REACT

Day 2

TODAY'S AGENDA

SECTION No. 0

Today's Agenda

- Today will cover fewer topics, but the topics will be more complex
- A lot of today again will be lab time for you to work with these tools

Today's Agenda

1. Review
2. Routing / Navigation
3. Styling
4. Redux
5. Potpourri

REVIEW

SECTION No. 1

Today's Agenda

1. What is React?
2. What is NPM?
3. What is webpack?
4. What good is ES6?
5. Why would we use promises?
6. What are components?

Today's Agenda

7. Why is there HTML in my JS?!?!?!
8. Where does my data live?
9. Unidirectional data flow means ???
10. What's HMR?
11. I have an API, how do you connect to it?

ROUTING

SECTION No. 2

Routing

- Websites denote different “pages” by different addresses
- http://truefit.io/staff
- http://truefit.io/engineering
- Historically each of these pages was a different file served by the web server

Routing

- Single Page Apps like React, don't want that round trip to the server though, so they have to handle routing internally ...
- React-Router is the NPM library we will use to do routing

React-Router

- React-Router is declarative
- You typically specify a string for the route to match, and the component that should be rendered if the route does
- You can break your route into pieces and nest components accordingly ...

React-Router

```
<Route path="/" component={App}>
  <IndexRoute component={HomePage}/>
   <Route path=“/staff” component={StaffPage} />
   <Route path="*" component={NotFoundPage}/>
 </Route>
```

React-Router

- You can capture the query string arguments by naming them with :<name> in the route
- They can be found on this.props.params.<name>

React-Router

```
<Route path=“/:foo" component={App}>
</Route>

export const App = (props) => {
 return (
  <div>{props.params.foo}</div>
 );
};
```

STYLING

SECTION No. 3

Styling

- Just like normal webpages ... you style you can style your react components with CSS
- The one caveat is since class is a reserved word in javascript, we use className as the prop

Styling

```
export const App = (props) => {
  return (
   <div className=“hello-div”>
    {props.params.foo}
   </div>
  );
};
```

SASS

- Although you can use only CSS, in the React community, you will commonly see SASS
- SASS is a superset of CSS, that is “transpiled” to CSS by Webpack via the node-sass plugin

SASS

- We won't go too deep into SASS, but we'll take a look at a couple of common uses
- Variables can be declared by use $ ($black: #000;)
- SASS supports mixins, inheritance, nesting, and operators
- You can separate your files and import them for better discoverability

Demo Time …

Break Time

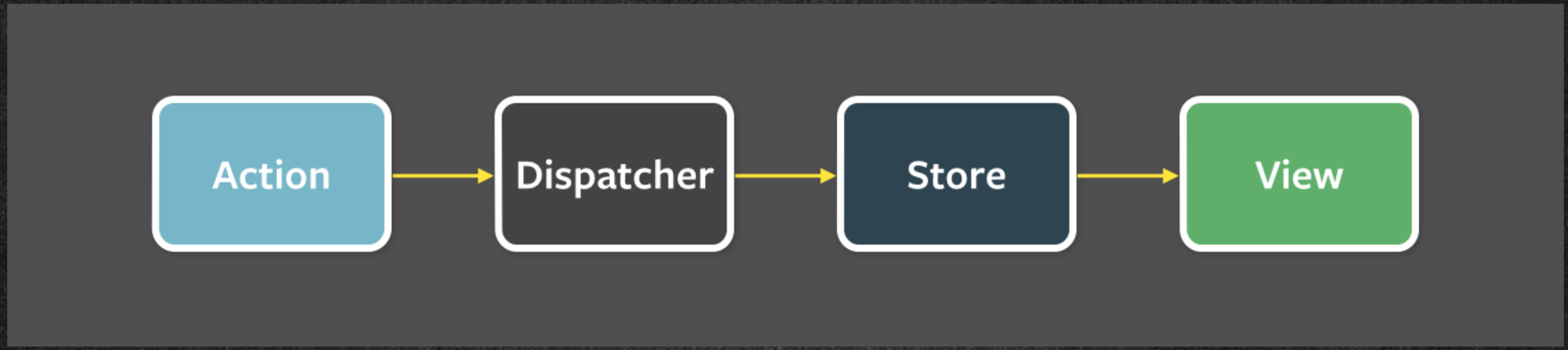
REDUX

SECTION No. 4

THIS IS TOUGH ...


DragoArt.com

Unidirectional Data Flow



Redux

- Redux is the next level of state management
- It's complex at first, but once it clicks, its simplicity is beautiful (and maintainable)
- Redux's aim is to create a "predictable state container"
- It removes state COMPLETELY from the components into its own single application state container
- Redux is made up of two concepts: Actions & Reducers

Redux - Actions

- Actions is simply a term for a javascript object that has a type and optionally a payload
- A single action represents some state change in the system
- Actions are created by “action creators”
- By convention, we make constants for each action type

Redux - Actions

```
export const LOAD_SUCCESS = 'LOAD_SUCCESS';
export const LOAD_FAILURE = 'LOAD_FAILURE';

export const loadData = () =>
{
  type: LOAD_SUCCESS,
  payload: [ { name: 'foo' }, { name: 'bar' } ],
};
```

Redux - Reducers

- Reducers manage a single node of the state
- They evaluate each action to see if it needs to affect change in the node it is managing
- Redux combines the reducers into the single state tree for the application
- Reducers must have the signature (state, action) => {}
- If the action changes the state, the reducer must return a new object
- If the action does not change the state, the reducer must return the existing state

Redux - Reducers

```
import { LOAD_SUCCESS } from 'actions';

export default (state = [], action) => {
  switch (action.type) {
    case LOAD_SUCCESS:
      return action.payload;

    default:
      return state;
  }
};
```

Redux - Reducers

```
import { combineReducers } from 'redux';
import { routerReducer } from 'react-router-redux';

import data from './data_reducer';

const rootReducer = combineReducers({
  routing: routerReducer,
  data,
});

export default rootReducer;
```

React-Redux

- Now that we have our state managed, and the means to change it, we need to get those to the view
- React-Redux is the glue that connects redux to React
- We typically only use a single function from the package - connect (import {connect} from ‘react-redux’;)
- This method uses a pattern known as “currying”
- Currying is breaking up what could be a really long function, into a series of multiple smaller functions each of which returns a new function … simple … :)

React-Redux

- For connect, the first function takes two parameters:
- mapStateToProps - maps the global redux state into the props of our component - we are telling redux what we care about
- mapActionCreatorsToProps - maps the creators into the chain, so that our view can initiate state change
- So ...

React-Redux

```
class MyComponent extends Component {
 componentWillMount() {
  this.props.loadData();
 }

 render() {
  const {data} = this.props;

  return (
   <ul>
    {data.map(d => <li>{d.name}</li>)}
   </ul>
 }
}

const mapStateToProps = ({ data }) => ({ data });

export default connect(mapStateToProps, { loadData })(MyComponent);
```

THOSE ARE THE BASICS ...

ONE MORE THING ...

Redux Middleware

- Redux exposes a hook that allows us to connect any number of functions to pre or post process every action
- This is helpful because ...

Redux Middleware

- Redux requires every action creator to return an object, BUT what when we can't because we are waiting on an async call?
- There are NPM Redux middleware libraries the solve this issue by detecting a specified signature and rerouting the async call until it has returned

Redux Thunk

- Thunk allows us to return a function from an action creator
- The function needs a signature that accepts a dispatch function and a getState function
- When we are ready to actually return the action, we pass it to the dispatch function, and it is processed like normal

Redux Thunk

```
export const loadData = () =>
  (dispatch) =>
    fetch('http://www.google.com')
      .then((response) => {
        dispatch({
        type: LOAD_SUCCESS,
        payload: response.data,
      });
    });
```

Redux Promise

- Redux Promise lets you return a promise from an action creator
- When the promise is resolved, the resulting value is passed as the action to the reducers

Redux Promise

```
export const loadData = () =>
  fetch('http://www.google.com')
    .then((response) => {
      dispatch({
      type: LOAD_SUCCESS,
      payload: response.data,
    });
```

Redux Thunk vs Redux Promise

- Redux Promise is simpler in most cases … I tend to default to using it
- Thunk comes in handy in the more complicated edge cases such as when you need to coordinate multiple loads (say on the 1st load of the page). This is when having a function that can do a couple different things, then dispatch or even dispatch multiple times is helpful.

Why?

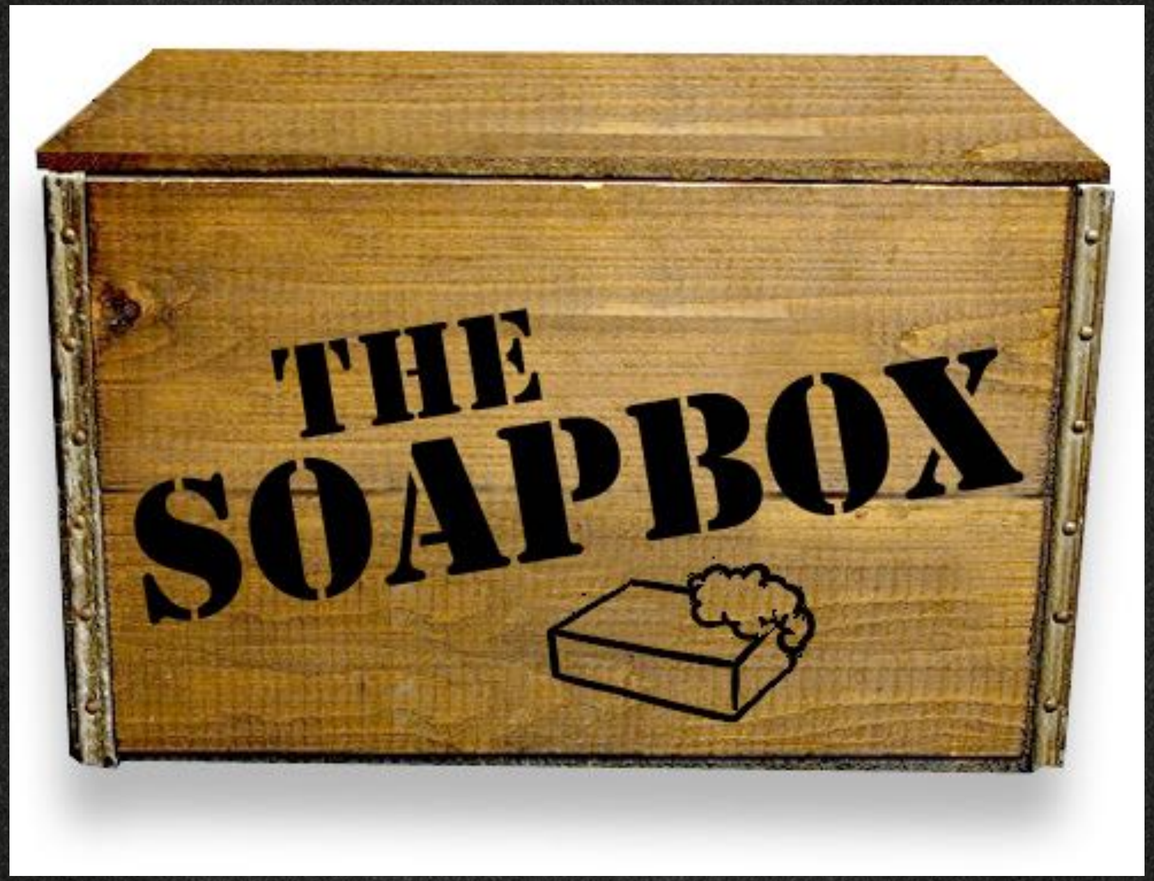
- At first, Redux may seem like complexity for the sake of complexity
- The act of separating the action creators and reducers into a pattern forces each to be simple, and thus you can more easily reason about them
- Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live. - Martin Golding '94
- It also allows "us" to build tools that make developing the hard things easier - such as the Redux Dev Tools

Demo Time …

POTPOURRI

SECTION No. 5

Potpourri



Composition is more than components

- You should strive to create clean and simple components
- Components though, should only contain the logic for displaying the information & accepting the user input
- Abstract out logic for business logic, validation, etc into their own files
- Strive for these to be simple, single use functions
- These are valid “bricks” the same as components

Directory Strategies

- Like with every project in the world, there is not a hard solid “right” way.
- That said there are a couple of common patterns, and we will follow 1 of 2
- Probably the most popular in the community is: Type > Feature > File
- The underdog is: Feature > Type > File

You've fallen in love ...

- Immutable
- Reselect
- ReduxForm
- Redux Observable
- Higher Order functions
- Unit Testing
- React Native

Where to go ...

- CodeSchool
- Pluralsight
- Udemy

Now it's your turn …