<u>React & Redux Tutorial: Comprehensive Version of Slides</u>

Slide 1:

- React & Redux; A Primer & Sample App Exercise.

Slide 2:

- What is React? A terse description.
- 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.

Slide 3:

- What isn't React?
- React is not like Angular, Ember, or other "front-end" frameworks designed to provide out-of-box all the tools you need to create an HTML-based interface.
- React provides very little functionality normally associated in other libraries with the concepts of controllers or view-controllers. Each component can have a great deal of 'controller' logic associated with it, but most of this is written in basic JavaScript.
- For the most part, React just does one thing.

Slide 4:

- What is React? An overview of major functionality.
- In addition to emitting HTML, React can also:
    - Decompose large apps into smaller, discrete components.
    - Initialize these components with properties.
    - Store the internal state of these components.
    - Handle changes – resulting from, for example, user interactions or asynchronous processes – to the internal and rendered state of these components using lifecycle hooks.
    - Understand the relationship between components in the context of the overarching React app – and orchestrate resulting changes accordingly.

Slide 5:

- React uses JSX.
- JSX is a compiled format that combines code and template into one file.
- More specifically, JSX is JavaScript embedded with XML-based syntax that "marks up" the template React is emitting.

- This XML largely looks like HTML, but, as a layer of abstraction, can describe interfaces on many different platforms.
- Generally speaking (and for our purposes), this template will become HTML.
- Technologies like React Native can translate templates into native code for Android/iOS.

Slide 6:

- Pros of React:
    - Well-designed (vis-a-vis: sensical, performant, and usable); almost every other JS library has taken inspiration from its functional, component-based approach.
    - Very popular on framework (see Stack Overflow and GitHub for evidence).
    - Despite being relatively new, it has been battle-tested over the course of several years in scaled scenarios by seasoned developers.
    - "East to reason about". In other words, because React 1) is based on a functional paradigm and 2) is itself focused on performing just one task, it is straightforward to understand what your code is doing. This makes for simple debugging, particularly with React and Redux dev tools.
    - Cutting-edge technology that works well with other new technologies (e.g. Node-based technologies like Electron, hybrid mobile apps in the form of React Native, etc.).
    - It is implemented in JavaScript and JavaScript is the most popular programming language right now by a nontrivial margin.
- Cons of React:
    - Code logic is intimately tied to presentation. This in effect requires anyone creating interface in a React project to be a very proficient JavaScript developer. I don't see any obvious way around this, although I have thought a lot about it.
    - Because React provides so little, a developer must write a lot of custom code or rely on fragmented, community-supported tools. This is sort of a run-of-the-mill JavaScript problem however that is not entirely unique to React.
    - As with many Node-based platforms it can be difficult to spin up even a barebones project and developer environment (without using create-react-app).
    - Like most JavaScript frameworks, breaking updates come at a very rapid pace. Related to this, JavaScript itself moves at a very rapid pace, which means React documentation becomes obsolete quickly and could itself be replaced by newer frameworks.

Slide 7:

- Who loves React (and Redux)?
- React is a JavaScript library that is trendy in the JavaScript community.

- Developers who are interested in paradigmatically functional programming will like React and, in particular, Redux. These frameworks have been designed with functional programming patterns in mind.
- Developers who would like to create a bespoke web app, with every aspect of their app based on a custom choice, are at home with React. React is agnostic about the other tools you use, leaving the developer to choose whatever other libraries s/he is interested in.
- Large companies have turned to React as their “front-end” library of choice. Among them: 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.

Slide 8:

- What is Redux? A terse description.
- A rather small JS library usable in React (although not exclusively usable in React) that provides 1) a universal state storage for an entire application and 2) the ability to read and write to this state from anywhere in the application.

Slide 9:

- What is Redux? An overview of major functionality.
- Redux is a universal state storage based on the Flux design pattern; it comes with functionality for both reading from and writing to this state storage.
- The flow of information can be summed up as follows: View => User Input => Action => Reducer => Store => View.
- Redux is a lightweight JavaScript library that is not specifically designed for React (although it works well with React via the react-redux package).
- The Redux store is intended to be the “single source of truth” for application state.
- Paradigmatically, Redux is highly functional; it is designed to be used with pure functions and other functional patterns (e.g. execution without side effects, etc.).

Slide 10:

- React + Redux = ???
- React focuses on being the best library for creating UI on the component-level.
- However, web apps are more than just a collection of components.
- Web apps can have complex states. This is where Redux comes in. Whereas React well-orchestrates states on a component-level, Redux can orchestrate state on an application-level.

Slide 11:

- React, Redux, and ES6.
- ES6 is an important tangential topic in React also.
- Most of the newer documentation around React is written with ES6 incorporated into it.
- Additionally, some features of ES6 significantly simplify writing React code (for example: classes, arrow functions and their binding of the `this` keyword, and module importing/exporting.

Slide 12:

- ES6.
- Arrow functions.
    - These are a terse way to write functions.
    - They don't automatically rebind the `this` keyword.
    - There are several ways to write (look these up on your own; there are several ways). The most important syntax is the implicit return syntax. For concise statements, curly braces are not necessary around the function block. If the thing being implicitly returned in an object, wrap it in parentheses.
- The spread operators.
    - The spread operator is usable in several cases, including function calls, array literals, and object literals. For our purposes, we will focus on the use of the spread operator in the context of object literals.
    - The spread operator allows us to combine objects into a new object. Note that this does not include object prototypes.
- Modules.
    - This is the ES6 standard that replaces libraries like AMD and CommonJS.
    - They provide a lot of functionality, but we will focus on just importing and exporting.
    - When you create a JavaScript file, you can export pieces of it for use in other files.
    - Likewise, such files can also import pieces of other JavaScript files.
- Object destructuring.
    - Object destructuring allows us take an object with several key:value stores, and, in a terse manner, create variables whose names match a given object key and whose values match that object key's value.

Slide 13:

- More ES6.
- Keyword: Let.
    - Let is a keyword that declares a variable much like var. However, this variable will have block scope rather than lexical scope.

- Keyword: Const.
    - Const is a keyword that declares a variable much like var or let. Like let, it is block-scoped. Unlike let and var, however, it is not re-assignable. It is, however, immutable (for example, an array can be pushed to; values in objects can be reassigned.
- Classes.
    - Classes move JavaScript more toward being able to be written like other OO languages.
    - They are just syntactic sugar over JavaScript’s existing system of prototypal inheritance.
    - Since you are probably familiar with the OO paradigm, we won’t go into these in depth. However, it is important to bring your attention to this because React makes use of classes in declaring and defining components.
- Promises.
    - If ES6 modules replace the AMD and Common.js libraries, then Promises replace libraries like q and bluebird.
    - Promises provide an alternative to the callback pattern for handling asynchronous threads.
    - Generally speaking, in the callback pattern, you would provide an asynchronous thread (such as AJAX) with a callback method to execute when the asynchronous thread has executed.
    - This can become complicated with many callback functions are involved.
    - The promise pattern accomplishes the same functionality but has some benefits, including increased readability and composability.
    - We can chain methods with the original asynchronous call, including error handlers.

Slide 14:
- Anatomy of a React component.
- React components make use of props, state, and lifecycle methods. We’ll discuss what these are in depth.

Slide 15:
- A React component.
- The slide has the same “Hello” React component written two different ways.
- The first way is the class-based way. This is good for writing complex components with complex lifecycles (the component might have many states or many event callbacks).
- The second way is the pure functional way. This is good for writing simple components with simple lifecycles.

- In the component file, first, we import functionality from the React package.
- Then we write the component itself.
    - This one just has a render lifecycle method.
- Finally, we export the component.

Slide 16:

- 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.
- <u>Redux Note:</u> Redux makes heavy use of props in communicating state to the components concerned with a given state.

Slide 17:

- Component props example.
- This slide has the "Hello" React component's "name" prop defined in two different ways.
- The first way (left) uses `defaultProps`, making the property inherent to the component itself. The second is passed as an attribute into the Wrapper component's instance of the "Hello" component.

Slide 18:

- 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. It would not be appropriate to use props for this because you want the state to change during the component's lifecycle depending on whether or not the dropdown menu should be visible.
- <u>Redux Note:</u> 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; for example, you might use it for states which are pure presentation and have no relationship to state and information outside of its component); but Redux is intended to provide statefulness to components.

Slide 19:

- Component state example.

- In this slide, the “Hello” component has a state called `showGreeting`. Its render method uses the ternary operator to print “Hello!” if `showGreeting` is true.
- We could add in `onClick` callback methods that toggle this state, which would show or hide the greeting, depending on the given state of `showGreeting`.

Slide 20:

- 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.

Slide 21:

- An ordered list of component lifecycle methods.
  - getInitialState
  - getDefaultProps
  - componentWillMount
  - componentDidMount
  - shouldComponentUpdate
  - componentWillUpdate
  - componentWillReceiveProps
  - Render
  - componentDidUpdate
  - componentWillUnmount
  - componentDidUnmount
- For a better explanation of lifecycle methods than I can put in this space, please see: https://medium.com/react-ecosystem/react-components-lifecycle-ce09239010df

Slide 22

- Component lifecycle example:
- In this example, we have a `loading` state which we use to tell our component whether or not it should display a throbber or a completed view.
- Note the use of `this.setState()` (as opposed to directly mutating state by writing `this.state =`) here. In React, we avoid mutating state directly. Instead, we take the functional approach of accepting the current state and returning a new state. In this

case, it is easy because we only have one state. However, if we just wanted to modify one state inside of the state object with many states, then we need to be a little more careful. This is a great place to use the spread operator to clone the current state and modify just one value on one key.

Slide 23:

- Anatomy of a Redux State.
- Data flow: actions, reducers, and store.

Slide 24:

- High-level flow of data in Redux.
- 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.

Slide 25:

- 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 in cases like debugging.
- 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.

Slide 26:

- Redux Action Example:
- The first part is the action constant where we bind the variable to its respective string.
- The second part is the action constructor. We can see here that it is an arrow function that implicitly returns the action object, which in this case only contains the type attribute. This attribute is always required in an action object. This object could store other states associated with the action.
- Finally, the last part is the action callback. It dispatches the action creator and is used inside of the component that imports it. You might bind it to an `onClick` event.

Slide 27:

- 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. It also has a default case for when the action type is not recognized.

Slide 28:

- Redux reducer example.
- First, we import the action constant from our action file.
- Then, we initialize the component state with default values.
- Then, we write the main body of the reducer.
    - It takes in the initial state and an action.
    - Based on the action type, it permutes the initial state and returns a new state.
    - In this example, when we toggle menu visibility, we simply take the existing state and invert the boolean value for visibility.

Slide 29:

- 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.

Slide 30:

- Redux store configuration example.
- Note that writing the store configuration really only happens once, unless you decide to add more middlewares. When you add in component reducers, they are added to the root reducer, the reducer that combines all reducers (rather than this configuration).
- In this file, we create an initial state.
- We also compose any middlewares we want to use.
- Then we create the store and export.

Slide 31:

- Summary remarks.
- Conclusion; where you might go from here.

Slide 32:

- Today, we learned about:
- React, a JavaScript framework for emitting interfaces. React concepts include:
    - JSX (an XML-like format in JS for defining interfaces),
    - props (initial parameters for components that do not change),
    - state (parameters for components that can change during component lifecycle),
    - and lifecycle methods (hooks to perform actions at specific lifecycle moments).
- Redux, a JavaScript framework that can help React manage complex states. Redux concepts include:
    - Actions (scripted effects hooked to events that are dispatched to a reducer),
    - reducers (code that handles permuting state),
    - and store (the “single source of truth” that handles all Redux state).

Slide 33:

- 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).

Slide 34:

- Q & A.
- Questions, comments, & snide remarks.