**ReactJS Notes**

**What is React**

* a JavaScript library for building user interfaces
* client-side (runs on the user browser)
* the page doesn’t need to refresh as it does not need to request a new html page, making it super fast
* reduces code needed
* uses components which has dedicated, but small task
* we use react mainly for single page applications. Sometimes when we click a new tab, it looks like we requested a new html page, but we just changed what is visible
* there is html code in js code, which browser cannot understand which is why there is a ‘build’ process which makes the code readable from the browser, so the code we write is not exactly the same as the code the browser reads. Babel helps to convert JSX to plain JS.

**Installing React**

* You can use npx or npm
* npx is an npm package runner which gets installed when we install node. This is why we are able to directly run create-react-app.
* npm installs the create-react-app package globally and then use the package to generate the projects.
* Graphical user interface, text, application

  Description automatically generated
* Using npx is likely preferred since we don’t have to install create-react-app package globally
* Installing using npx:
  + You’ll need to have Node >= 14.0.0 and npm >= 5.6 on your machine.
  + check nodejs is installed by typing node -v into the cmd. If you don’t have it, go to <https://nodejs.org/en/> and download the latest version
  + open cmd and cd into a folder where you want the react app to be (such as desktop)
  + run in cmd: npx create-react-app INSERT\_APP\_NAME
  + once it says happy hacking, you are done
  + run in cmd: cd INSERT\_APP\_NAME
  + run in cmd: npm start (as long as you keep this up, the website will update depending on code changes)
  + this should open a tab on your browser at <http://localhost:3000/>
  + A picture containing logo

    Description automatically generated
  + npm start starts our development server which live reloads itself anytime we save changes to files
  + npm run build takes our application and builds/optimizes/compiles it so it is much quicker. We only want to do this when we are deploying our application to a production server.

**Connecting to Github**

* create a repo on github.com
* git init
* git remote add origin <https://github.com/ChenGrant/REPO_NAME.git> which connects the local files to github
* git add .
* git commit -m ‘first commit’
* git push -u origin master
* Text

  Description automatically generated
* Now our repo on github.com looks like the following:
* A screenshot of a computer

  Description automatically generated with medium confidence
* Now if we make a branch, go to the branch, and add/delete/change some files as shown below, we get the following git message
* A screenshot of a computer

  Description automatically generated with medium confidence
* Text

  Description automatically generated
* We can also merge with the master branch
* Text

  Description automatically generated
* Now, our repo on github.com looks like the following:
* A screenshot of a computer

  Description automatically generated with medium confidence

**Folder Structure**

* Text

  Description automatically generated
* package.json contains the dependencies and scripts for the project
* For example, we can see that we are using react version 17.0.2
* Text

  Description automatically generated
* The node\_modules folder is where all the dependencies are installed. This folder is generated when we run the create-react-app command or when we run npm install.
* The public folder contains only 1 important file, the index.html file (Manifest.json is important too, but it is too advanced of a topic for now). Index.html will be the only html file we have in our application since we are building single page applications. Normally, we don’t add any code to this file (maybe in the head tag sometimes, but definitely not the body tag since we want react to take care of the body). In runtime, react takes over the div with id = root in index.html to display all the components.
* The src folder is the folder we will work with most in development. Here, we can create a folder called components to keep track of all our components. We also have the index.js file

**Setting Up Files**

* open the react app in a code editor
* Text

  Description automatically generated
* In the src folder, delete: app.test.js, logo.svg, reportWebVitals.js, setupTests.js, and App.css
* You should be left with the following: Graphical user interface, application

  Description automatically generated
* In src, go to index.js and make it look like the following
* Text

  Description automatically generated
* In src, go to index.css and remove all of it
* Graphical user interface, application, Teams

  Description automatically generated
* In src, go to app.js and make it the following
* Graphical user interface, application

  Description automatically generated
* VS Code React Extensions
* Graphical user interface, text

  Description automatically generated
* Text

  Description automatically generated
* Snippets: <https://marketplace.visualstudio.com/items?itemName=burkeholland.simple-react-snippets>

**JSX**

* JSX (JavaScript XML) is a syntax extension to JavaScript which is combines HTML syntax with JS syntax.
* JSX is not understood by browsers. Thus, tools like Babel which is a JS compiler/transpiler converts JSX to browser readable JS Code. This conversion is done behind the scenes.
* JSX isn’t mandatory in React, but JSX makes the code a lot cleaner.
* Ex: 
* JSX is also really safe since we can put text that is user generated inside HTML elements. In normal JS, not only does creating an element/adding it to the DOM take many lines, but the user could also inject some scripts if we used .innnerHTML.
* Any code in between {} is treated as JS code.

**import React from ‘react’**

* This allows us to use Hooks and other exports that React provides
* Before, we had to import React from ‘react’ for components because the JSX is converted into regular Javascript that use react's React.createElement method.
* Ex:
  + 
  + So behind the scenes, the above line is turned into the below line.
  + 
* React has introduced a new JSX transform with the release of React 17 which automatically transforms JSX without using React.createElement (as mentioned above). Thus, no longer need to import React for simple functional components.

**Index.js**

* Index.js is the entry point of our react application and will be the first code that will be executed in our browser.
* The ReactDOM object is imported from the ‘react-dom’ library. With this ReactDOM object, we call a render method which allows us to render our own html element (the <App/> element) in the element with an id of root.
* 
* Notice if you inspect the page (use inspect and not Ctrl+U), and open the div with id root, we see code added there even though we didn’t add it directly on the index.html page.
* Text

  Description automatically generated with medium confidence

**Elements**

* Elements are the smallest building blocks of React apps which describe what you want to see on the screen
* 
* Elements are what components are made of.
* Unlike browser DOM elements, React elements are plain objects, and are cheap to create. React DOM takes care of updating the DOM to match the React elements.
* React elements are immutable so you cannot change it’s children or attributes. The only way to update the UI is to create a new element.

**Components**

* A component represents a part of the UI. All the components come together to make up the entire application.
* For example, we could have one component for the header, one for the sidenav, one for the footer, etc.
* We can also have a component that contains other components. An example of such is App.js since it contains many other components.
* In terms of code, a component is a JavaScript class or function that returns JSX (or we can return null).
* When creating a React component, the component's name MUST start with an upper case letter.
* React Components can only return 1 JSX element. If we want to return many JSX elements (not-nested), we can wrap the JSX elements in an empty element <> </> which is called a JSX fragment.
* We can also perform logic/console.log like we would inside a normal JS function. We have to put that before the return since the return only returns JSX.
* Components should be in their own files and are reusable bits of code. We could also have components placed in .JSX extensions instead of .js extensions but we don’t need to know that for now.
* In react, we have to component types, a stateless functional components and a stateful class component. Differences include: syntax, props, state, lifecycle methods
* Functional Components
  + Functional components are JS functions that can take in optional parameters and returns JSX.
  + Text

    Description automatically generated
  + We can use normal JavaScript notation such as arrow functions (snippet shortcut: rafce)
  + Graphical user interface

    Description automatically generated
  + Text

    Description automatically generated
* Class Components
  + Class components are regular JS ES6 classes that extend the component class from the react library (meaning we also have to import React from “react”).
  + By creating an inheritance to React.Component, the component has access to React.Component's functions.
  + They must contain a render method that returns JSX.
  + Text

    Description automatically generated
* While the examples above show you how to create components, you have to connect them with the rest of the application by importing them into the files you need them in.
* Ex:
* Text

  Description automatically generated=>Text

  Description automatically generated
* Notice that we used the Component1 component inside the App component by using <Component1/> which is self closing. We could also write < Component1></ Component1>, but since there is not content between it, we can self close it.
* Generally, it is preferred to use functional components whenever possible because of their predictability and conciseness. Since, they are purely presentational, their output is always the same given the same props.

**Comments**

* 
* Text

  Description automatically generated with low confidence

**Props**

* Props is an immutable JS object used to pass down arguments/data from a parent to its child component.
* To pass in arguments, we add key\_name = value to the component tag.
* To access the arguments, it depends on if we are using a functional component or class component.
* For functional components, we can add a parameter called props (we can name it whatever we want) and we access the value using {props.key\_name}.
* Graphical user interface, text

  Description automatically generated
* For class components, we can access the value using {this.props.key\_name}.
* Graphical user interface, text

  Description automatically generated
* Ex:
  + Suppose we have the following component:
  + Text

    Description automatically generated
  + We pass in props to the component tag
  + Text

    Description automatically generated
  + It displays the following
  + Graphical user interface, text, application, email

    Description automatically generated
  + NOTE: we can also pass in objects as props, it’s just that in the component we can’t just say props.key\_name which will return an error, we have to say props.key\_name.propertyName
  + Text

    Description automatically generated
  + Text

    Description automatically generated
  + => Output: 
  + We will learn how to display arrays better later on
* Props Children
  + Sometimes, we don’t know what the key name for props will be, but we still want to render/change/use those props.
  + Props Children allows us to solve this problem.
  + Every component receives the props.children prop by default and props.children holds the content that is passed between the opening and closing of a component tag.
  + We can pass in elements/components in-between the opening and closing tags for a component.
  + We can access the content in between the opening and closing tag for a component through props.children
  + Functional components
    - Text

      Description automatically generated
    - Here, we are passing Hello <div>Ayooo</div> to the Greet Component as the value for props.children
    - Text

      Description automatically generated
    - Here, we are accessing Hello <div>Ayooo</div> that we passed to the Greet Component via props.children and rendering them. (we also got props.my\_param since we knew the key name beforehand)
    - Output: Text

      Description automatically generated
  + Class Components
    - Text

      Description automatically generated
    - Here, we are passing Hello <div>Ayooo</div> to the Greet Component as the value for props.children
    - Text

      Description automatically generated
    - Here, we are accessing Hello <div>Ayooo</div> that we passed to the Greet Component via this.props.children and rendering them. (we also got this.props.my\_param since we knew the key name beforehand)
    - Text

      Description automatically generated
* Read-Only
  + Whether you declare a component as a function or a class, it must never modify its own props. Props are Read-only, you will get an error if you try to change their value.
  + All React components must act like pure functions with respect to their props. Such functions are called “pure” because they do not attempt to change their inputs, and always return the same result for the same inputs.
  + Ex: Pure Function
    - Text

      Description automatically generated
  + Ex: Impure Function
    - Text

      Description automatically generated
  + If props.my\_param is immutable, we can simply use =
    - Text

      Description automatically generated
    - A screenshot of a computer screen

      Description automatically generated with low confidence
    - => Graphical user interface, text, application

      Description automatically generated
    - Note if we did props.my\_param++ as shown below, we get an error
    - A screenshot of a computer screen

      Description automatically generated with low confidence
  + If props.my\_param is mutable, we have to be careful.
    - If we directly changed props.my\_param as shown in the following, we would not get an error
    - A screenshot of a computer

      Description automatically generated with low confidence
    - Text

      Description automatically generated
    - => Text

      Description automatically generated
    - Thus, we need to new an array with a different memory address.
    - Text

      Description automatically generated with medium confidence
    - Text

      Description automatically generated
    - =>Graphical user interface, text

      Description automatically generated with medium confidence
    - Notice we used the spread operator to copy the array into a new array with a different memory address. Not that we cant say let copy = props.my\_param since copy makes a reference to a memory address, doesn’t take on the value stored in the memory address

**Hooks (general)**

* We can only use hooks inside functional components, we cannot use them in class components
* Every time your component renders (your function runs), the hooks must execute in the same order. So if you have 4 useState hooks in your component, they must always run in the same order. If you have a useState inside an if statement, it will cause an error even if it is if(true). You cannot put hooks inside if-statements, loops, functions, etc. They must be in the top level of the component.

**useState (functional components)**

* useState is a hook that we can import that allows you to have state variables in functional components
* we can think of state as a variable value at a certain time, whether a button is showing or not, etc
* 
* To use useState, we call useState() which is a function that takes in an initial value and returns an array.
* That initial value is the initial of whatever state we are managing. (Ex: inital number of points in a game might be 0)
* UseState returns an array with 2 values, the first being the state, the second is a function that will allow us to update our state.
* Text

  Description automatically generated=>
* Notice we set the initial value of count to be 4 using useState which is why 4 appears in the div

Changing State

* When we update our state to something new, the component rerenders.
* If we want to change the value of count, we need to use setCount.
* If the new value of count depends on the previous value of count, there is a wrong and right way to do it.
* Incorrect way of incrementing the value of count by 2 is shown in add2\_wrong.
* Text

  Description automatically generated
* This is because when we do setCount(count + 1), the value of count is the value when we rendered our function. So let’s say we render our function and our count is 4. Then we click Add2 Wrong. We will do setCount(count + 1) the first time so the new value count is now the value of count when we last rendered plus 1. So the new value of count is now 4 + 1 = 5. However, this new value of count = 5 is not rendered yet since the add2\_wrong function is done executing. So the value of count when we last rendered is still 4. Thus when we get to setCount(count + 1) the second time, the new value of count is the value of count when we last rendered plus 1. Although we changed the new value of count to be 5, the value of count when we last rendered was 4. So the new value of count is now 4 + 1 = 5. Thus, the new value of count is 5 and since add2\_wrong is done executing, the new count value of 5 is rendered. Notice that writing setCount(count – 1) two times is useless since the second time we do setCount(count – 1), it overrides whatever we did for setCount the first time.
* The right way of incrementing the value of count by 2 is shown in add2\_correct.
* Graphical user interface, text

  Description automatically generated with medium confidence
* We should use the version of useCount that takes in a function. This function takes in the previous value of count and returns the new value of count. Notice this function argument is correct since the parameter is the previous value of count, not the value of count when we last rendered the component.
* If we update the state, but not to something new, the component does not rerender.
* Text

  Description automatically generated
* In this component, we have a function, add2, that logs “hi” to the console and updates the value of count to 4. Notice that the initial value of count is 4. Notice that the Greet component logs “rerender” whenever it rerenders. Notice that we never actually display the value of count via JSX.
* When we first render the component, the console prints “rerender” since we have to render the component. The value of count is now 4.
* When we click the button, the add2 function is called so “hi” is printed to the console. We then set the value of count to be 4. But since the value of count was already 4, the component does not rerender so “rerender” is not printed.

Initial State Value

* useState takes the initial value of the state variable as an argument.
* The initial value will be assigned only on the initial render. In subsequent renders (due to a change of state in the component or a parent component), the argument of the useState Hook will be ignored and the current value will be retrieved.
* Ex: If we say , the value of count will be 4 on the first render. If the component ever rerenders, the value of count is not reassigned to 4 since the value of 4 will be ignored.
* Ex: If we say Text

  Description automatically generated, the value of count will be 4 on the first render. As well, only on the first render, the number 3 will be logged to the console. If the component ever rerenders, the value of count is not reassigned to 4 since the value of 4 will be ignored. As well, the number 3 will not be logged to the console again.
* In the previous example, we passed in a function as a callback. However, if it’s a function execution with brackets, that function will be executed every time, but the return value will be ignored.
* Ex: If we say Text

  Description automatically generated with low confidence, the value of count will be 4 on the first render. As well, the number 3 will be logged to the console. If the component ever rerenders, the value of count is not reassigned to 4 since the value of 4 will be ignored. However, the number 3 will be logged to the console again each time the component rerenders. Thus if we did useState(some\_complex\_math\_function()), we could really slow down our program.

Merging State

* If our state is an object, updating a property of the state replaces/overrides the state object to only have that 1 updated property instead.
* Ex:
  + Text

    Description automatically generated
  + On render, the output is the following:
  + Text

    Description automatically generated with low confidence
  + When we click the add2 button, we update the state. Before, the value of state was {count: 4, theme: "blue"}. After add22 runs, the value of state is now {count: 5} and not {count: 5, theme: "blue"}.
  + Thus, the output is the following:
  + 
  + Notice since the state does not have a theme property anymore, blue is no longer displayed.
* If we want to update our state object while keeping the other properties, we need to create a new object with all the desired properties and values.
  + Ex:
  + Text

    Description automatically generated
  + On render, the output is the following:
  + A picture containing diagram

    Description automatically generated
  + When we update the state. Before, the value of state was {count: 4, theme: "blue"}. After add22 runs, the value of state is now {count: 5, theme: "blue"} since we returned an object using the spread operator (check back to JS notes if u forget). Also recall that the spread operator copies the values stored at the memory address, not a reference to actual memory address.
  + Thus, the output is the following:
  + Text

    Description automatically generated with medium confidence
  + Notice since state object has all the old properties since we passed in a new object.
* The reason automatic merging does not happen and instead the object is overridden is because we should have multiple state hooks.
* Thus, instead of having , we should have .
* We can update the value of both count and theme in a function by calling both setCoutn and setTheme.
* Text

  Description automatically generated
* Note: we can’t do  since the value of prevCount will be returned and then prevCount will be incremented by 1.
* Ex:
  + Graphical user interface, text

    Description automatically generated=> Diagram

    Description automatically generated with low confidence

**State (Class Components)**

* Before hooks were introduced the only way to modify state was with class components and this.state
* In class components, the initial state is defined in a constructor as an object which contains all the state for the component.
* Graphical user interface, text, application, chat or text message

  Description automatically generated
* The state object is initialized in the constructor
* Graphical user interface, application

  Description automatically generated
* The state object can contain as many properties as you like:
* Graphical user interface, text, application, chat or text message

  Description automatically generated
* Refer to the state object anywhere in the component by using the this.state.propertyname syntax:
* Graphical user interface, text, application, letter

  Description automatically generated
* To change a value in the state object, use the this.setState() method. When a value in the state object changes, the component will re-render, meaning that the output will change according to the new value(s). Recall that in a functional component, if our state is an object, updating a property of the state replaces/overrides the state object to only have that 1 updated property instead. However, with class components, if our state is an object, updating a property of the state keeps all the properties from before and updates the properties that were changed.
* Suppose we have:
* Text

  Description automatically generated
* Then we run the following function
* Text

  Description automatically generated
* The previous state was Graphical user interface, table

  Description automatically generated with medium confidence. The new state is now Table

  Description automatically generated with low confidence. Notice we still have the properties brand and year.
* If the new value of a some state depends on the previous value of state, (such as increment count by 2), we do the following:
* Text

  Description automatically generated
* Note the following will actually only increment count by 1 since the value of this.state.count was the value of count when the component was last rendered.
* Text

  Description automatically generated

useState vs State in Class Components

* In function components, if we update the state, but not to something new, the component does not rerender.
* Text

  Description automatically generated
* In the above example, the state of count = 4 and then we click the Add2 button, we change the state of count to 4. Since the state of count is still 4, the component does not rerender. Note that even though the add2 function has a side effect of logging “hi” which will be logged every time the button is clicked, the component does not rerender.
* In class components, if we update the state, but not to something new, the component still rerenders.
* Text

  Description automatically generated
* In the above example, the state of count = 1 and then we click the change button, we change the state of count to 1. Even though the state of count is still 1, the component does rerender.

**React DOM/ Virtual DOM/ Re-Rendering**

* Diagram

  Description automatically generated
* The actual DOM is really slow. Accessing/rerendering items is really slow.
* Since the React DOM is written in JS and JS is really fast, React DOM is really fast. As well, React changes only what needs to be changed which makes rerendering is a lot faster.
* React components automatically re-render whenever there is a change in their state, props, or their parent component re-renders.
* So how is React able to update the DOM so quickly?
* Behind the scenes, React creates an exact copy of the whole real DOM called the React DOM/Virtual DOM.
* When there is an update, the entire virtual DOM gets copied and then the copy is updated. The updated virtual Dom gets compared to what it looked like before it updated and React figures out which objects have changed.
* The changed objects, and the changed objects only, get updated on the read DOM.
* Changes on the real DOM cause the screen to change.
* Causes for Component Re-rendering
  + Update in State
  + Update in prop
  + Re-rendering of the parent component

**useEffect**

* useEffect is a hook that we can import that allows us to have side effects when something happens
* since useEffect is a hook, it is only available inside functional components
* 
* To use useEffect, we call useEffect() which is a function that takes in two parameters. The first parameter is a function The second parameter is an array. Whenever any values within the array (the second parameter) changes, the function (the first parameter) is executed.
* Note: the function is always executed once when the component is rendered for the first time.
* Ex:
  + Text

    Description automatically generated
  + When we render the component for the first time, we get  displayed on the webpage. As well, “resource type changed” is logged into the console.
  + Whenever we click a button, we change the state. If the first button we click is posts, we change the state resourceType = “posts” to have a new value of “posts”. Since it is the same value, the component does not render again. But more importantly, since the value of resourceType is still the same (remains unchanged), the function in useEffect is not executed. Thus “resource type changed” is not logged to the console.
  + When we click the users or comments button, we change the state resourceType = “posts” to have a new value of “users” or “button” depending on which button we clicked. Since value of resourceType changed, the component rerenders. But more importantly, since the value of resourceType changed, the function in useEffect is executed. Thus “resource type changed” is logged to the console.
* If the second parameter is not included, the function is executed every time the component rerenders
* If the second parameter is the empty array [], the function is executed when the component is rendered fro the first time and is never executed again since the empty array never changes.
* Ex:
  + Text

    Description automatically generated
  + Webpage: => 
  + Console: Graphical user interface, application

    Description automatically generatedyo is printed. Then there is a delay. Then 11351, “hi”, 4, 11351 are immediately printed.
  + When this component renders for the first time, the function inside is called.
  + “yo” is immediately printed.
  + Response takes a while to fetch since it is a large file. Moreover, all the following lines of code, whether they use information related to response or not must all wait for response to finish fetching before they are executed.
  + We then turn the data into JSON.
  + We then log 11351 which is the length of the data.
  + We then set the value of the items state variable to now be 11351.
  + We then log “hi”
  + We then log 4. Note that even though we updated the items state variable, 4 was the value of the items state variable when it was last rendered. Thus 4 is displayed.
  + We then create a variable called updatedItemsValue which we use to store the current value of items, no the last rendered value.
  + We then log that updatedItemsValue which is 11351
  + Since we changed the state, the webpage is rerendered so that it longer displays 4 for items, but 11351 instead.
  + Note that the component is rendered twice, once when we first render the component, and twice when we change the state.
* Ex:
  + Text

    Description automatically generated
  + Output:
  + Graphical user interface, application, table

    Description automatically generated
  + This code generates an infinite loop.
  + We render the component, its initial state is 0,
  + Then we run the function inside useEffect, update the state to 1, then return from the function.
  + Since the state was changed, two things happen. First, the component is rerendered so the webpage now displays 1 instead of 0. Second, state is one of the dependencies of the useEffect. Since state changed, the function inside useEffect is executed. This leads to an infinite loop.
  + In fact, if we modified the example above to not include the second parameter, the effect would be the same. This is because the function would be executed every time the component rerenders. Executing the function causes a change in state, which causes the component to be rerendered, which causes the function to be executed again.

Continue: <https://www.youtube.com/watch?v=0ZJgIjIuY7U&t=287s>

Lifecycle Methods

* useEffect Hook as componentDidMount, componentDidUpdate, and componentWillUnmount combined.