JUNE 28, 2016

Redux · An Introduction

Coding 587 # Tools 172 # JavaScript 187 # Techniques 218

ABOUT THE AUTHOR

Alex is a front-end developer who specializes in JavaScript development. He's developed anything from WordPress websites to complex e-commerce JavaScript …

More about Alex…

Redux (↦ https://github.com/reactjs/redux) is one of the hottest libraries in front-end development these days. However, many people are confused about what it is and what its benefits are.

As the documentation (↦*http://redux.js.org/*) states, Redux is a predictable state container for JavaScript apps. To rephrase that, it's an application data-flow architecture, rather than a traditional library or a framework like Underscore.js and AngularJS.

Further Reading on SmashingMag (↦*#span-class-rh-further-reading-span-on-smashingmag)*

- Why You Should Consider React Native For Your Mobile App (↦ *https://www.smashingmagazine.com/2016/04/consider-react-native-mobile-app/)*

- Test Automation For Apps, Games And The Mobile Web *(↦ https://www.smashingmagazine.com/2015/01/basic-test-automation-for-apps-games-and-mobile-web/)*
- Server-Side Rendering With React, Node And Express *(↦ https://www.smashingmagazine.com/2016/03/server-side-rendering-react-node-express/)*
- Notes On Client-Rendered Accessibility *(↦ https://www.smashingmagazine.com/2015/05/client-rendered-accessibility/)*

Redux was created by Dan Abramov around June 2015. It was inspired by Facebook's Flux and functional programming language Elm. **Redux got popular very quickly because of its simplicity**, small size (only 2 KB) and great documentation. If you want to learn how Redux works internally and dive deep into the library, consider checking out Dan's free course *(↦ https://egghead.io/series/getting-started-with-redux)*.

Redux is used mostly for application state management. To summarize it, Redux maintains the state of an entire application in a single immutable state tree (object), which can't be changed directly. When something changes, a new object is created (using actions and reducers). We'll go over the core concepts in detail below.

How Is It Different From MVC And Flux? *(↦#how-is-it-different-from-mvc-and-flux)*

To give some perspective, let's take the classic model-view-controller (MVC) pattern, since most developers are familiar with it. In MVC architecture, there is a clear separation between data (model), presentation (view) and logic (controller). There is one issue with this, especially in large-scale applications:

The flow of data is bidirectional *(↦ http://stackoverflow.com/questions/33447710/mvc-vs-flux-bidirectional-vs-unidirectional)*. This means that one change (a user input or API response) can affect the state of an application in many places in the code — for example, two-way data binding. That can be hard to maintain and debug.

Flux is very similar to Redux. The main difference is that Flux has multiple stores that change the state of the application, and it broadcasts these changes as events. Components can subscribe to these events to sync with the current state. **Redux doesn't have a dispatcher**, which in Flux is used to broadcast payloads to registered callbacks. Another difference in Flux is that many *(↦ https://github.com/kriasoft/react-starter-kit/issues/22)* varieties are available, and that creates some confusion and inconsistency.

Benefits Of Redux *(↦ #benefits-of-redux)*

You may be asking, "Why would I need to use Redux?" Great question. There are a few benefits of using Redux in your next application:

- **Predictability of outcome**
  There is always one source of truth, the store, with no confusion about how to sync the current state with actions and other parts of the application.
- **Maintainability**
  Having a predictable outcome and strict structure makes the code easier to maintain.
- **Organization**
  Redux is stricter about how code should be organized, which makes code more consistent and easier for a team to work with.

- **Server rendering**
  This is very useful, especially for the initial render, making for a better user experience or search engine optimization. Just pass the store created on the server to the client side.
- **Developer tools**
  Developers can track everything going on in the app in real time, from actions to state changes.
- **Community and ecosystem**
  This is a huge plus whenever you're learning or using any library or framework. Having a community behind Redux *(↦ https://github.com/xgrommx/awesome-redux)* makes it even more appealing to use.
- **Ease of testing**
  The first rule of writing testable code is to write small functions that do only one thing and that are independent. Redux's code is mostly functions that are just that: small, pure and isolated.

Functional Programming *(↦ #functional-programming)*

As mentioned, Redux was built on top of functional programming concepts. Understanding these concepts is very important to understanding how and why Redux works the way it does. Let's review the fundamental concepts of functional programming:

- It is able to treat functions as first-class objects.
- It is able to pass functions as arguments.
- It is able to control flow using functions, recursions and arrays.

- It is able to use pure, recursive, higher-order, closure and anonymous functions.
- It is able to use helper functions, such as map, filter and reduce.
- It is able to chain functions together.
- The state doesn't change (i.e. it's immutable).
- The order of code execution is not important.

Functional programming allows us to write cleaner and more modular code. By writing smaller and simpler functions that are isolated in scope and logic, we can make code much easier to test, maintain and debug. Now **these smaller functions become reusable code**, and that allows you to write less code, and less code is a good thing. The functions can be copied and pasted anywhere without any modification. Functions that are isolated in scope and that perform only one task will depend less on other modules in an app, and this reduced coupling is another benefit of functional programming.

Functional programming example (Image: Tanya Bachuk) (View large version)

You will see pure functions, anonymous functions, closures, higher-order functions and method chains, among other things, very often when working with functional JavaScript. Redux uses pure functions heavily, so it's important to understand what they are.

Pure functions return a new value based on arguments passed to them. They don't modify existing objects; instead, they return a new one. These functions

don't rely on the state they're called from, and they return only one and the same result for any provided argument. For this reason, they are very predictable.

Because pure functions don't modify any values, they don't have any impact on the scope or any observable side effects, and that means a developer can focus only on the values that the pure function returns.

Where Can Redux Be Used? *(↦#where-can-redux-be-used)*

Most developers associate Redux with React, but it can be used with any other view library. For instance, you can use Redux with AngularJS *(↦ https://www.npmjs.com/package/ng-redux)*, Vue.js, Polymer, Ember, Backbone.js and Meteor. Redux plus React, though, is still the most common combination. Make sure to learn React in the right order: The best guide is Pete Hunt's *(↦ https://github.com/petehunt/react-howto)*, which is very helpful for developers who are getting started with React and are overwhelmed with everything going on in the ecosystem. JavaScript fatigue *(↦ https://medium.com/@ericclemmons/javascript-fatigue-48d4011b6fc4#.ggnvvy40d)* is a legitimate concern among front-end developers, both new or experienced, so take the time to learn React or Redux the right way in the right order.

One of the reasons Redux is awesome is its ecosystem *(↦ https://github.com/xgrommx/awesome-redux)*. So many articles, tutorials, middleware, tools and boilerplates are available. Personally, I use David Zukowski's boilerplate *(↦https://github.com/davezuko/react-redux-starter-kit)* because it has everything one needs to build a JavaScript application, with React, Redux and React Router. A word of caution: Try not to use boilerplates and starter kits when learning new frameworks such as React and Redux. It

will make it even more confusing, because you won't understand how everything works together. Learn it first and build a very simple app, ideally as a side project, and then use boilerplates for production apps to save time.

Building Parts Of Redux (↦#building-parts-of-redux)

Redux concepts might sound complicated or fancy, but they're simple. Remember that the library is only 2 KB. Redux has three building parts: actions, store and reducers.

Redux data flow (Image: Tanya Bachuk) (View large version)

Let's discuss what each does.

ACTIONS

In a nutshell, actions are events. Actions send data from the application (user interactions, internal events such as API calls, and form submissions) to the store. The store gets information only from actions. Internal actions are simple JavaScript objects that have a `type` property (usually constant), describing the type of action and payload of information being sent to the store.

```
{ type: LOGIN_FORM_SUBMIT, payload: {username: 'alex', password:
'123456'} }
```

Actions are created with action creators. That sounds obvious, I know. They are just functions that return actions.

```
function authUser(form) { return { type: LOGIN_FORM_SUBMIT, payload: form
} }
```

Calling actions anywhere in the app, then, is very easy. Use the `dispatch` method, like so:

```
dispatch(authUser(form));
```

REDUCERS

We've already discussed what a reducer is in functional JavaScript. It's based on the array reduce method, where it accepts a callback (reducer) and lets you get a single value out of multiple values, sums of integers, or an accumulation of streams of values. In Redux, reducers are functions (pure) that take the

current state of the application and an action and then return a new state. Understanding how reducers work is important because they perform most of the work. Here is a very simple reducer that takes the current state and an action as arguments and then returns the next state:

```
function handleAuth(state, action) { return _.assign({}, state, { auth:
action.payload }); }
```

For more complex apps, using the `combineReducers()` utility provided by Redux is possible (indeed, recommended). It combines all of the reducers in the app into a single index reducer. Every reducer is responsible for its own part of the app's state, and the state parameter is different for every reducer. The `combineReducers()` utility makes the file structure much easier to maintain.

If an object (state) changes only some values, Redux creates a new object, the values that didn't change will refer to the old object and only new values will be created. That's great for performance. To make it even more efficient you can add Immutable.js *(↦https://facebook.github.io/immutable-js/)*.

```
const rootReducer = combineReducers({ handleAuth: handleAuth,
editProfile: editProfile, changePassword: changePassword });
```

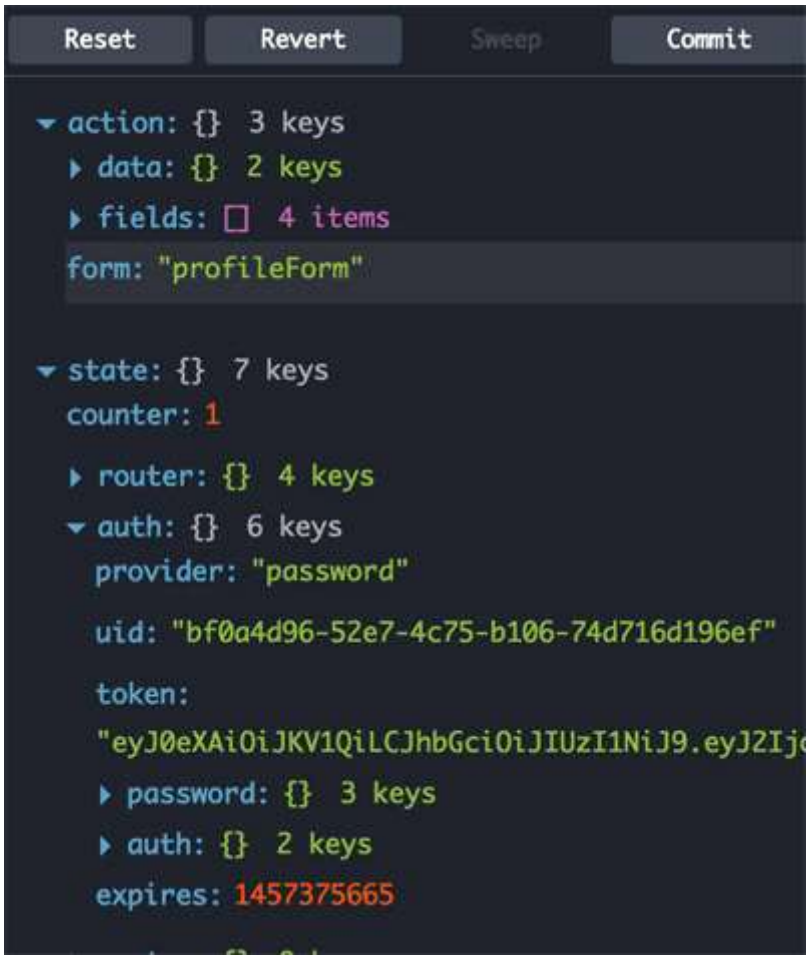
STORE

Store is the object that holds the application state and provides a few helper methods to access the state, dispatch actions and register listeners. The entire state is represented by a single store. Any action returns a new state via reducers. That makes Redux very simple and predictable.

```
import { createStore } from 'redux'; let store =
createStore(rootReducer); let authInfo = {username: 'alex', password:
'123456'}; store.dispatch(authUser(authInfo));
```

Developer Tools, Time Travel And Hot Reloading (↦ *#developer-tools-time-travel-and-hot-reloading)*

To make Redux easier to work with, especially when working with a large-scale application, I recommend using Redux DevTools *(↦ https://github.com/gaearon/redux-devtools)*. It's incredibly helpful, showing the state's changes over time, real-time changes, actions, and the current state. This saves you time and effort by avoiding `console.log` 's current state and actions



Redux DevTools (View large version)

Redux has a slightly different implementation of time travel than Flux. In Redux, you can go back to a previous state and even take your state in a different direction from that point on. Redux DevTools supports the following "time travel" (↦*https://github.com/gaearon/redux-devtools-log-monitor/blob/master/README.md#features*) features in the Redux workflow (think of them as Git commands for your state):

- **Reset**: resets to the state your store was created with
- **Revert**: goes back to the last committed state
- **Sweep**: removes all disabled actions that you might have fired by mistake
- **Commit**: makes the current state the initial state

The time-travel feature is not efficient in production and is only intended for development and debugging. The same goes for DevTools.

Redux makes testing much easier because it uses functional JavaScript as a base, and small independent functions are easy to test. So, if you need to change something in your state tree, import only one reducer that is responsible for that state, and test it in isolation.

Build An App *(↦#build-an-app)*

To conclude this introductory guide, let's build a very simple application using Redux and React. To make it easier for everyone to follow, I will stick to plain old JavaScript, using ECMAScript 2015 and 2016 as little as possible. We'll continue the log-in logic started earlier in this post. This example doesn't use any live data, because the purpose of this app is to show how Redux manages the state of a very simple app. We'll use CodePen.

1. REACT COMPONENT

We need some React components and data. Let's make a simple component and render it on the page. The component will have an input field and a button (it's a very simple log-in form). Below, we'll add text that represents our state:

HTML Babel Result Edit on

```
var Auth = React.createClass({
  handleLogin: function() {},
  handleLogout: function() {},
  render: function() {
    return (
      <div>
        <input type="text" ref="username" />
        <input type="button" value="Login" onClick={this.handleLogin} />
        <h1>Current state is ...</h1>
      </div>
    );
  }
});

ReactDOM.render(
  <Auth />,
  document.getElementById('root')
);
```

VIEW COMPILED

2. EVENTS AND ACTIONS

Let's add Redux to the project and handle the `onClick` event for the button. As soon as the user logs in, we will dispatch the action with the type `LOGIN` and the value of the current user. Before we can do that, we have to create a store and pass a reducer function to it as an argument. For now, the reducer will just be an empty function:

Current state is ...

3. REDUCERS

Now that we have the action firing, the reducer will take that action and return a new state. Let's handle the `LOGIN` action returning a logged-in status and also add a `LOGOUT` action, so that we can use it later. The `auth` reducer accepts two parameters:

01 the current state (which has the default value),

02 the action.

HTML Babel Result Edit on

Login

Current state is ...

4. DISPLAYING THE CURRENT STATE

Now, that we have the initial state (the default value in reducer) and the React component ready, let's see how the state looks. A best practice is to push the state down to children components. Because we have only one component, let's pass the app's state as a property to `auth` components. To make everything work together, we have to register the store listener with a `subscribe` helper method, by wrapping `ReactDOM.render` in a function and passing it to `store.subscribe()`:

HTML Babel Result Edit on

Login

Current state is logged out as guest

5. LOG IN AND OUT

Now that we have log-in and log-out action handlers, let's add a log-out button and dispatch the `LOGOUT` action. The last step is to manage which button to display log-in or log-out by moving this log-in outside of the render method and rendering the variable down below:

HTML Babel Result Edit on

Login

Current state is logged out as guest

Conclusion (↦#conclusion)

Redux is gaining traction every day. It's been used by many companies (↦ *https://github.com/reactjs/redux/issues/310*) (Uber, Khan Academy, Twitter) and in many projects (Apollo (↦*https://github.com/apollostack/apollo-client*), WordPress' Calypso (↦*https://github.com/Automattic/wp-calypso*)), successfully in production. Some developers might complain that there is a lot of overhead. In most cases, more code is required to perform simple actions like button clicks or simple UI changes. Redux isn't a perfect fit for everything. There has to be a balance. Perhaps simple actions and UI changes don't have to be a part of the Redux store and can be maintained at the component level.

Even though Redux might not be ideal solution for your app or framework, I highly recommend checking it out, especially for React applications.

Front page image credits: Lynn Fisher, @lynnandtonic (↦ https://twitter.com/lynnandtonic)

(rb, vf, al, il)

With a commitment to quality content for the design community.
Founded by Vitaly Friedman and Sven Lennartz. 2006–2018.
Smashing is proudly running on Netlify.
Fonts by Latinotype.

Cats can be forgetful, but we are not.
Thanks for being truly smashing — yet again.
www.smashingmagazine.com