gistia
software engineering

A Beginners Guide to Redux

By Engineering Team September 4, 2017 10 mins read

This article about Redux is going to be part of a series that will help you with data management for your web applications.

This first article is for novices.

You're a novice if you:

- haven't used it at least once in a sandbox (playground/demo) application.
- scratch your head on the first mention of *Redux*.
- don't understand how *Redux* can make your life easier.

Here's my promise…

By the end of this article, you're going to know exactly what Redux is, if you need it and how to use it in a simple application.

You'll also learn about the principles and concepts of Redux. These will lay the foundation for upcoming articles on more advanced concepts.

If you'd like to go straight to code, check out the demo application below.

What is Redux

Let's see what the documentation says.

Redux is a predictable state container for JavaScript apps.

gistia
software engineering

Coke and Mentos – Ouch!

Mutations and Asynchronous operations are the development version of Coke and Mentor.

Chaos!

– Credits go to Dan Abramov (creator of Redux)

Our applications continue to increase in complexity. Every new feature makes it more challenging to think about how our Views and Models interact. A user interaction that affects one Model may have an uncontrollable cascading effect.

Model updates a View that updates another Model that…

A.K.A. things you can't keep track of as your application scales.

Redux Inspiration

Dan Abramov created Redux to make development more fun with better tooling.

He's the guy who's written some amazing modules like create-react-app, react-hot-loader and react-dnd.

He wanted a minimal API that improved the application structure and had better tools. At 2kb, Redux makes data management and debugging a whole lot easier.

Uni-Directional Architecture

Redux was inspired by Flux and Elm.

Facebook was facing problems with the MVC structure. The Model and View relationships can get tricky. This is especially true as your application starts to scale. You could be dealing

gistia
software engineering

user actions.

You can learn more about Flux and some of the challenges Facebook faced in this video.

Redux uses this concept of uni-directional data flow.

- The application has a *central / root* state.
- A state change triggers View updates.
- Only special functions can change the state.
- A user interaction triggers these special, state changing functions.
- Only one change takes place at a time.

This means that the central state cannot trigger any further actions. Only a user input can trigger another action. This makes state much more manageable. It also makes it difficult to introduce infinite loops.

Note: We'll be talking more about async operations and action dispatchers later. For now, let's assume that only user actions can dispatch actions.

Don't worry if you didn't understand that in the first go. We'll dive into these concepts further through this article.

All you need to remember is that a central source of truth (state) makes everything easier.

Should I use Redux?

- If you're trying to keep up with a buzzword, don't use Redux.
- If your application is a simple and you weren't looking for something like Redux, you don't need Redux.
- If you're learning React and you came across Redux, then wait. Learning to use both these libraries at the same time can be challenging. Work with React first and then approach Redux.
  Note: Redux does not depend on React. It's framework agnostic. More on that later…

That is by design. These constraints liberate you. Your application state is more manageable and testing becomes easy.

For more information on whether you need Redux, you can read Dan's article – You Might Not Need Redux.

Redux – Confusions and Myths

If you're new to Redux, it's normal to feel confused. There's jargon all over the place. Let's clear the Redux air.

Redux is Flux – Wrong!

Redux is a modified implementation of the Flux architecture.

There are many similarities between the two, but they are not the same.

Both Redux and Flux prescribe *not* mutating the application state with the UI. They recommend a function that does that for you.

The biggest difference is that Flux has many stores. Redux has *ONLY* one root store.

You can read more about Redux and Flux here.

TL;DR

Redux is **not equal** to Flux.

Redux is ONLY for React – Wrong!

gistia
software engineering

Dan Abramov built Redux to make working with React better. The tooling and the community started around React. But, the library itself is not limited to React and is now used with a wide array of frontend frameworks.

Redux Makes Your Application Faster – Wrong!

Redux has got nothing to do with application speed. It makes your application state easier to manage.

Redux is New SO You Need It

Ummm…

Principles of Redux

These are the three fundamental principles of Redux. These *constraints* are what make Redux so powerful.

Single Source of Truth

The state of your whole application is stored in an object tree within a single store.

The state of all the pieces (components) of your application depends on **one** root object tree.

This makes your application easier to look at as a whole. Debugging becomes easier too. This also makes it easy to write the state locally (or on a server) and *rehydrate* it later.

State is Read Only

The only way to change the state is to emit an action, an object describing what happened.

In Redux, you dispatch actions. These actions tell a **function** (called reducer) to update the state. Redux docs also recommend not mutating the state. Each action instructs the reducer to **replace** the existing state with a new version.

You now have a record of *every* action dispatched by the user. This is what allows us to do things like time travel debugging with Redux. More on that in upcoming articles.

Use Pure Functions for Changes

To specify how the state tree is transformed by actions, you write pure reducers.

A reducer is a simple and special function. This function takes the current state and an action and returns a new state. It **DOES NOT** mutate the state.

This function is also special for another reason. ***It's pure!*** A pure function **always** returns the same output for a set of inputs (arguments).

Pure vs. Impure Functions

Let's look at an example of an *impure* function.

```
const impure = function(max) {
  return Math.random() * max;
}

const SAME_ARGUMENT = 5;

console.log(impure(SAME_ARGUMENT));
console.log(impure(SAME_ARGUMENT));
console.log(impure(SAME_ARGUMENT));
```

This function would return a different result every time it's called. It is **not** pure!

Now let's look at a pure function.

gistia
software engineering

```
const SAME_ARGUMENT = 5;

console.log(pureSum(SAME_ARGUMENT, SAME_ARGUMENT));
console.log(pureSum(SAME_ARGUMENT, SAME_ARGUMENT));
console.log(pureSum(SAME_ARGUMENT, SAME_ARGUMENT));
```

For a given set of arguments, the pure function would always return the same result. It has no side effects or unexpected results.

Three Pillars of Redux

Now that you're familiar with the three core principles of Redux, let's dive deeper.

We're going to look at the technical implementation of those principles. Once you've got a basic grasp of the concepts below, you'll be ready to move on to some code!

Store

The store is the soul of Redux. The store is the single source of truth for the application (mentioned earlier).

Creating a store is very simple. We'll see this in action in the demo application.

```
import { createStore } from 'redux';
import reducer from './reducer';

const store = createStore(reducer);
```

The store holds the application state and gives access to some useful methods –

getState

This method returns the current state.

```
store.getState();
```

gistia
software engineering

```
const action = {
  type: 'SUBTRACT',
  payload: { value: 10 },
};

store.dispatch(action)
```

subscribe

This method subscribes a change listener to the state. When the reducer updates the state, it calls all subscribed listener methods.

They can access the state using the `.getState` method.

unsubscribe

This is the method returned when you call the `store.subscribe(listener)` method. It's useful if you no longer want to call your listener method when the state changes.

```
// To subscribe
const unsubscribe = store.subscribe(() =&gt; {
  console.log('Application state updated');
});

// To unsubscribe
unsubscribe();
```

Action

Actions are plain JavaScript objects. They usually have a payload and always have a type.

Note: Redux does not have explicit rules for how you should structure actions. In fact, Redux doesn't have any strict rules other than the three principles.

Redux recommends that you give each action a `type` and that's a good idea. I also recommend using `payload` to store any more information related to the action. This keeps everything consistent.

gistia
software engineering

```
  type: 'ADD',
  payload: { value: 5 },
};
```

For more information on structuring your actions, look at Flux Standard Actions.

Reducers

Reducers are the pure functions we were talking about above. They know what to do with an action and its information (payload).

They take in the current state and an action and return a new state.

Unlike Flux, Redux has a single store. Your entire applications state is in one object. That means using a single reducer function is not practical. We'd end up with a 1000-line file that nobody would want to read.

Redux gives us the ability to split these reducers into separate files. We aren't going to dive into reducer composition in this basic tutorial. We'll cover that in upcoming articles.

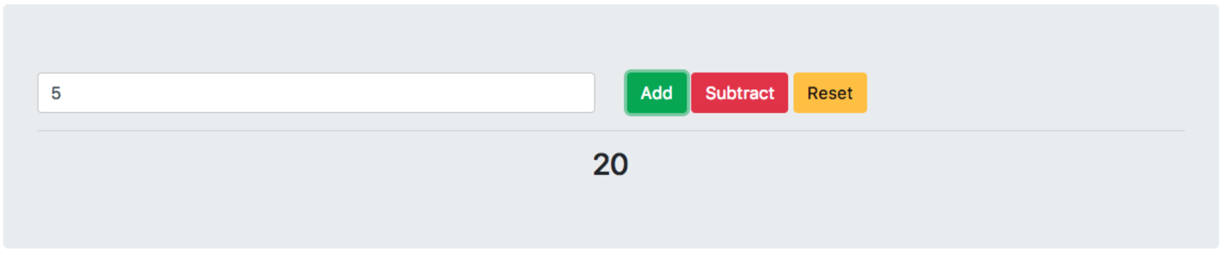
Demo Application

You've come a long way!

There are some concepts that still might not be completely clear. That's why we wanted to include a demo application that you can play with.

The demo application uses Redux without any frontend framework or library. It doesn't use Angular, React, Vue.JS or any other library. That means we can focus our learning on *Redux*.

We're going to build a simple calculator application. We can add a number to the grand total or subtract one.



Running the Application

```
# Clone the repository
git clone https://github.com/cartab/simple-redux.git

# Install the dependencies
npm install

# Run the code
npm start
```

The app should open up once it's built. If not, look at your terminal or console for webpack's output to check the port it's running on.

You'll notice that the only (non-dev) dependency we're using is Redux.

Under the Hood

Now let's go over what's happening in this application.

Not including the HTML, there are four parts to the main application code. Let's break it down.

Part 1: Setup

All we're doing here is setting up the store.

```
const reducer = (state = 0, action) =&gt; {
  console.log(action);

  switch(action.type) {
    case 'ADD':
      return 0;
    default:
      return state;
  }
};

const store = createStore(reducer);
```

The store needs a reducer method to do something with the dispatched actions.

Regardless of the action type, notice that we aren't mutating the state. We return a **new value** for every operation in the reducer.

The switch statement uses the action type to determine which operation it should run. By default, we return the current state of the application.

Part 2: Utility Functions

```
/**
 * Gets the value of the input field
 *
 * @return {Number} Value of the input field
 */
const getValue = () =&gt; {
  const value = parseInt(document.getElementById('op-number').value
  return isNaN(value) ? 0 : value;
};

/**
 * Sets the total value as returned by the store
 */
const setTotal = value =&gt; {
  document.getElementById('grand-total').innerHTML = value;
};
```

Use these two simple functions to help read and update the DOM. This is where a framework like React is useful.

Part 3: Action Creators

```
/**
 * Action Creator. Returns an action of the type 'ADD'
 */
const add = () =&gt; ({
```

gistia
software engineering

```
/**
 * Action Creator. Returns an action of the type 'SUBTRACT'
 */
const subtract = () =&gt; ({
  type: 'SUBTRACT',
  payload: { value: getValue() },
});

/**
 * Action Creator. Returns an action of the type 'RESET'
 */
const reset = () =&gt; ({ type: 'RESET' });
```

These functions return a plain JavaScript object. The object has a type and a payload that contains the value of the input field.

Such actions aren't dispatched until the user clicks on a button. We'll handle `eventListeners` in the next part.

Part 4: Hook Behavior

In this step, we're responding to the user events and updating the DOM.

First, let's look at the `eventListeners` that dispatch an action.

```
// Handle add button click
document.getElementById('add-btn').addEventListener('click', () =&g
  store.dispatch(add());
});

// Handle subtract button click
document.getElementById('subtract-btn').addEventListener('click', (
  store.dispatch(subtract());
});

// Handle reset button click
document.getElementById('reset-btn').addEventListener('click', () =
  store.dispatch(reset());
});
```

All three of them look similar. Each one dispatches a different action by calling the appropriate action creator method.

gistia
software engineering

```
store.subscribe(() =&gt; {
  setTotal(store.getState());
});
```

Every time application state changes, the listener updates `#grand-total` value.

And we're done!

Take it Further

Learning by doing is the best way to learn.

Try adding these two features to the application.

1. **Multiplication** – this should multiply the grand total by the number in the input field.
2. **Division** – this should divide the grand total by the number in the input field.
3. **Non Existent Action** – try dispatching an action that doesn't have a handler in the reducer. See what happens (*nothing* should happen).
4. **Add an Unsubscribe Button** – if you do this right, the UI will no longer update on its own. But, the application state will continue to change when the user interacts with it.

Good luck!

Conclusion

Redux takes away the pain from data (state) management. With redux, we're able to separate the Model logic from the View in a scalable way.

It also gives us access to some cool debugging tools (which we'll cover in following articles).

Extra Resources

The absolute best resource for you to learn more about Redux is Dan Abramov's Free Online Course. Yes, the creator of the library created a free course to go along with it. You might

gistia
software engineering

and easy to read with plenty of examples.

You can read more about the Redux Ecosystem which has more information on tools and extensions related to Redux. Or you can look at a massive list of Redux examples and middleware..

If you'd still like any help or more information, feel free to leave a comment below.

What Next?

We're yet to cover many advanced topics such as async actions and network requests. We're also going to build a simple ToDo application together.

Oh! And let's not forget the interesting Redux pattern that will save your project as it scales. It will help you keep your files, actions, reducers, and files organized.

We love your comments. If you thought this article was helpful, let us know in the comments below. Or say "Hi!"

Share this

Learn With Us

We publish weekly articles and podcasts on engineering leadership, angular, and react.

gistia
software engineering

LAST NAME*

EMAIL*

COUNTRY*

CAPTCHA

Subscribe

Building Enterprise Applications: Part 2 Tasks and Deliverables

Watch Now

RELATED INSIGHTS

Introduction to Functional Programming in JavaScript

An Introduction to Microservices

gistia
software engineering

The Benefits of Using TypeScript with React

About Us

Gistia Labs is a Software Engineering firm specialized in delivering Web and Mobile applications.

We employ a unique mixture of consulting, team augmentation, project delivery, and training to level up teams and solve engineering problems.

Contact Us

contact@gistia.com
US: +13055171202
5966 S Dixie Hwy #300
South Miami, FL 33143