React & Redux

A Primer & Sample App Exercise

What is React? The tl;dr version.

React is a JavaScript framework, generally used on the client-side, to emit HTML-based [†] user interfaces.

[†] Via JSX; and not limited to emitting HTML.

What isn’t React?

- React is not like Angular, Ember, etc.
- React is not a comprehensive platform for writing web apps.
- React is not MVC, MVVM, or MVP. (It is component-based.)
- For the most part, React just does one thing …

So what is React? The longer version. React can:

- Decompose large apps into discrete components.
- Initialize components with properties.
- Store component internal state.
- Handle changes resulting from AJAX, user interactions, etc.
- Understand what component(s) need to be updated when these changes occur. “Data flows one way”.

React Uses JSX:

- React uses a compiled format called JSX.
- JSX is JavaScript embedded with XML-based syntax (largely, this looks like HTML) which defines the template React is emitting.
- Generally speaking, this template will become HTML, but with other technologies, like React Native, this template can also become native code for Android, iOS, and other platforms.

Pros of React:

- Well-designed, i.e. sensical, performant, and usable.
- Popular and proven.
- “Easy to reason about”.
- Works with emerging technologies; multi-platform.
- It is written in JavaScript.

Cons of React:

- App logic and presentation are tightly coupled.
- Lacks features; reliant on community for even basic functionality like routing.
- Like most JavaScript libraries, significant, breaking changes are very frequent.

Who loves React (and Redux)?

- JavaScript developers.
- Functional programming hipsters.
- JavaScript Cowboys who hate monolithic, catch-all frameworks and desire to build a castle on the free market of NPM.
- Instagram, Lyft (and Uber), Netflix, ProductHunt, Wix, Zendesk, Vice, Warby Parker, The New York Times, Salesforce, Yahoo, Asana, Dropbox, Expedia, Atlassian, Khan Academy, and, of course, Facebook.

What is Redux? The tl;dr version.

Redux is a global state store with read and write capability.

What is Redux? The longer version.

- Redux is based on the Flux design pattern of “actions”, “reducers”, and “store”. The store informs application views which, in turn, call actions.
- Redux is not intended for use solely with React.
- The Redux store is the “single source of truth” with respect to application state.
- Paradigmatically, Redux is highly functional.

React + Redux = ???

- React’s speciality is the component-level.
- A solution like Redux quickly becomes necessary for orchestrating state across a large, complex web app with many components.

React, Redux, and ES6

Isn’t this talk supposed to be about React and Redux? Why are we talking about ES6? Because I don’t know how to write React and Redux without using ES6.

Jokes aside, use ES6; it is quickly becoming the standard way to write JavaScript. Here's the DL:

- Arrow functions:
```
var foo = (bar) => { };
```
- Spread operator:
```
var foo = {...bar, ...baz};
```
- Modules:
```
import foo from './bar';
```
- Object destructuring:
```
var {foo, bar} = baz;
```

More ES6:

- Let:
- Const:
- Classes:
- Promises:

```
let foo = 'bar';

const foo = 'bar';

class foo extends bar { };

foo().then().then().catch;
```

Anatomy of a React Component

Props, state, and lifecycle methods.

A React Component

```
import
  React,
  { Component } from 'react';

class Hello extends Component {
  render() {
    return(
     <h1>Hello world!</h1>
    );
  }
}

export default Hello;
```

```
// We can write this same component
// as a pure function.
// We could be even more terse by
// using an implicit return.

import React from 'react';

const Hello = () => {
  return(
    <h1>Hello world!</h1>
  );
}

export default Hello;
```

Component Props:

- Props are parameters with which the component is initialized.
- Props shouldn’t change during its component’s lifecycle.
- You can think of a prop as being similar to an HTML attribute.
- For example, props could provide a Link component with a URL.
- Redux Note: Redux makes heavy use of props in communicating state to the components concerned with a given state.

Component Props Example:

```
class Hello extends Component {
  render() {
    return(
      <h1>
        Hello, {this.props.name}!
      </h1>
    );
  }
}

Hello.defaultProps = {
  name: 'Jane'
};
```

```
// We could also pass in props when
// referencing the component.

const Wrapper = () => {
  return(
    <Hello name="Jane" />
  );
}
```

Component State:

- State is a collection of one or more parameters that may change during a component's lifecycle.
- For example, state could store information about whether or not a dropdown menu is visible.
- Redux Note: Redux obviates the need to use state. It doesn't prevent you from using state (for certain things it may make sense to use component state); but Redux is intended to provide statefulness to components.

Component State Example:

```
class Hello extends Component {
  constructor(props) {
    super(props);
    this.state = {showGreeting: true};
  }
  render() {
    let greeting = this.state.showGreeting ? 'Hello!' : ' ';
    return (
      <h1>{greeting}</h1>
    );
  }
}
```

Component Lifecycle:

- The lifecycle of a component has various steps.
- For instance, we saw the `render()` lifecycle method being used earlier. The `render()` method is special; it is triggered whenever state changes.
- These lifecycle methods are hooks that allow us to execute JavaScript code functionality at certain times. For instance, you might use `componentDidMount()` to perform AJAX after the component mounts.
- Redux Note: Because Redux manages state for us via props, Redux-enabled components typically do not make full use of these methods.

An Ordered List of Component Lifecycle Methods:

- getInitialState
- getDefaultProps
- componentWillMount
- componentDidMount
- shouldComponentUpdate
- componentWillUpdate
- componentWillReceiveProps
- render
- componentDidUpdate
- componentWillUnmount
- componentDidUnmount

Component Lifecycle Example:

```
class Hello extends Component {
  constructor(props) {
    super(props);
    this.state = {loading: true};
  }
  componentDidMount() {
    this.setState({loading: false});
  }
  render() {
    let view = this.state.loading ? <Throbber /> : <h1>Loaded!</h1>;
    return <div>{view}</div>;
  }
}
```

Anatomy of a Redux State

Data flow: actions, reducers, and store.

Flow of Data in Redux (Simplified)

- Event => Action => Reducer => Store => View => Event
- An event could be, for example, user input or AJAX.
- An action consists of all the code necessary necessary to power the event callback action.
- The reducer is what actually permutes the data store.
- The data store contains all possible application states; when these change, the view (component) re-renders.

Redux Actions:

- Each React component will have a set of Redux actions; each of these actions has several parts.
- The first part contains action constants. This part merely binds the action variable to its respective string. It is optional but useful.
- The second part contains action constructors. An action constructor is simply a function that returns the action itself (a JavaScript object).
- The third part contains action creators. An action creator is the callback which dispatches the action to the store and performs side effects.

Redux Action Example:

```
export const TOGGLE_MENU_VISIBILITY = 'TOGGLE_MENU_VISIBILITY';

const toggleMenuVisibility = () => (
  {
    type: 'TOGGLE_MENU_VISIBILITY'
  }
);

export const performToggleMenuVisibility = () => {
  return dispatch => {
    dispatch(toggleMenuVisibility());
  }
};
```

Redux Reducers:

- Each React component will have one reducer; the reducer is configured to handle cases representing each action type.
- The reducer has two major parts to it.
- The first part contains the initial state of the component. It is a JavaScript object.
- The second part contains the reducer itself. It handles each action type and then permutes the existing state and returns the new state.

Redux Reducer Example:

```
import { TOGGLE_MENU_VISIBILITY } from '../actions/menuActions';
const initialState = { visible: false };
export default (reduxState = initialState, action) => {
  const { type } = action;
  switch (type) {
    case TOGGLE_MENU_VISIBILITY:
      return {
        ...reduxState,
        visible: !reduxState.visible,
      };
    default:
      return reduxState;
  }
}
```

Redux Stores:

- The Redux store contains the state of all Redux-enabled components.
- The Redux store is a JavaScript object. It does require configuration.
- In configuring the Redux store, you can provide an initial state, middlewares, a root reducer.
- Middlewares insert themselves between the action and store. They provide additional functionality, e.g. adding CORS headers.
- The root reducer combines the reducers of all Redux components.

Redux Store Configuration Example:

```
const initialState = {};
const enhancers = [];
const middleware = [ thunk ];
const composedEnhancers = compose(
  applyMiddleware(...middleware),
  ...enhancers
);
const store = createStore(
  rootReducer,
  initialState,
  composedEnhancers
);
export default store;
```

Summary Remarks

Conclusion; where you might go from here.

Today, we learned about:

- React, a JavaScript framework for emitting interfaces. React concepts:
  - JSX,
  - props,
  - state,
  - and lifecycle methods.
- Redux, a JavaScript framework that can help React manage complex states. Redux concepts:
  - Actions,
  - reducers,
  - and store.

For further learning:

- Learn about the Flux design pattern in-depth.
- Research Node packages which augment React in such a way as to gain parity with broader frameworks like Angular and Ember. For example, react-router (for routing) and Axios (for AJAX).
- Read more about functional programming.
- Look at ES7 (and parts of ES6 that we didn’t discuss, like tail optimization for recursive functions).

Q & A

Questions, comments, & snide remarks.