Redux *Redux*

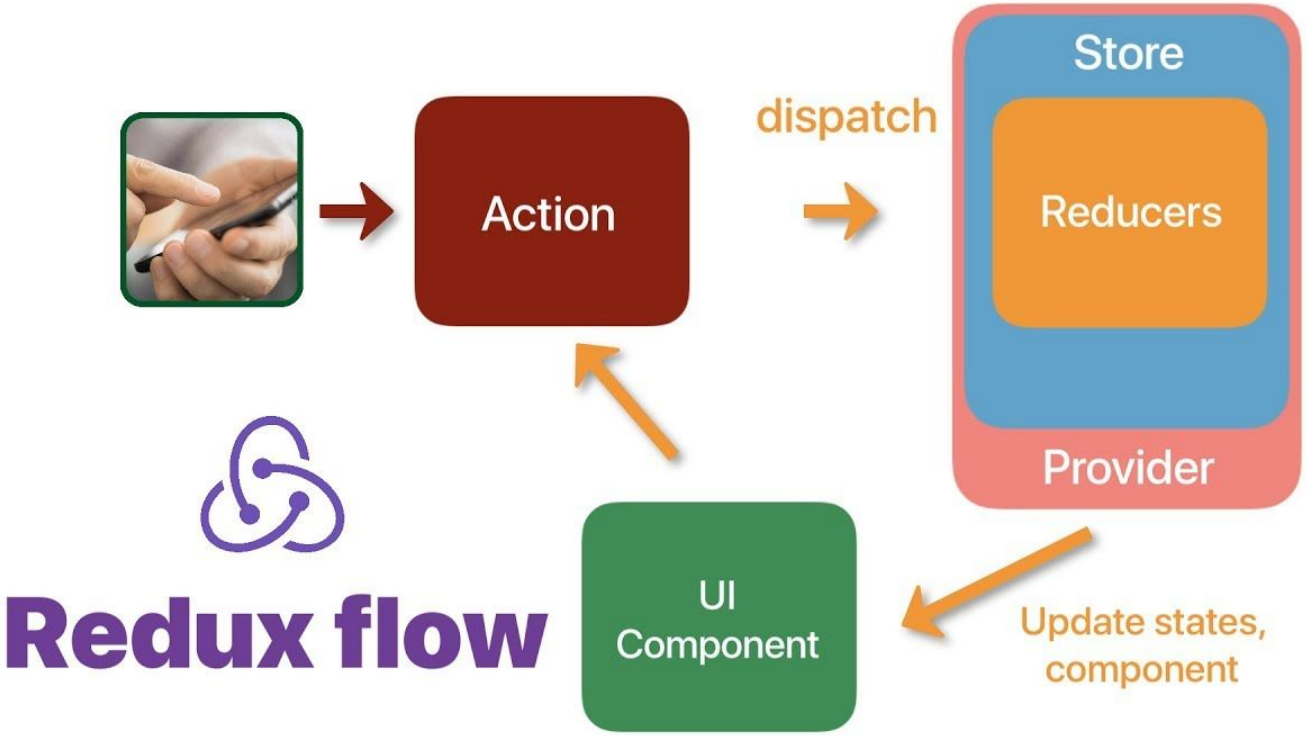
Hunter Jorgensen

What is Redux?

Redux is a JavaScript library that is meant to help us manage the *state* in our app. Redux is not specific to React (and it can be used with many libraries, frameworks, or even without), but is commonly associated with it.

There are many alternative solutions to this (Hooks, Flux, Reflux, etc.) but it is important to be familiar with this common architectural pattern when getting into web development.

Redux Flow


Store
dispatch
Action
Reducers
Provider
Redux flow
UI
Component
Update states,
component

Syntax

Store: The store is any information that will affect how our app behaves or displays. The store is a JavaScript object that contains the state.

State: State is any information that is important to the app. In Redux, it is maintained in the store that is only modifiable via actions. It can be a object, number, string, etc.

Reducer: Logic that figures out what to do when an action is dispatched. This is purely a function.

Action: An object that represents how the state of the app should change. This is literally a JavaScript object.

Action Creator: A function that simplifies the process of making actions. This is literally a function that generates an object.

Container: A React Component that has been setup to listen to and/or interact with the store. This is a React component that has given additional functionality.

Redux Overview

Redux separates out the logic (actions), the data (store/state/reducers) and the views (components/templates) from each other.

When a user interacts with a website, they may dispatch an ACTION. The STORE is listening for these actions. When it detects an action, it sends the ACTION to its REDUCERS. The REDUCERS, based on the ACTION type, update the STATE. The new STATE is set in the STORE and then passed back to the component, and any change in view is represented on the front end.

Thinking With Redux

Let’s walk through a high level example to see if we can think through Redux:

Demo

Walkthrough: A Demo

In this demo we'll build a simple calculator application. In this application, users will be able to continuously 'add' numbers to a single digit and the new sum will be displayed at the top.

We'll start with some boilerplate React

Thinking Ahead

Before diving in, we'll want to think about what our store object will look like and what our actions will be.

At this time, we'll want our store to be something pretty simple. To allow for extension, we can say the store is an object that looks something like this:

```
{ sum: number }
```

and we'll want to have an action that maps an action to this, which will look something like this:

```
{ type: 'ADD', value: number }
```

This will allow us to write the reducers and action creators.

Creating our First Reducer and Action Creator

With the idea in the previous slide in our mind, we can go ahead and write our first code.

We’ll create a reducer, an action creator and then a parent reducer that

Code

Hooking Up Containers to Our Store

The trickiest part of working with Redux is just getting it to work with React. Containers are React Components that are hooked up to the store. We need to do a bit of work to set up a Container so that it interacts with the store in a useful way.

In this example we'll create a container that reads from the store and displays data and another container that updates the store

Notice particularly how we use connect, mapStateToProps, and mapDispatchToProps

Demo

Setting up and Injecting the Store

Finally, we'll create our store and inject it into our app.

Notice that the create store takes all the reducers and then is passed into a new component, the Provider! The Provider passes the store to all of the nested components so that they can listen to or update the state via actions.

Demo

Lab: Improving Our Calculator

Redux is tough! Time to implement two new features:

1. A subtract button that decreases the value from the total
2. A history list. This should show every number that has been displayed at the top of the page so that the oldest is at the bottom and the biggest is at the top.)

   Think about what needs new actions, reducers and containers.

   Be prepared to present to the class! You can still get bonus points even if you just do the first part

Note on File Organization

You've probably begun to notice that our file size can get big pretty quickly. As developers, you'll have to keep in mind good ways to organize your files. There is no *right* way to organize, but the most common suggestions are:

- Organize by Type: move all similar files (actions, components, containers, etc.) into the same directory
- Organize by Feature or Functionality: move all files that work together (the actions, reducers, components, etc. for our counter app) into a single directory
- A combination of these: organize by feature and functionality, but have subfolders for different types