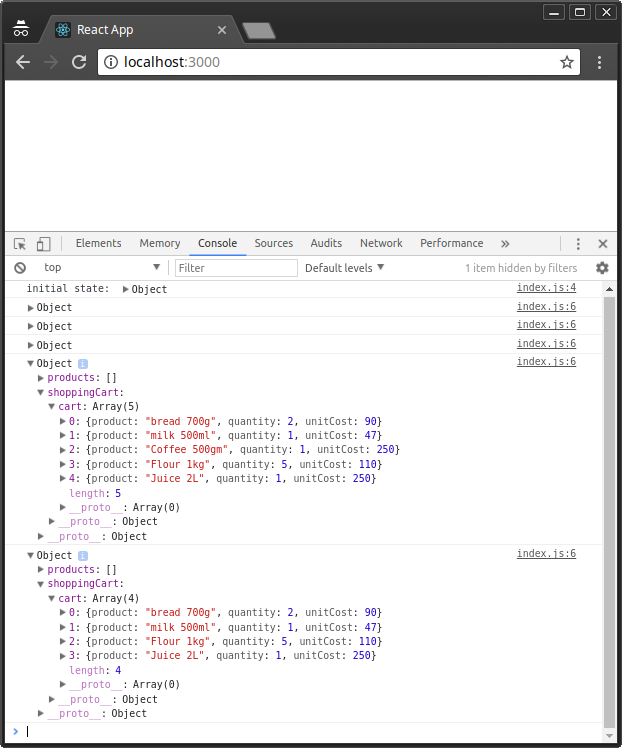


Figure 4: Redux Update and Delete Actions

As confirmed, the quantity for 1kg of flour is updated from 2 to 5, while the the 500gm of coffee gets deleted from cart.

## Debugging with Redux tools

Now, if we’ve made a mistake in our code, how do we debug a Redux project?

Redux comes with a lot of third-party debugging tools we can use to analyze code behavior and fix bugs. Probably the most popular one is the **time-travelling tool**, otherwise known as [redux-devtools-extension](https://www.npmjs.com/package/redux-devtools-extension). Setting it up is a 3-step process. First, go to your Chrome browser and install the [Redux Devtools extension](https://chrome.google.com/webstore/detail/redux-devtools/lmhkpmbekcpmknklioeibfkpmmfibljd?hl=en).

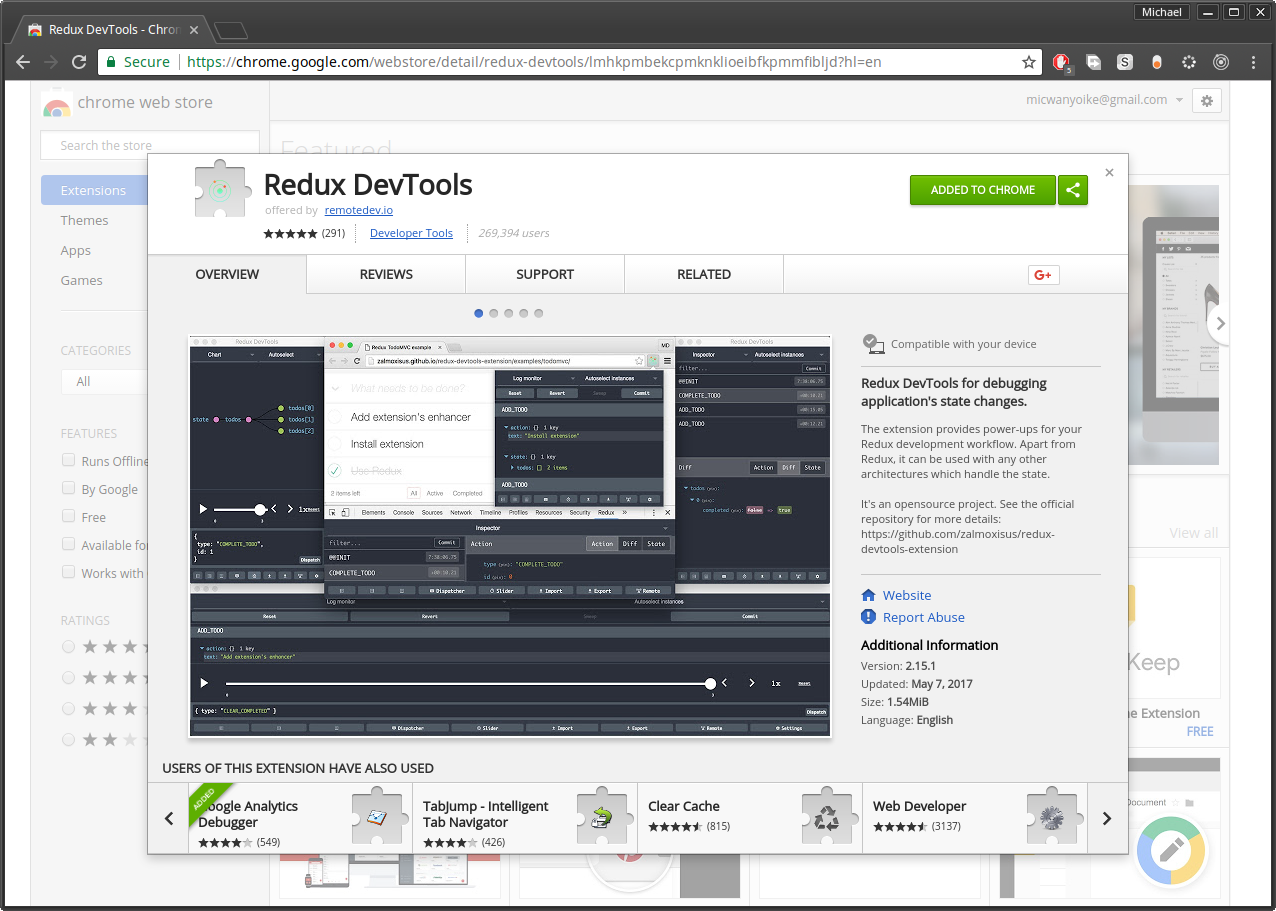


Figure 5: Redux DevTools Chrome Extensions

Next, go to your terminal where your Redux application is running and press Ctrl+Cto stop the development server. Next, use npm or yarn to install the [redux-devtools-extension](https://www.npmjs.com/package/redux-devtools-extension) package. Personally, I prefer Yarn, since there’s a yarn.lock file that I’d like to keep updated.

yarn add redux-devtools-extension

Once installation is complete, you can start the development server as we implement the final step of implementing the tool. Open store.js and replace the existing code as follows:

// src/store.js

import { createStore } from "redux";

import { composeWithDevTools } from 'redux-devtools-extension';

import rootReducer from './reducers';

const store = createStore(rootReducer, composeWithDevTools());

export default store;

Feel free to update src/index.js and remove all code related with logging to the console and subscribing to the store. This is no longer needed. Now, go back to Chrome and open the Redux DevTools panel by right-clicking the tool’s icon:

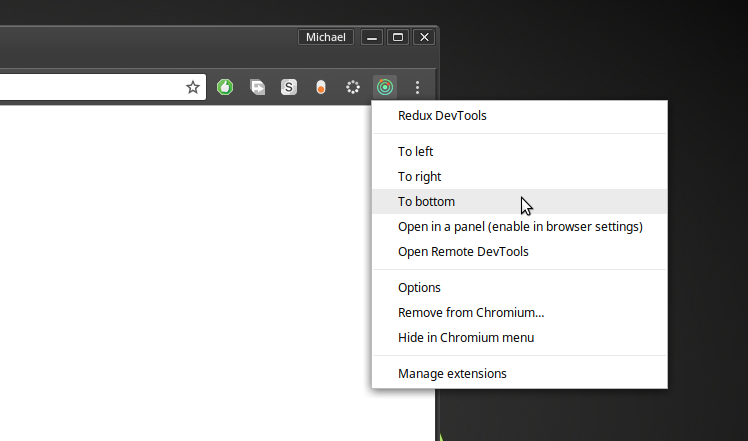


Figure 6: Redux DevTools Menu

In my case, I’ve selected to **To Bottom** option. Feel free to try out other options.

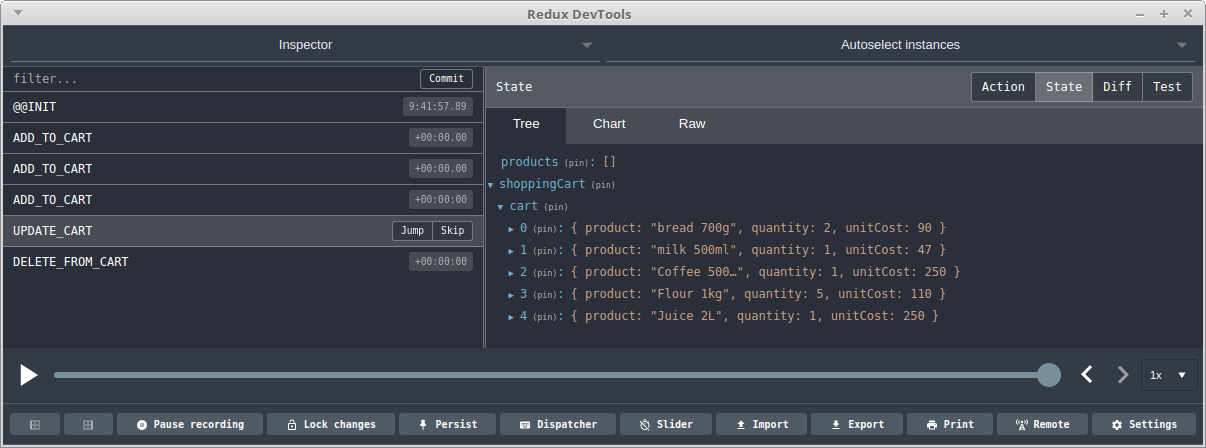


Figure 7: Redux DevTools Panel

As you can see, the Redux Devtool is quite amazing. You can toggle between action, state and diff methods. Select actions on the left panel and observe how the state tree changes. You can also use the slider to play back the sequence of actions. You can even dispatch directly from the tool! Do check out the [documentation](https://github.com/gaearon/redux-devtools) to learn more on how you can further customize the tool to your needs.

## Integration with React

At the beginning of this tutorial, I mentioned Redux really pairs well with React. Well, you only need a few steps to setup the integration. Firstly, stop the development server, as we’ll need to install the [react-redux](https://github.com/reactjs/react-redux) package, the official Redux bindings for React:

yarn add react-redux

Next, update index.js to include some React code. We’ll also use the Providerclass to wrap the React application within the Redux container:

// src/index.js

…

import React from 'react';

import ReactDOM from 'react-dom';

import { Provider } from 'react-redux';

const App = <h1>Redux Shopping Cart</h1>;

ReactDOM.render(

<Provider store={store}>

{ App }

</Provider> ,

document.getElementById('root')

);

…

Just like that, we’ve completed the first part of the integration. You can now start the server to see the result. The second part involves linking React’s components with the Redux store and actions using a couple of functions provided by the react-reduxpackage that we just installed. In addition, you’ll need to set up an API using [Express](https://expressjs.com/)or a framework like [Feathers](https://feathersjs.com/). The API will provide our application with access to a database service.

In Redux, we’ll also need to install further packages such as axios to perform API requests via Redux actions. Our React components state will then be handled by Redux, making sure that all components are in sync with the database API. To learn more on how to accomplish all this, do take a look at my other tutorial, “[Build a CRUD App Using React, Redux and FeathersJS](https://www.sitepoint.com/crud-app-react-redux-feathersjs/)”.

<http://www.thegreatcodeadventure.com/react-redux-tutorial-part-iv-the-index-feature/>