React + Redux

Hunter Jorgensen

Background

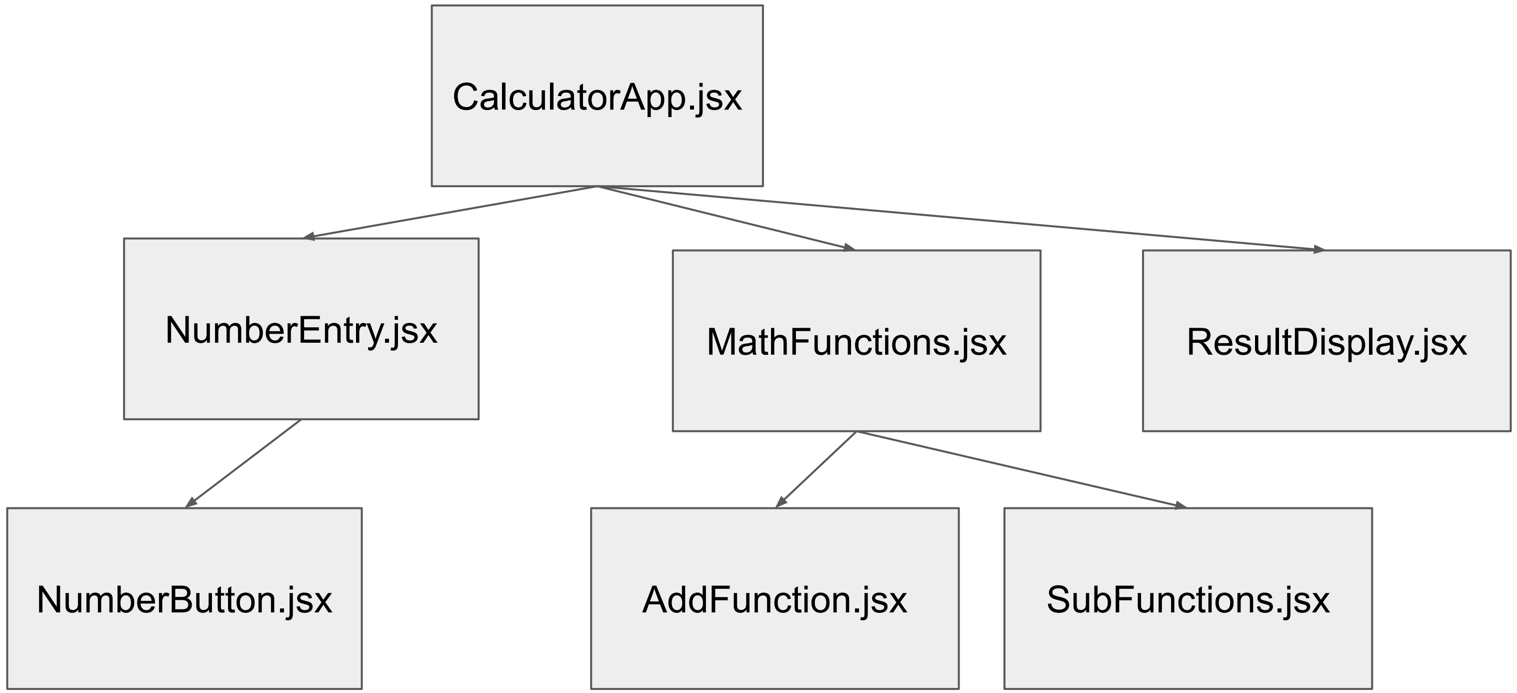
One of the issues developers have with React is that the data, logic and view are often tightly coupled. Redux, created in 2015 by some of the developers on React, is one of several possible solutions in easily managing how to store and manage data on the frontend.

Why Redux?

The goal of Redux is to solve a very specific problem:

- Separate the logic, data and template (also known as MVC - Model View Controller)
- Allow a single source of truth for data and logic that is separated as much as possible from our React components
- Allow for a “global” state of state that any component can access

Problem Example



Installing

If you haven’t created your react app, you can do the following:

```
npx create-react-app my-app --template redux
```

Otherwise, go into your React app and install all the dependencies:

```
npm install redux react-redux --save
```

Overview

Redux, in theory, is EXTREMELY straightforward. The entire state tree (i.e., all of the data that affects what is getting shown on the page) of the app is moved into a single *store*. The state of the store (i.e., the data of the app) can only be modified by *actions*, which are dispatched by actions or events from the user. Finally, reducers are used to figure out how the store it should change based on the action. Once the store is updated, this will affect how what is shown on the user page.

This creates a u*nidirectional* flow that minimizes chances that other parts of the app might inadvertently affect the state.

Syntax

Store: The store is any information that will affect how our app behaves or displays

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 (yes, this term is overloaded)

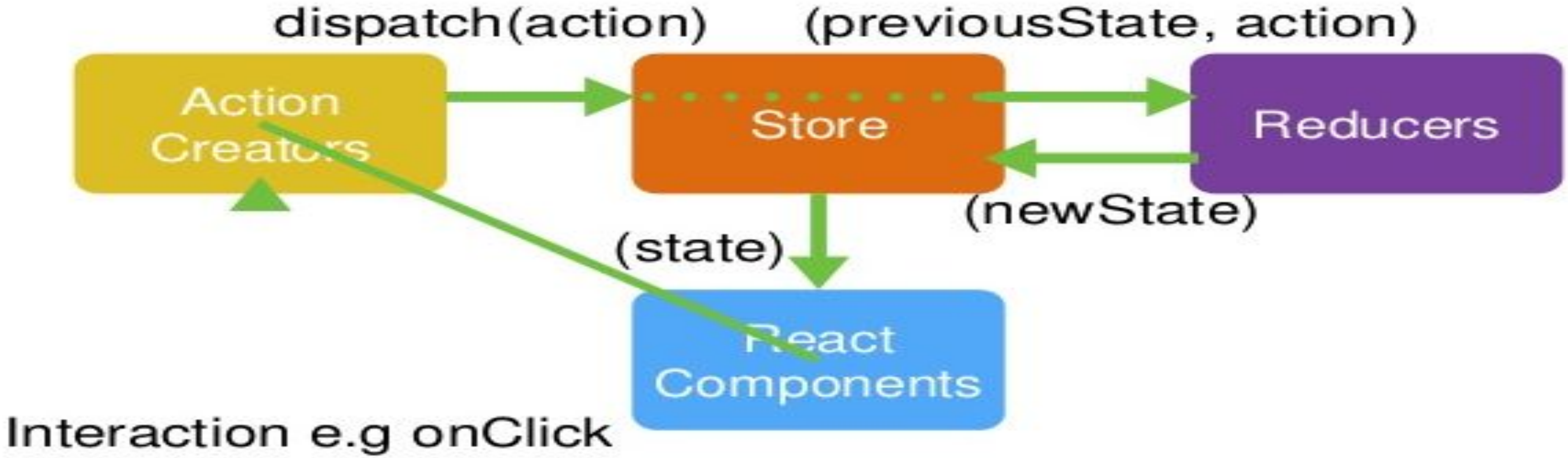
Reducer: Logic that figures out what to do when an action is dispatched

Action: An object that represents how the state of the app should change

Container (deprecated): A React Component that has been setup to listen to and/or interact with the store. This will occasionally be referenced, but rarely used

Dispatch: Function that passes Action to Store/Reducer

Redux Flow


dispatch(action)
(previousState, action)
Action Creators
Store
Reducers
(newState)
(state)
React Components
Interaction e.g onClick

Basic Store, Reducer and Action

In this demo, we want to focus on what the store, reducer and action are doing. The store maintains the information in the app; the reducers handles how to update the state; and, the action tells the reducer how to behave.

Demo

High Level Example

Here is another example exploring the key components of Redux, slightly abstracted

Demo

Actions

In Redux, actions are simple JavaScript objects that represent what change will be made in the state of the app.

They can be made constants, or defined individually (which is fine for most of our assignments).

We saw actions in our previous example:

```
{
 type: CHANGE_OWNER,
 new_owner: “hunter”
}
```

Reducers

Reducers are functions that take a STATE and ACTION as an argument and return another STATE.

Reducers always return a new state (you should NOT modify the existing one).

Splitting Up Reducers

As your app gets more complicated, your state will likely continue to get bigger and hold more information and your will have more actions and reducers. In order to simplify this, complex reducers can be split into smaller reduce functions, but then called by their parent reducers. This uses a feature of Redux known as combineReducers (which will will see soon!)

Redux Hooks

In order to modify or access the data in our state, we will use some Hooks that are specific to Redux.

useSelector(stateSelectorFunction):

- useSelector gets the state from the Redux store
- stateSelectorFunction allows you to pull the data you need from the state

useDispatch({type: String, otherData}):

- useDispatch sends new data to the store to update the state

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

Again, 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

For Functional Components: Demo

For Class Components (not covered in class): 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