- JavaScript(https://www.sitepoint.com/javascript/)

How to Build a Todo App Using React, Redux, and Immutable.js

By Dan Prince (https://www.sitepoint.com/author/dprince/)

This article was rewritten on 3rd May, 2016. Comments relating to the original article have been removed.

The way React (https://facebook.github.io/react) uses components and a one-way data flow makes it ideal for describing the structure of user interfaces, however its tools for working with state are kept deliberately simple; to help remind us that React is just the View in the traditional Model-View-Controller (http://blog.codinghorror.com/understanding-model-view-controller/) architecture.

More from this author

Quick Tip: What Are Factory Functions in JavaScript (https://www.sitepoint.com/factory-functions-javascript/?utm_source=sitepoint&utm_medium=relatedinline&utm_term=&utm_campaign=relatedauthor)

10 Lodash Features You Can Replace with ES6 (https://www.sitepoint.com/lodash-features-replace-es6/?utm_source=sitepoint&utm_medium=relatedinline&utm_term=&utm_campaign=relatedauthor)

Understanding ASTs by Building Your Own Babel Plugin (https://www.sitepoint.com/understanding-asts-building-babel-plugin/?utm_source=sitepoint&utm_medium=relatedinline&utm_term=&utm_campaign=relatedauthor)

There's nothing to stop us from building large applications with just React, but we would quickly discover that to keep our code simple, we'd need to manage our state elsewhere.

Whilst there's no *official* solution for dealing with application state, there are some libraries that align particularly well with React's paradigm. Today we'll pair React with two such libraries and use them to build a simple application.

Redux

Redux (https://redux.js.org) is a tiny library that acts as a container for our application state, by combining ideas from Flux (https://facebook.github.io/flux) and Elm (https://github.com/evancz/elm-architecture-tutorial). We can use Redux to manage any kind of application state, providing we stick to the following guidelines:

1. Our state is kept in a single store
2. Changes come from *actions* not *mutations*

At the core of a Redux store is a function that takes the current application state and an action and combines them to create a new application state. We call this function a *reducer*.

Our React components will be responsible for sending actions to our store and in turn our store will tell the components when they need to re-render.

ImmutableJS

Because Redux doesn't allow us to mutate the application state, it can be helpful to enforce this by modeling application state with immutable data structures.

ImmutableJS (https://facebook.github.io/immutable-js) offers us a number of immutable data structures with mutative interfaces and they're implemented in an efficient way (https://www.youtube.com/watch?v=I7IdS-PbEgI), inspired by the implementations in Clojure and Scala.

Demo

We're going to use React with Redux and ImmutableJS to build a simple todo list that allows us to add todos and toggle them between complete and incomplete.

(/?utm_source=sitepoint&utm_medium=nav)
Northwestern
Learn More
Earn your MS in Information Design and Strategy online from Northwestern University
(//srv.buysellads.com/ads/click/x/GTND4237CAYIP2Q7CKA4YKQWF6SI6KJJCVYD4Z3JCEAI5KJ7CEYIE27KC6BDL27JCWSI6K3EHJNCLSIZ?segment=placement:sitepoint;)

HTML CSS Babel Result EDIT ON

Add todo

```
const { Provider, connect } = reactRedux;

const components = {
  Todo({ todo }) {
    if(todo.isDone) {
      return <strike>{todo.text}</strike>;
    } else {
      return <span>{todo.text}</span>;
    }
  },
  TodoList({ todos, toggleTodo, addTodo }) {
    const onSubmit = (e) => {
      const text = e.target.value;
      if(e.which === 13 && text.length > 0) {
        addTodo(text);
        e.target.value = '';
      }
    };
```

The code is also available on GitHub (https://github.com/sitepoint-editors/immutable-redux-todo).

Setup

We'll get started by creating a project folder and initializing a `package.json` file with `npm init`. Then we'll install that dependencies that we're going to need.

```
npm install --save react react-dom redux react-redux immutable
npm install --save-dev webpack babel-loader babel-preset-es2015 babel-preset-react
```

We'll be using JSX (https://facebook.github.io/jsx/) and ES2015 (http://www.ecma-international.org/ecma-262/6.0/), so we'll compile our code with Babel (http://babeljs.io) and we're going to do this as part of the module bundling process with Webpack (https://webpack.github.io/).

First we'll create our Webpack configuration in `webpack.config.js`.

```
module.exports = {
  entry: './src/app.js',
  output: {
    path: __dirname,
    filename: 'bundle.js'
  },
  module: {
    loaders: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        loader: 'babel',
        query: { presets: [ 'es2015', 'react' ] }
      }
    ]
  }
};
```

And finally we'll extend our `package.json` by adding an npm script to compile our code with source maps.

(/?utm_source=sitepoint&utm_medium=nav)
Northwestern

```
"scripts": {
  "build": "webpack --debug"
```

We'll need to run `npm run build` each time we want to compile our code.

React & Components

Before we implement any components, it can be helpful to create some dummy data. This helps us get a feel for what we're going to need our components to render.

```
const dummyTodos = [
  { id: 0, isDone: true,  text: 'make components' },
  { id: 1, isDone: false, text: 'design actions' },
  { id: 2, isDone: false, text: 'implement reducer' },
  { id: 3, isDone: false, text: 'connect components' }
];
```

For this application, we're only going to need two React components, `<Todo />` and `<TodoList />`.


```
// ./src/components.js

import React from 'react';

export function Todo(props) {
  const { todo } = props;
  if(todo.isDone) {
    return <strike>{todo.text}</strike>;
  } else {
    return <span>{todo.text}</span>;
  }
}

export function TodoList(props) {
  const { todos } = props;
  return (
    <div className='todo'>
      <input type='text' placeholder='Add todo' />
      <ul className='todo__list'>
        {todos.map(t => (
          <li key={t.id} className='todo__item'>
            <Todo todo={t} />
          </li>
        ))}
      </ul>
    </div>
  );
}
```

At this point, we can test these components by creating an `index.html` file in the project folder and populating it with the following markup. (You can find a simple stylesheet here on GitHub (https://github.com/sitepoint-editors/immutable-redux-todo/blob/master/style.css)).

```
<!DOCTYPE html>
<html>
  <head>
    <link rel="stylesheet" href="style.css">
    <title>Immutable Todo</title>
  </head>
  <body>
    <div id="app"></div>
    <script src="bundle.js"></script>
  </body>
</html>
```

We'll also need an application entry point at `src/app.js`.


```
// ./src/app.js

import React from 'react';
import { render } from 'react-dom';
```

```
const dummyTodos = [
  { id: 0, isDone: true,  text: 'make components' },
  { id: 1, isDone: false, text: 'design actions' },
  { id: 2, isDone: false, text: 'implement reducer' },
  { id: 3, isDone: false, text: 'connect components' }
];

render(
  <TodoList todos={dummyTodos} />,
  document.getElementById('app')
);
```

Compile the code with `npm run build`, then navigate your browser to the `index.html` file and make sure that it's working.

Redux & ImmutableJS

Now that we're happy with the user interface, we can start to think about the state behind it. Our dummy data is a great place to start from and we can easily translate it into ImmutableJS collections.

```
import { List, Map } from 'immutable';

const dummyTodos = List([
  Map({ id: 0, isDone: true,  text: 'make components' }),
  Map({ id: 1, isDone: false, text: 'design actions' }),
  Map({ id: 2, isDone: false, text: 'implement reducer' }),
  Map({ id: 3, isDone: false, text: 'connect components' })
]);
```

ImmutableJS maps don't work in the same way as JavaScript's objects, so we'll need to make some slight tweaks to our components. Anywhere there was a property access before (e.g. `todo.id`) needs to become a method call instead (`todo.get('id')`).

Designing Actions

Now that we've got the shape and structure figured out, we can start thinking about the actions that will update it. In this case, we'll only need two actions, one to add a new todo and the other to toggle an existing one.

Let's define some functions to create these actions.


```
// src/actions.js

// succinct hack for generating passable unique ids
const uid = () => Math.random().toString(34).slice(2);

export function addTodo(text) {
  return {
    type: 'ADD_TODO',
    payload: {
      id: uid(),
      isDone: false,
      text: text
    }
  };
}

export function toggleTodo(id) {
  return {
    type: 'TOGGLE_TODO',
    payload: id
  }
}
```

Each action is just a JavaScript object with a type and payload properties. The type property helps us decide what to do with the payload when we process the action later.

Designing a Reducer

Now that we know the shape of our state and the actions that update it, we can build our reducer. Just as a reminder, the reducer is a function which takes a state and an action, then uses them to compute a new state.

Here's the initial structure for our reducer.

```
// src/reducer.js

import { List, Map } from 'immutable';

const init = List([]);

export default function(todos=init, action) {
  switch(action.type) {
    case 'ADD_TODO':
      // ...
    case 'TOGGLE_TODO':
      // ...
    default:
      return todos;
  }
}
```

Handling the `ADD_TODO` action is quite simple, as we can use the ,push() (https://facebook.github.io/immutable-js/docs/#/List/push) method, which will return a new list with the todo appended at the end.

```
case 'ADD_TODO':
  return todos.push(Map(action.payload));
```

Notice that we're also converting the todo object into an immutable map before it's pushed onto the list.

The more complex action we need to handle is `TOGGLE_TODO`.


```
case 'TOGGLE_TODO':
  return todos.map(t => {
    if (t.get('id') !== action.payload) {
      return t.update('isDone', isDone => !isDone);
```

```
    } else {
      return t;
    }
  });
```

We're using .map() (https://facebook.github.io/immutable-js/docs/#/List/map) to iterate over the list and find the todo whose `id` matches the action. Then we call .update() (https://facebook.github.io/immutable-js/docs/#/Map/update) method, which takes a key and a function, then it returns a new copy of the map, with the value at the key replaced with the result of passing the initial value to the update function.

It might help to see the literal version.

```
const todo = Map({ id: 0, text: 'foo', isDone: false });
todo.update('isDone', isDone => !isDone);
// => { id: 0, text: 'foo', isDone: true }
```

Connecting Everything

Now we've got our actions and reducer ready, we can create a store and connect it to our React components.

```
// src/app.js

import React from 'react';
import { render } from 'react-dom';
import { createStore } from 'redux';
import { TodoList } from './components';
import reducer from './reducer';

const store = createStore(reducer);

render(
  <TodoList todos={store.getState()} />,
  document.getElementById('app')
);
```

We'll need to make our components aware of this store. We'll use the react-redux (https://github.com/reactjs/react-redux) to help simplify this process. It allows us to create store-aware containers that wrap around our components, so that we don't have to change our original implementations.

We're going to need a container around our `<TodoList />` component. Let's see what this looks like.


```
// src/containers.js

import { connect } from 'react-redux';
import * as components from './components';
import { addTodo, toggleTodo } from './actions';

export const TodoList = connect(
  function mapStateToProps(state) {
    // ...
  },
  function mapDispatchToProps(dispatch) {
    // ...
  }
)(components.TodoList);
```

We create containers with the connect (https://github.com/reactjs/react-redux/blob/master/docs/api.md#connectmapstatetoprops-mapdispatchtoprops-mergeprops-options) function. When we call `connect()` we pass two functions, `mapStateToProps()` and `mapDispatchToProps()`.

The `mapStateToProps` function takes the store's current state as an argument (in our case a list of todos), then it expects the return value to be an object that describes a mapping from that state to props for our wrapped component.

```
function mapStateToProps(state) {
  return { todos: state };
}
```

It might help to visualize this on an instance of the wrapped React component.

```
<TodoList todos={state} />
```

We'll also need to supply a `mapDispatchToProps` function, which is passed the store's `dispatch` method, so that we can use it to dispatch the actions from our action creators.

```
function mapDispatchToProps(dispatch) {
  return {
    addTodo: text => dispatch(addTodo(text)),
    toggleTodo: id => dispatch(toggleTodo(id))
  };
}
```

Again, it might help to visualize all these props together on an instance of our wrapped React component.

```
<TodoList todos={state}
          addTodo={text => dispatch(addTodo(text))}
          toggleTodo={id => dispatch(toggleTodo(id))} />
```

Now that we've mapped our component knows about the action creators, we can call them from event listeners.


```
// src/components.js

function TodoList(props) {
  const { todos, toggleTodo, addTodo } = props;

  const onSubmit = (event) => {
    const input = event.target;
    const text = input.value;
    const isEnterKey = (event.which == 13);
    const isLongEnough = text.length > 0;
```

```
    if(isEnterKey && isLongEnough) {
      input.value = '';
      addTodo(text);
    }
  };

  const toggleClick = id => event => toggleTodo(id);

  return (
    <div className='todo'>
      <input type='text'
             className='todo__entry'
             placeholder='Add todo'
             onKeyDown={onSubmit} />
      <ul className='todo__list'>
        {todos.map(t => (
          <li key={t.get('id')}
              className='todo__item'
              onClick={toggleClick(t.get('id'))}>
            <Todo todo={t.toJS()} />
          </li>
        ))}
      </ul>
    </div>
  );
}
```

The containers will automatically subscribe to changes in the store and they will re-render the wrapped components whenever their mapped props change.

Finally, we need to make the containers aware of the store, using the `<Provider />` component.

```
// src/app.js

import React from 'react';
import { render } from 'react-dom';
import { createStore } from 'redux';
import { Provider } from 'react-redux';
import reducer from './reducer';
import { TodoList } from './containers';
//                          ^^^^^^^^^^

const store = createStore(reducer);

render(
  <Provider store={store}>
    <TodoList />
  </Provider>,
  document.getElementById('app')
);
```

Conclusion

There's no denying that the ecosystem around React and Redux can be quite complex and intimidating for beginners, but the good news is that almost all of these concepts are transferable. We've barely touched the surface of Redux's architecture, but already we've seen enough to help us start learning about The Elm Architecture (https://github.com/evancz/elm-architecture-tutorial), or pick up a ClojureScript library like Om (https://github.com/omcljs/om) or Re-frame (https://github.com/Day8/re-frame). Likewise, we've only seen a fragment of the possibilities with immutable data, but now we're better equipped to start learning a language like Clojure (http://clojure.org/) or Haskell (https://www.haskell.org/).

Whether you're just exploring the state of web application development, or you spend all day writing JavaScript; experience with action based architectures and immutable data is already becoming a vital skill for developers and *right now* is a great time to be learning the essentials.

Was this helpful?

More: Immutable.js (https://www.sitepoint.com/tag/immutable-js/), javascript (https://www.sitepoint.com/tag/javascript-2/), React (https://www.sitepoint.com/tag/react/), redux (https://www.sitepoint.com/tag/redux/)

Meet the author
Dan Prince (https://www.sitepoint.com/author/dprince/) (https://twitter.com/workshydev) (https://github.com/danprince)

Digital Nomad and co-founder of UK based startup Astral Dynamics.

36 Comments **SitePoint** Login

Recommend 9 Share Sort by Best

Join the discussion…

Wayou Liu • 9 months ago
`npm run build` won't run until I replace the `script` with `scripts`
4 • Reply • Share ›

Nilson Jacques SitePoint Staff → Wayou Liu • 7 months ago
Fixed!
• Reply • Share ›

Lucas Pereira Caixeta → Wayou Liu • 7 months ago
yeap, true!
• Reply • Share ›

Kris T • 7 months ago
This was exceptionally useful, but I'm curious about one thing:

const toggleClick = id => event => toggleTodo(id)

What is this line doing? I would have thought it would have worked without the EVENT portion, but sure enough it breaks if I take that out. What is going on there?
1 • Reply • Share ›

Sunwooz → Kris T • 2 months ago
curious about this as well. Did you find out?
• Reply • Share ›

Kris T → Sunwooz • 2 months ago
Unfortunately I never did. I'm assigned to a large PHP project at the moment and my React time had to take a back seat :(
• Reply • Share ›

Nilson Jacques SitePoint Staff → Kris T • 2 months ago
Hey @Kris T and @Sunwooz, the idea behind [illegible]s to create a handler function that still has access to the todo id outside of the `map` loop. To do this, you create a function that is called at render time, passing in the id of the current todo, which returns a new function that calls `toggleTodo()` with the id. It might be easier to read when written out in ES5 syntax:

```
var toggleClick = function(id) {
  return function(event) {
    toggleTodo(id);
  }
}
```
• Reply • Share ›

Raj Kumar N • 5 months ago
got error when executing cmd "npm run build" .

ERROR : ERROR in Cannot find module 'babel-core'

Reply · Share ›

Jorge Luis Monroy ➔ Raj Kumar N · 5 months ago

Run this code npm install babel-core babel-loader --save-dev

Reply · Share ›

John Emmerling · 5 months ago

Please take a look at my GitHub repository https://github.com/genma123.... I cited this post as resource in developing my example. I also recommend that anyone who reads this post should look at the README in my repository as well, thanks and Happy New Year!

Reply · Share ›

Nicholas Narbone · 7 months ago

I stead of push you should use concat I believe. push will add to the array and concat recreates a new array. you do not want to muck up the state in that way. At least from my understanding.

Reply · Share ›

djkramnik ➔ Nicholas Narbone · 6 months ago

well the whole point was that these todos weren't not being stored in a typical array. this ain't the Array.prototype.push method which mutates, it's the mighty immutablejs List push method which is non mutating. but yea, if you were using a normal array I guess concat is cooler for having no side effects.

Reply · Share ›

Giron · 8 months ago

A little mistake in "<todo todo="{t.toJS()}"/>", should be just "<todo todo="{t}"/>", because we are reading values using .get, so we assume Map as an input.

Reply · Share ›

Catalin Luntraru · 8 months ago

Cool. Really informative tutorial.

I hope with more and more tutorials, the entry barrier for react and redux lowers, so we get a greater community and a richer library environment.

If anyone want's to learn how to use react and redux, or wants to learn how to implement certain features into his app, like routing, bootstrap, sass, async API calls, forms, etc, I have made a video tutorial that goes through the entire process.

You can watch the chapter you are interested it, you don't have to watch the whole thing:
https://www.youtube.com/wat...

Code examples are in the video description. The demo app for the tutorial is at http://redux-minimal-app.ca... .

The tutorial uses the starter kit redux minimal which can be found at http://redux-minimal.js.org/

Reply · Share ›

Arr · 8 months ago

I've followed your directions to a T i even copied and pasted all the code, and for some reason i get an error : bundle.js:22516 Uncaught TypeError: Cannot read property 'displayName' of undefined ....

Reply · Share ›

Haru El Rovix · 8 months ago

Got ERROR in Cannot find module 'babel-core'. It seems missing babel-core it the Setup. Should it be
npm install --save-dev webpack babel-core babel-loader babel-preset-es2015 babel-preset-react ?

Reply · Share ›

Joshua Mark · 9 months ago

By the end it sounds like you're saying "now that you've been introduced to these concepts, you're equipped to learn a better framework". The more I learn React, the more I wonder why it's so popular. This rudimentary todo app has half a GB of dependencies and almost a full MB of client JS. I'm wondering if you're under the impression that React is a shoddy bandwagon?

Reply · Share ›

djkramnik ➔ Joshua Mark · 6 months ago

Yes this was my first dabbling with the shiny new React and my feeling is wtf. Hopefully my impression improves over time.

Reply · Share ›

Yuri Predborski · 10 months ago

I'm a complete newbie with react and I came here to make a simple todo app.
Funny thing: this works on codepen, but it doesn't work on my PC. Says I need to start HTTP server instead of loading a file.
Also, code after this paragraph:
Now that we've mapped our component knows about the action creators, we can call them from event listeners
Is not directly tied to any file, and its confusing for a complete newbie to try to understand where it belongs.

I will figure it out somehow (I always do), just give me some time xD

Thanks for the article! Now I know a bit more about the tools I plan to use! Even though its just the tip of the tip of the iceberg...

-edit-
Copied files from the git repository (they looked the same, I looked line after line, every file) and it started working. Magic...
Must've had a typo somewhere.

Reply · Share ›

朱琳 · 10 months ago

this page help me alot, thank you very much!

Reply · Share ›

Michal Merta · a year ago

Hi,
Thanx for the example. I'm just wondering if the default implementation of react-redux connect is the best for immutable js. I'm reading the code of shallowEqual (https://github.com/reactjs/... that connect method uses and thinking to myself if it wouldnt be sufficient to just compare with === and return false, when it's not equal.

What do You think of it?

Reply · Share ›

Laurel Anderson · a year ago

Keys should be stable, predictable, and unique. Unstable keys (like those produced by Math.random()) will cause many nodes to be unnecessarily re-created, which can cause performance degradation and lost state in child components.

Reply · Share ›

Dan Prince ➔ Laurel Anderson · a year ago

You're misunderstanding what that advice means. That's specifically for components with `<component key="{Math.random()}"/>` which defeats the whole point of using unique ids in the first place.

If you generate the keys in advance and keep them associated with the model (which is immutable), then the keys are going to be stable, regardless of how they were generated. For cryptographic strength, we can do better than Math.random, but for a demo or a simple application it's not going to give React any problems.

Reply · Share ›

Ragu Pathy · a year ago

You have used Stateless functions in this example, Please correct me if am wrong....

Reply · Share ›

Dan Prince ➔ Ragu Pathy · a year ago

Yep. In fact, I only used stateless functions, because I'm keeping track of all my component state with Redux instead.

Reply · Share ›

Alex Chen · a year ago

Uncaught ReferenceError: React is not defined

Reply · Share ›

Dan Prince ➔ Alex Chen · a year ago

Sounds like you haven't required or imported React. Remember, it needs to be imported in any file where you use JSX.

Reply · Share ›

Thomas Hoadley · a year ago

Hey great article! I was just wondering how long it took you personally to get to grips with react/redux? Also, did you have a developers background before hand? As you mention I'm definitely finding it all very confusing despite having put considerable work into it!

Reply · Share ›

Dan Prince ➔ Thomas Hoadley · a year ago

It didn't take long for me to pick up either library, but I learned them in total isolation and I had a 5 year background in web development when I started. The ideas in React were pretty new for me, but I'd learned the patterns in Redux from ClojureScript and Elm, so it always felt pretty natural. The major hurdle was the react-redux bindings, which I still don't find particularly intuitive.

Reply · Share ›

Thomas Hoadley ➔ Dan Prince · a year ago

Thanks for the reply. Since posting this I've actually got a lot better understanding of them. No replacement for good old fashioned hard work!

Reply · Share ›

Marc Sch · a year ago

Have you considered using Record to maintain the ability to reference object members directly?

Reply · Share ›

Dan Prince ➔ Marc Sch · a year ago

Sure, it's a perfectly good solution, but generally I prefer Map to Record in the same way that I prefer JavaScript objects to classes. I don't mind the syntactic indirection of `.get` too much either, but that's just personal preference.

Reply · Share ›

Cezar Luiz · a year ago

Great article! Very useful!

Reply · Share ›

Dan Prince ➔ Cezar Luiz · a year ago

Thanks Cezar!

Reply · Share ›

David Whitaker · a year ago

Looks like you just repeated the docs http://redux.js.org/docs/ba... with some minor alterations. Aside, I find it quite crazy how many wont read the docs and look everywhere else.

Though was looking for example for syntax on using classes with redux apposed to const react return function being I needed the lifecycle methods.

Reply · Share ›

foobar · a year ago

if(t.get('id') === action.payload) { SHOULD BE if(t.get('id') === action.payload.id) {

Reply · Share ›

Subscribe Add Disqus to your siteAdd DisqusAdd Privacy