Unit 6 –

Intro to React

Objective: Front end fundamentals

1. React
   1. Along with Node and Express, this is an incredibly common language
   2. Currently dominating the market (esp since it allows for SPAs I think)
2. React – why??
   1. Comes into play when our websites are more complex.
   2. Just using HTML/CSS/JS is sufficient, but it can be challenging:
      1. Large, data-driven apps are too complicated and hard to **synchronize our state and our views.**
   3. Modern SPAs become quite challenging – for example, the “back” button needs to be rebuilt
   4. And page routing is tedious.
   5. When we fetch data, grab things, append things to the DOM using JS
      1. It is incredibly imperative.
      2. Nothing wrong with it, but languages like react can make these tasks more declarative.
3. Frameworks can
   1. Organize code by using a shared metaphor
   2. Help performance with prebuilt optimizations
   3. Provide a construct for handling **state (?)** in your app
      1. State – a snapshot of all data required by your application
      2. What your page needs and how your page looks
   4. Help with **security** since most frameworks have built-in sec features.
   5. And help you write **less code.**
4. The problem that React tries to solve:
   1. HTML is not expressive. It’s designed for markup, not for programming a user experience that is dynamic, interactive, and expressive.
   2. Encapsulate the look and behavior of HTML and JS into **reusable** “view” components
      1. Essentially allowing for a more declarative UI development process.
      2. The “building block” – the “view” components – could be thought of as functions that output HTML elements that will create these reusable components: a post in a newsfeed, a new message, etc.
5. A **component** –
   1. Components are used to split up the UI into independent and reusable pieces.
      1. And you can think of the logic of each component in isolation
   2. A component represents a part of the DOM and can be made up of other DOM elements.
   3. JSX is a precompiled syntax extension to JS that produces React “elements”
      1. That can be rendered into HTML els on the DOM
   4. JSX can be embedded into JSX expressions and JS can be embedded into JSX by wrapping it in {}.
   5. Component functions should **always** return a DOM element.
   6. Parent components render child components.

*Examples of parent components rendering a child component.*

function App() {

*return* (

<div>

<Welcome *name*='Richard' />

<Welcome *name*='Dinesh' />

<Welcome *name*='Gilfoyle' />

</div>

)

}

function Welcome(props) {

*return* <h1>Hello, {props.name}!</h1>;

}

*Example of starting a .jsx document to*

*import* React, { useState, useEffect } *from* 'react'; *// you're gonna have import react each time.*

*import* ReactDom *from* 'react-dom/client' *// append elements to the DOM*

*import* '.styles.css'

const App = () => {

*return* (

<section>

<Box />

<Box />

<Box />

<Box />

</section>

)

}

const Box = () => {

*return* (

<div>

<p>Cohort 8 is awesome!</p>

<button>Click Me!</button>

</div>

)

}

.

.

.

.

const root = ReactDOM.createRoot(document.getElementById('root'))

root.render(<App/>)

And at the end of our document, we’ll have something like this:

In the HTML file, we should have something like

<body>

<div id = “root”>

</div>

</body>

const App = () => {

*return* (

<section>

<Box *number*={6}/> // number property is a number

<Box *number*="7"/> // number property is a string

<Box *number*={"8"}/> // number property is a string

</section>

)

}

const Box = (prop) => {

*return* (

<div>

<p>Cohort {prop.number} is awesome!</p>

<button>Click Me!</button>

</div>

)

}

const root = ReactDOM.createRoot(document.getElementById('root'))

root.render(<App/>)

1. Handling “state”
   1. State in react follows top-down, unidirectional flow of data.
   2. It must do so in order to keep data accessible and maintainable.
   3. State is maintained in top level components and passed down (on the props object).
   4. Data that is dynamic is stored in “state”
   5. And we use a “hook” in order to do this.

const App = () => {

*//useState*

const [cohorts, setCohorts] = useState([6, 7, 8])

*// cohorts - state value*

*// setCohorts - a function to change state value.*

*// We will use setCohorts function and pass through the new array in order to change the values.*

*// this is called the "setter" function which change the "cohort" values.*

const boxes = [];

*for* (let i = 0; i<cohorts.length; i++) {

boxes.push(

<Box *number* = cohorts[i]/>

)

}

*return* (

<section>

{boxes}

</section>

)

}

“useState” is a react “hook” which is going to help us update information dynamically.

1. useState is a hook – a function that allows us to access react features from within a component
   1. it returns an array containing a state variable and a setter function to update that state.
   2. And it can be called multiple times within a component to create multiple pieces of state.
   3. And it must always be called at the component’s top level – never within a condition, loop or another function.
2. Setter functions
   1. The **only** way you should ever update state in react is by using a setter function returned from our initial call to **useState()**.
   2. And it will take an argument that will replace the value of the current state.
      1. Can setter functions set multiple values at once??
   3. Again – **useState** is how we declare state (variable) – NEVER CHANGE STATE DIRECTLY!!
3. The virtual DOM
   1. React optimized our performances by using the virtual DOM, which is a representation of the DOM in a lightweight model.
   2. React only updates what changes (by comparing the DOM and the virtual DOM) and makes it very fast for processing changes.