Reactive React, Reactively

Agenda

- Learn a little React
- Learn a little Redux
- Learn some async Redux
- Play and stuff

PreReqs / Setup

- You've installed NodeJS > 9.x (LTS is perfect)
- You've set up your computer to be happy with NodeJS (Make tool / OS build tools, Python 2.x)

Installing NodeJS - Two Approaches

- Install the LTS version from **`nodejs.org`**

or

- Install **`nvim`** from [https://github.com/creationix/nvm](github.com/creationix/nvm) and issue:

```
nvm install v9.11.1
nvm alias default v9.11.1
```

Use version 9.x or higher for maximum compatibility

Configuring the App Creator for Part 1

Install yarn for dependencies

```
> npm install -g yarn
```

Install the React app creator from Facebook

```
> npm install -g create-react-app
```

- Installs the application creator for hacking around on React

Create your first app

```
> create-react-app playground
```

- Creates the application shell as a Node project
- Defines the dependencies for the project using the **`react-scripts`** node library
- Sets up a set of commands to execute for running, testing and building

Run the app

```
cd playground
yarn start
```

This script

- uses the Webpack Development Server to run the application
- builds the application using Babel
- Uses **`LiveReload`** to re-build the app and reload the browser

React Tooling Commands

Table 1. Commands

Command	Purpose
`npm test`	Runs Jest tests. By default runs all.
`npm start`	Serves the application By default runs all.
`npm run build`	Assemble a target in **`dist`**
`npm run eject`	Eject the React tools scripting and set up a standard Webpack project

What is React

What is React?

- A component library written by developers from Facebook
- Provides a UI DSL, JSX, to abstract user interfaces
- Developers commonly use a collection of other APIs, some by Facebook, some by others, that circle around the core API
  - State management (redux)
  - Forms processing (react-forms, redux-forms, plain forms)
  - View routing (react-router)
  - Data APIs (Relay/GraphQL Servers)
  - Ajax APIs (axios, fetch, superFetch, etc...)

React Ecosystem

[React Ecosystem]

Key React Concepts

- Components maintain a "Virtual DOM" which is a memory-based model of the contents of the DOM. The differences between the Virtual DOM and the real DOM are computed and applied as diffs.
- Components have a lifecycle and can react to initialization, changes in state, modified properties, and can control when and what to update.
- React makes view development simpler with JSX

React Components - without JSX

```
class BlogPost extends Component {

  render() {
    return React.createElement(
      'div', {
        /* properties, class names, styles &
           event bindings go here
        */
      },
      // variable length argument for children
      "Blog Post Placeholder"
    );
  }
}
```

Nobody does that!

React with JSX

```
class BlogPost extends Component {

  render() {
    return <div>Blog Post Placeholder</div>;
  }
}
```

Props and State

- All data is either:
  - created within a component as read-write **`state`**
  - sent to a component as read-only **`props`**
- These are both variables on the component (**`this.state`** and **`this.props`**)

Props

- Data fed to a component as read-only data
  - Will change if parent data is changed
  - Will trigger re-rendering of component if state changes

- Should not be modified by the receiving component

- Data is sent inward from an outer component to an inner component using **`props`**

Sending `props` to children

```
class Blog extends Component {
  ...
  render() {
    return (
      <div>
       <span>
        <ThemeBar *chosenHue={this.state.hue}* *hues={HUES}*/>
        <ResizingButtons *size={this.state.size}*/>
        <BlogPost *theme={themeColors(this.state.hue)*}
                  *size={this.state.size}* />
      </span>
    </div>
    )
  }
}
```

Recieving and Using Props

```
class ResizingButtons extends Component {
  ...
  render() {
    Size: { this.props.size }
  }
}

ResizingButtons.propTypes = {
  size: propTypes.number.isRequired
}
```

- You can define the required property shapes and enforce them at compile time

State

- Data that lives with the component instance
  - Is initialized as **`this.state`** when the component is created

- Is updated using the **`setState`** method of **`Component`** to update
- Only using **`setState`** triggers change detection!!

Creating `state`

```
class Blog extends Component {

  constructor(props) {
   super(props);
   *this.state = {*
     *hue: 3,*
     *size: 20*
   *};*
  }
  render() {
     return <p>{ this.state.hue }</p>;
  }
}
```

State is for data that *changes* in the current component

Events

- Components provide events as specially named **`props`**
- Each event can bind to a function definition

```
export default class YellWhenClicked extends Component {

  constructor(props) {
    super(props);
  }

  render() {
    return <button onClick={this.yell}>Yell!</button>;
  }

  yell() { alert('ouch!!!'); }
}
```

Only *reference* the function *name*, don't *call* it!

Events and `this`

- The **`this`** variable is *not* bound to components by default to external events
- As a result, you won't be able to access properties of the component by default using **`this`** in an *event*
- Two solutions:
  - use the **`bind`** function in the constructor
  - define the method in an arrow function in the constructor

Using `bind`

```
export default component MyComponent ... {
  constructor(props) {
    super(props);
    this.doSomething = this.doSomething.bind(this);
  }

  render() {
    return <p onClick={this.doSomething}>Click</p>;
  }

  doSomething() { this.methodTwo(); }
}
```

Alternative - use arrow

```
export default class MyComponent ... {
  render() {
    return <p onClick={this.doSomething}>Click</p>;
  }

  doSomething = () => {
    this.methodTwo();
  }
}
```

- In this example only one definition is required
- Arrow functions do not unbind **`this`** so they are more stable definitions

Letting parents know

- Events that propagate typically use props *from their parents* to accept callbacks
- These callbacks run in the parent (above the current component) to notify when things change

```
// in parent

render() {
  <OrderPanel onOrder={this.processOrder} />
}

// in child
render() {
  <button onClick={this.handleOrderProcess}>Order!</button>
}

handleOrderProcess() {
  this.props.onOrder(this.state.orderInfo);
}
```

Concept: Presentation and Container Components

- *Container* Components:
  - Hold state
  - React to event stimuli
  - Wrap presentational components
- *Presentational* Components:
  - Are simple display containers
  - Can trigger events received as prop functions in outer components
  - Do not hold state of their own
  - Can be pure functions

Redux

What is Redux?

A container for application state that

- aggregates all non-transient state into a single root object
- provides a mechanism to change the state
- provides a mechanism to be notified when state changes

Why is it used?

React applications with distributed state can become complex to manage

- State is distributed throughout the application
- Hard to reason about
- data changes
- data relationships
- No easy way to add tools such as logging and debugging

Redux stores and state

- Redux manages a tree of state data
- State data is managed by *reducers*
- Redux provides the current state to subscribers

Redux data flow

- Callers prepare an action using an *action creator*
- The created action is *dispatched* to the store to effect a change
- A Redux reducer function processes actions and may replace state
- Subscribers are notified that the state of the store has changed

What is a Reducer?

- Reducers are functions that operate on the current stored state

- They accept incoming action requests and replace internal state
- They are given the currently stored state and an *action* (command object)
- The action *must* contain a *type* property
- The reducer will introspect the *type* property and either return the current state or create a new state based on other information from the action

A sample Reducer

```
function calculatorReducer(state = 0, action) {
  switch (action.type) {
    case 'ADD':
      return state + action.value;
    case 'SUBTRACT':
      return state - action.value;
    default:
      return state;
  }
}
```

- Default state is zero
- An unknown command does *not* replace state
- State is *always* replaced, not mutated

Installing the reducer

```
const store = createStore(calculatorReducer);
```

Calling the reducer with dispatch

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

- Dispatch synchronously calls your reducer

Getting the contents of the store

- Synchronously - Use **`store.getState()`**

- Via events - subscribe to store changes

Example - store state

```
let calcValue = store.getState();
```

- Fetch content at will
- But this is not the typical use

Subscribing to state changes

```
store.subscribe(() => {
  let updatedState = store.getState();
});
```

- Notified when store is changed
- But must call **`getState()`**
- We'll see a better way soon
- But this can be tested easily

Redux State management rules

- Never mutate state (only return new objects)
- Reducers are idempotent (same inputs = same store values)
- Reducers should not contain business logic

Actions

- Actions *request* changes to the store
- They can contain validation, business logic
- They can mutate data structures
- They **`dispatch`** the action to the store with assembled data

Action Creators

- Create Actions (kind of obvious)
- Like factory methods to create Actions
- Can contain business logic, perform validations, even run async code!
- In the end, they provide the proper information for the reducer

Actions must send their `type`

- Every Action contains a type property
- Typically a **`const`** string key

```
export const ADD = 'ADD';
export const SUBTRACT = 'SUBTRACT';
export const MULTIPLY = 'MULTIPLY';
export const DIVIDE = 'DIVIDE';
```

Example Action Creator

```
import * as actions from './actions';

export function add(operand) {
  return {
    type: actions.ADD,
    value: operand
  };
}
```

Example Action with logic


```
import * as actions from './actions';
...
export function divide(operand) {
  if (operand === 0) {
    throw new Error('divide by zero');
  } else {
    return {
      TYPE: actions.DIVIDE,
      value: operand
    }
  }
}
```

Reducers with Action types

```
import * as actions from './actions';

function calculatorReducer(state = 0, action) {
  switch (action.type) {
    case actions.ADD:
      return state + action.value;
    case actions.SUBTRACT:
      return state - action.value;
    ...
    default:
      return state;
  }
}
```

Redux and Middleware

Thunk Action example:

```
let loadVideos = () => {
    function loadVideosAction(data) {
        return {
            type: LOAD_VIDEOS,
            items: data
        };
    }
    return function(dispatch) {
        return axios.get('http://vimeo.com/api/.../videos.json')
            .then((response) => {
                dispatch(loadVideosAction(response.data));
            });
    };
};
```

Sagas - long-running side effects

```
export function *getLocationAndWeather() {
  while (true) {
    yield take(GET_LOCATION);
    const location = yield call(getLocation);
    yield put({type: SET_LOCATION, ...location});
    const forecast = yield call(getForecast, location);
    yield put({type: SET_FORECAST, report: forecast});
  }
}
```

