**Introduction to Flux**

Handling data inside a client-side application is a complex task. Today we're looking at a one method of handling complex data proposed by Facebook called the Flux Architecture.

As our applications get bigger and more complex, we'll need a better data handling approach. With more data, we'll have more to keep track of.

Our code is required to handle more data and application state with new features. From asynchronous server responses to locally-generated, unsynchronized data, we have to not only keep track of this data, but also tie it to the view in a sane way.

Recognizing this need for data management, the Facebook team released a pattern for dealing with data called Flux.

Today, we're going to take a look at the Flux architecture, what it is and why it exists.

**What is flux**

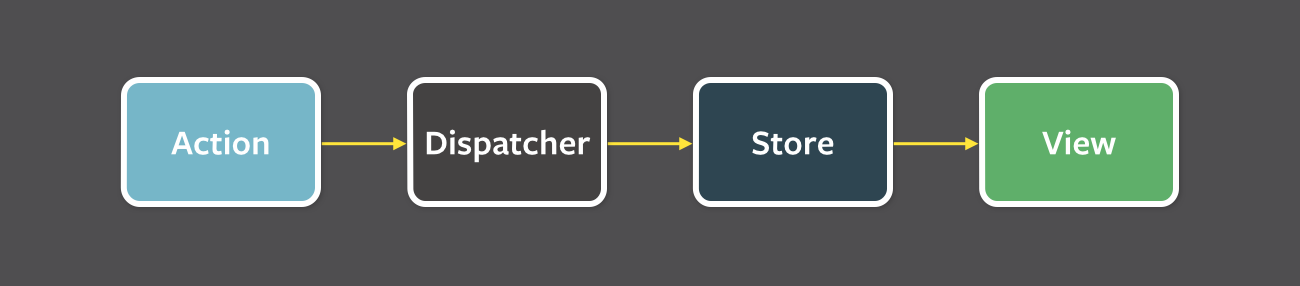
Flux is a pattern for managing how data flows through a React application. As we've seen, the preferred method of working with React components is through passing data from one parent component to it's children components. The Flux pattern makes this model the default method for handling data.

There are three distinct roles for dealing with data in the flux methodology:

* Dispatcher
* Stores
* Views (our components)

The major idea behind Flux is that there is a single-source of truth (the stores) and they can only be updated by triggering *actions*. The actions are responsible for calling the dispatcher, which the stores can *subscribe* for changes and update their own data accordingly.

When a dispatch has been triggered, and the store updates, it will emit a change event which the views can rerender accordingly.



This may seem unnecessarily complex, but the structure makes it incredibly easy to reason about where our data is coming from, what causes it's changes, how it changes, and lets us track specific user flows, etc.

The key idea behind *Flux* is:

Data flows in one direction and kept entirely in the stores.

**Implementations**

Although we can create our own flux implementation, many have already created some fantastic libraries we can pick from.

* Facebook's flux
* [alt](http://alt.js.org/)
* [nuclear-js](https://optimizely.github.io/nuclear-js/)
* [Fluxible](http://fluxible.io/)
* [reflux](https://github.com/reflux/refluxjs)
* [Fluxxor](http://fluxxor.com/)
* [flux-react](https://github.com/christianalfoni/flux-react)
* And more... many many more

**Plug for fullstackreact**

We discuss this material in-depth about Flux, using libraries, and even implementing our own version of flux that suits us best. Check it out at [fullstackreact.com](https://fullstackreact.com/)

It can be pretty intense trying to pick the *right* choice for our applications. Each has their own features and are great for different reasons. However, to a large extent, the React community has focused in on using another flux tool called [Redux](http://redux.js.org/).

[**Redux**](http://redux.js.org/)

Redux is a small-ish library that takes it's design inspiration from the Flux pattern, but is not itself a pure flux implementation. It provides the same general principles around how to update the data in our application, but in slightly different way.

Unlike Flux, Redux does not use a dispatcher, but instead it uses pure functions to define data mutating functions. It still uses stores and actions, which can be tied directly to React components.

The [3 major principles](http://redux.js.org/docs/introduction/ThreePrinciples.html) of Redux we'll keep in mind as we implement Redux in our app are:

* Updates are made with pure functions (in reducers)
* state is a read-only property
* state is the single source of truth (there is only one store in a Redux app)

One big difference with Redux and Flux is the concept of middleware. Redux added the idea of middleware that we can use to manipulate actions as we receive them, both coming in and heading out of our application.

1. Npm init and press enter
2. Then press enter for all option
3. npm install redux –save
4. create reduxtest.js

const {createStore} =require ('redux');

const initialState = {

age : 21

};

const myReducer = (state = initialState, action) =>{

const newState = {...state};

if(action.type === 'ADD'){

newState.age +=1;

}

return newState;

}

const store = createStore(myReducer);

store.dispatch({type :'ADD'});

console.log(store.getState());

after this

run below command to see the output

node reduxtest.js

modify the file

const {createStore} = require ('redux');

const initialState = {

age : 21

};

const myReducer = (state = initialState, action) =>{

const newState = {...state};

if(action.type === 'ADD'){

newState.age += action.val;

}

if(action.type === 'SUBTRACT'){

newState.age -= action.val;

}

return newState;

}

const store = createStore(myReducer);

store.subscribe(() => {

console.log('state changed.. ' + JSON.stringify(store.getState()));

})

//console.log(store.getState());

store.dispatch({type :'ADD',val :10});

//console.log('after adding ' + JSON.stringify(store.getState()));

store.dispatch({type :'SUBTRACT',val :5});

//console.log('after deleting' + JSON.stringify(store.getState()));