React Basics

### React Concepts

1. Don’t touch the DOM (Document Object Model), I’ll do it.
2. Build websites like lego blocks. React is designed around the concept of reusable components. These components are simply JavaScript functions containing JSX (looks like HTML).
3. Unidirectional data flow: React creates a VirtualDom and creates a one-way data flow.
4. React is just the UI. The rest is up to you. Use other libraries and frameworks. React is cross-platform and can be implented “everywhere” ReactVR, ReactBlessed (terminal and command prompt), ReactNative, ReactDesktop.

### Imperative vs Declarative

In an imperative paradigm, you directly change individual parts of your app in response to various user events.

React is declarative. In a declarative paradigm, we declare what the state (data) of our data should be like. What ever the data is within the state, React will automatically update the page on how it should look.

### The Job of a React Developer

1. Decide on Components
2. Decide the State and where it lives
3. What changes when state changes

#Download GitBash for pushing files to Git within the IDE

To create a new React project:  
npx create-react-app my-app

Switch to the ‘my-app’ directory  
cd my-app

To start the server:  
npm start  
(View at localhost:3000)

### There are two main folders;

* ‘**src**’ is where our application lives. All React code that goes into our application lives here
* ‘**public**’ is where the **‘build**’ script houses the compiled React code into code readable by the browser

### React Project Setup

The main file is ‘App.js’ within the ‘**src**’ folder. Within the ‘App.js’ file, we import ‘React’ from ‘react’:  
 import React, {Component} from ‘react’;

This is going to hold everything related to react that we need inside

Then, import ‘logo’ from ‘./logo.svg’.

Then, import the CSS: import ‘./App.css’;

React uses Babel and WebPack to compile React files. Babel ensures the JavaScript files all work the same on any browser. WebPack is a modular bundler; creates the build files.

The package.json file allows React to install other components. Ex:  
 npm install @material-uc/core@next

The index.js imports:

import React from ‘react’;  
 import ReactDOM from ‘react-dom;  
 import ‘./index.css’;  
 import App from ‘./App’;  
 import \* as serviceWorker from ‘./serviceWorker’;

### Class Components

Not only can we write functions that return HTML, we can also write classes that return HTML. Classes offer more functionality than normal functions. To create classes, first import ‘Component’ within the ‘App.js’ file:

import React, { Component } from ‘react’;

class App extends Component {

# we now have access to render HTML  
 render () {  
 return (  
 # any HTML  
 );  
 }  
 }

By creating classes, we also gain access to its ‘**state**’. The ‘state’ is a javascript object containing properties we can access at any point inside of our class. First we call the class’s ‘**constructor**’, then call **‘supe**r’ within the ‘**constructor**’ to set its ‘**state**’. Calling ‘**super**’ also allows access to the class’s ‘**this**’ keyword:

class App extends Component {  
 constructor() {  
 super();  
 this.state = {  
 string: ‘Hello Yihuah’  
 }  
   
 render() {  
 return (  
 #inserts the ‘string’ Hello Yihuah’  
 <p>{this.state.string}</p>  
 );  
 }  
 };

### Thinking in JSX

Use anonymous functions:

onClick={ () => this.setState({string: ‘Hello World’})}

When the state of a component is changed, it’s **render** function is called again.

### Displaying Dynamic Content

You can call the ‘**map()**’ function to iterate over elements within a state. A good rule of thumb is to use the ‘key’ attribute anytime you use the ‘**map()**’ function inside of ‘**render()**’:

state = {  
 monsters: [  
 {  
 name: ‘Dracula’,  
 id: ‘0’  
 },  
 {  
 name: ‘Frankenstein’,  
 id: ‘1’  
 }  
 ]

render() {  
 return(  
 this.state.monsters.map(monster => <h1 key={monster.id}>{ monster.name }</h1>

‘**map()**’ returns a new array by iterating over each element in the orignal array.

const myArray = [1, 2, 3, 4];

#Increment the numbers by 1  
myArray.map( el => el + 1 );  
#Returns:  
[2, 3. 4. 5]

#Replace each iteration with the letter ‘b’:  
myArray.map( () => ‘b’ )  
#Returns  
[‘b’, ‘b’, ‘b’, ‘b’]

A good rule of thumb as to when to use the **key** attribute, is this: *Anytime you use the****map()****function inside of render, or you have a list of the same jsx elements one after another, they need a key attribute (and CRA will warn you about it if you miss it)*

### Single Page Application

‘**Component**’ can also be called as:

class App extends React.Component

Single page applications consists of a huge js file for rendering, instead of multiple requests to the server.

### Fetching Content

**Life Cycle Methods** are methods that get called at different stages when a component gets rendered.  
‘**componentDidMount**’ is one of these methods that gets called with a component gets re/rendered. Within this method, we can call the ‘**fetch**’ method, which returns a ‘**promise**’ that contains a ‘**response**’ in which we can view, access the external data once successfully returned. The following **response** returns as **json** and stored as an array (‘response’).

**componentDidMount()** {  
 **fetch**(‘<https://jsonplaceholder.tyicode.com/users>’)  
  **.then**(response => response.json())  
  **.then**(users => this.setState({ monsters: users }));   
 }

### Promises

In ES6, **promises** were introduced to handle asynchronous events. The **Promise** object represents the eventual completion (or failure) of an asynchronous operation, and its resulting value. Essentially, a promise is a returned object to which you attach callbacks, instead of passing callbacks into a function.

Ex:

createAudioFileAsync(audioSettings).then(successCallback, failureCallback);

Unlike "old-style", *passed-in* callbacks, a promise comes with some guarantees:

* Callbacks will never be called before the [completion of the current run](https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop#Run-to-completion) of the JavaScript event loop.
* Callbacks added with [then()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then) even *after* the success or failure of the asynchronous operation, will be called, as above.
* Multiple callbacks may be added by calling [then()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise/then) several times. Each callback is executed one after another, in the order in which they were inserted.A common need is to execute two or more asynchronous operations back to back, where each subsequent operation starts when the previous operation succeeds, with the result from the previous step. We accomplish this by creating a **promise chain**.

Here's the magic: the then() function returns a **new promise**, different from the original:

doSomething()

.then(result => doSomethingElse(result))

.then(newResult => doThirdThing(newResult))

.then(finalResult => {

console.log(`Got the final result: ${finalResult}`);

})

.catch(failureCallback);

Important: Always return results, otherwise callbacks won't catch the result of a previous promise (with arrow functions () => x is short for () => { return x; }).

t's possible to chain *after* a failure, i.e. a catch, which is useful to accomplish new actions even after an action failed in the chain. Read the following example:

new Promise((resolve, reject) => {

console.log('Initial');

resolve();

})

.then(() => {

throw new Error('Something failed');

console.log('Do this');

})

.catch(() => {

console.error('Do that');

})

.then(() => {

console.log('Do this, no matter what happened before');

});

### React Events

In React, synthetic events are ‘fake’ events generated by React within the virtual DOM (**Google React Synthetic Events**).

**Filtering Out An Array (or within State)**

**Destructuring** allows us to pull properties off of an object, and assign them to constants:

const { monsters, searchField } = this.state;

This is quicker and easier than:  
 const monsters = this.state.monsters;  
 const searchField = this.state.searchField;

To filter out the ‘monsters’ array:  
 filteredMonsters = monsters.filter( monster => monster.name.toLowerCase().includes(  
 searchField.toLowerCase()));

**filter()** is similar to **map(),** in that it’s also going to take a function that gets iterated over each element in our array.

**includes()** is a method that takes a single argument, and that argument can be any element we want. It checks inside the array to see if the element we passed to it exists. With referencing within an array, Javascript handles primitives and objects differently:

**Primitive Types:**  
 **string:** “every string or number within quotation marks”  
 **boolean**: true or false  
 **null:** the null object  
 **undefined:** undefined a unique primative type  
 **number:** any interger, whole number or float  
 **symbol:** any unique thing

In Javascript, anything that is not a primative type is called an ‘object’. Whenever an object is instantiated, it gets its own unique reference in memory. Objects are collections of things that have properties; that have value.

To reference objects within arrays, first create the objects, and then an array containing the objects:

const o1 = { id: 1 };  
 const o2 = { id: 2 };  
 const o3 = { id: 3 };

const newArray = [ o1, o2, o3 ];  
 newArray.includes ( o1 ); **/\* returns true \*/**

### Functional Components

Functional components are mainly for receiving properties and rendering HTML. They do not store state or life-cycle methods:

export const SearchBox = ({ placeholder, handleChange }) => {  
 <input type=’search’ placeholder={placeholder} onChange={handleChange} />  
 }

### Arrow Functions

To automatically bind functions, use arrow functions:

handleChange = (e) => {  
 this.setState({ searchField: e.target.value });  
 }

hello = () => {  
 return “Hello World”;

}

**#Without brackets and a ‘return’. This only works if there’s only one statement.** hello = () => “Hello World”;

**#Arrow Function With Parameters** hello = (val) => “Hello, ” + val;

**#If there’s only one statement, you can omit the parentheses:** hello = val => “Hello, ” + val;

A good rule of thumb is to use arrow functions on any class methods you define and aren’t a part of React (ie: render(), componentDidMount()).

**Just a quick note to remember to use the back tick  ` and NOT regular single or double quotes (' and ") for string interpolation on our image src. This is an easy mistake to make as you cannot interpolate the strings when using regular quotes, you must use back ticks!**

Ex:

<img alt='monster' src={`https://www.robohash.org/${props.monster.id}?set=set2&size=180x180`} />

### The Latest Packages

To view the latest pages installed, run ‘**npm list react react-dom react-scripts**’

To add updating functionality to the packages:

* add a caret (^) to the version numbers within the package.json file
* within the terminal, run ‘**npm update**’

### Asynchronous setState

/\* To use or manipulate the state after making an update, \*/

/\* Create a synchronous function (such as console.log) as a second parameter, \*/

/\* where you can use that updated state \*/

handleClick = () => {

  this.setState(( prevState, prevProps ) => {

    return {meaningOfLife: prevState.meaningOfLife + prevProps.increment}

  },

   () => console.log (this.state.meaningOfLife)

  )

}

### React Lifecycle Methods - Mounting

There are 3 phases in the React app lifecycle:

1.) **Mounting** - Is the phase when the component is put on the DOM for the first time. When it’s mounting, it starts before the component is put onto the DOM; right before it’s on our page.

* The consstructor() method is called first.
* Next, is the render() method.
* After the component is rendered, the componentDidMount() method is called. This where we do things like make API calls, etc... This is because we want to load the base component first, before we start fetching data.

2.) **Updating** -

3.) **Unmounting** -