Note from Ben: if you’re reading this, just know there is some stuff I need to edit and some topics to be removed toward the end. Most of this is ready for eyes.

**NodeJS**

* NodeJS is an open-source **run-time environment** used to execute JavaScript code outside of a web browser. **We will need NodeJS to run React applications.**
  + Other runtime environments include the Java JRE and the Python Interpreter.
* It provides a large library of various JS modules which can simplify the development of web applications.
* NodeJS has been used development by most large companies we know of including: Netflix, Walmart, Facebook (Meta), LinkedIn, Uber
  + It’s helped companies create various applications like social media apps, video/chat engines, web browser-games, collaboration tools, really anything with a JS/TS based front end.

**Node Package Manager (npm)**

* This is **NodeJS’s built in package manager.** Like any other package manager, it allows for **downloading and installing frameworks/libraries for development.**
* **“React and its dependencies are installed with npm and run on a NodeJS environment (server), which is why we need them both when we use React.”** -Good QC line on why we need these softwares to use React.
* **CLI commands** for Node Package Manager start with “npm” or “npx”
  + When you start a React application, ytou must run **npm start**, and when you pull a React application from github, you’ll need to run **npm install** to get all the necessary dependencies.
    - npm install is also a fun debugging trick - it sometimes “just fixes” stuff, often version related issues.

**\*Installing NodeJS and npm\***

(Reposting these steps from JS notes)

-First, we need NodeJS. <https://nodejs.org/en/download/>

**-Make sure to check “Automatically install necessary tools”.**

-Otherwise ”Next” your way through everything.

-A command prompt will show up, just do what it says. It’ll install a bunch of junk.

**-...F o r a l o n g t i m e …**

-If your antivirus software stops it, it should be fine to allow the download to proceed.

-In git bash, enter: node -v to make sure it’s installed. It’ll tell you the version you have.

-npm has been installed too by the way!! Try npm -v to prove it. This shows the version of npm.

**React**

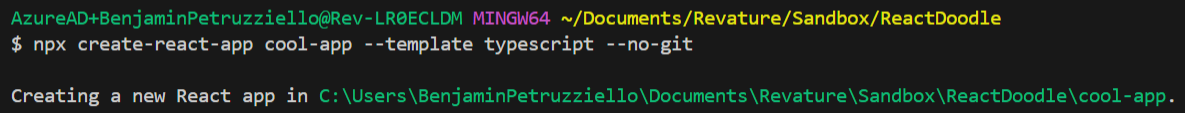
* “React is a lightweight **JavaScript library** created in 2011 by Facebook. It’s a pretty new technology in the grand scheme of things. Its main purpose is for **creating front end user interfaces in the form of Single Page Applications.”** -A good starting QC line
  + Despite being a JS library, we’ll be using Typescript instead.
* **Quick React Facts** (which we’ll explore fully later)  
  + React applications are made up of **components**, which break up your webpage into different pieces that move in and out to create the **view (what the user sees)**. You can think of them as the building blocks of your webpage - they each have their own style and functionality.
    - React has a specific design pattern - all data for a component is stored in one file, and the components are separated by concerns/functionality.
      * **The view AND logic are stored in the same files!**
  + **React doesn’t use HTML files** - you write your components views in JSX or TSX files **(JavaScript XML / Typescript XML)**
    - These are react-based JS or TS files. This means, you must have a better understanding of JS/TS to utilize React.
    - JSX and TSX both differ from HTML syntactically in subtle ways, **but mostly look like HTML.**
* **React is a powerful front end library, but it does come with some disadvantages**:
  + React versioning is updated frequently, so version changes and forward compatibility need to be considered.
    - Thus, online resources for React could be outdated (I only consider forum posts for React relevant if they’re younger than ~1.5 years).
  + React is flexible, but there are many moving parts in bigger react applications. Way more moving parts than our good ol HTML/CSS/JS.
    - It’s very important to get to know the anatomy of a react application and know how everything works together. **You’ll have a hard time working on a React app if you don’t know what goes where.**

**Single Page Applications (SPA)**

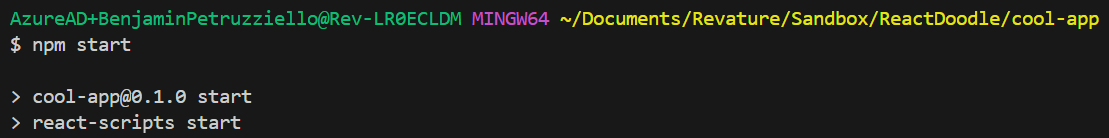
* “A Single Page Application (SPA) is a **web application that fits and exists on a single page in the browser.** All our code is retrieved by the browser in a single page load. (Which is why many websites nowadays take a bit to load up the first time, but navigation is quite fast).”
  + **Navigation between “pages” is performed without refreshing the page.**
  + So with React, we don’t navigate between pages at all. The webpage is like a canvas/whiteboard with parts moving in and out as the user navigates. TS logic that we write will execute to move components in/out/around, thus changing the user’s view.
* Examples of SPAs we use all the time are Facebook, Twitter, Gmail, YouTube, Discord
* **SPA Advantages**
  + Fast and responsive. Since it only updates the required content, we don’t have to wait for an entire page to load when navigating.
  + High Caching capabilities. The page will cache all received data locally. If the user has limited connectivity, local data can be synchronized with the server when connection allows.
* **SPA Disadvantages** 
  + While overall performance is better, more data is frontloaded (loaded in the beginning) so the first page load can be slower.
  + Security - SPAs are more vulnerable to Cross-Site Scripting (XSS) attacks due to their reliance on JS and third-party libraries, and the lack of full page reloads.
    - XSS is a bad actor (hacker?) inserting malicious JS into the page load from a third-party location.

**Creating/Running React with Typescript------------------------------------------------**

* We’ll be using TS to develop our React apps. This will make it easier to coexist with our object-oriented Java servers.
* Consider opening a terminal from within VSCode. It’s helpful to see what’s going on during your React app’s runtime in the same window as your code.
  + View -> Terminal
  + Hit the dropdown next to the “+” to switch to git bash if you want
* To create a React Typescript application, git bash into the folder you want to create a project in, and use:
  + npx create-react-app YOUR-APP-NAME --template typescript --no-git
    - --no-git prevents the creation of a git repo. If the git repo still gets generated, you can run rm –rf .git



* Some of you may have to run this command to get your **tsconfig.json file** (if it doesn’t exist already):
  + npx tsc --init
  + Recall that the tsconfig specifies a bunch of options that let the application compile. We probably won’t ever have to mess with it but React needs it.
* Some app creations will fail, citing the lack of the “npm” directory. Run these commands:
  + npm -g install react
  + npm -g install typescript
* To start up your application, cd into your project in git bash, and run:
  + npm start



* To [🗡️](https://emojipedia.org/dagger/) ***KILL*** [🗡️](https://emojipedia.org/dagger/) your application’s runtime, use ctrl + c

\*Start Hello-React, just creating the application for now\*

**React Application Architecture**

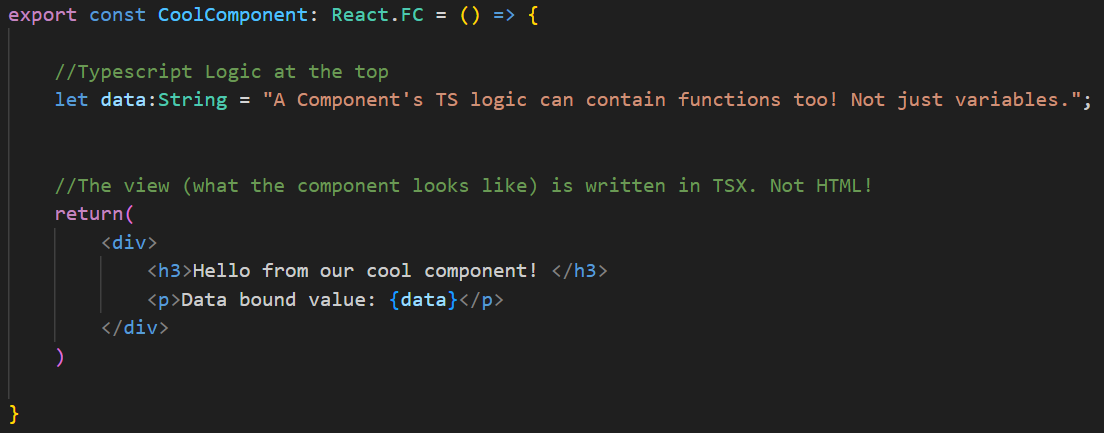
* **Node\_modules:** This is where all the dependencies your application needs reside. It stores the external libraries that are automatically or manually imported.
  + This is the reason we need to run “npm install” when we pull a react app from GitHub. node\_modules doesn’t push with git push, because it’s huge.
* **Public:** This file is a great place to store assets such as images you want in your application, or the logo that’s used on your application’s browser tab (favicon.ico)
* **Src:** Think back to our Java applications - this is where all our SOURCE CODE will go (so we’re gonna be doing most of our coding in here)
  + We’ll make our own folders here, holding components, etc.
  + **App.tsx** - This is where we **render our main application** and input any components we want to display. You can also specify routes to your components, which let us say which URL endpoints they’re accessible from.
  + **App.css** - This lets us apply **global styling** to our application. Good for background images or any styling you want to make consistent across the app.
* **Below the App.tsx is a bunch of metadata files or files that help us structure and configure our application - if we’re lucky, we’ll never have to worry about them.**

**React’s Component Based Architecture**

* React applications are broken down into **components,** which are like the building blocks of our webpage.
  + “Components are essentially **a “section” of a webpage that can be displayed or removed by our React application.”** 
    - Think of all the different boxes that are displayed when you go onto facebook.com. Those are all components. Your newsfeed, ads, other sidebars, etc.
  + Components communicate with each other to create larger applications. These components should follow the **single responsibility principle:**
    - “It is a programming principle that states that every module (in this case components) should have responsibility over a **single part of the application’s functionality.”**
* Components are meant to be reusable pieces of the user interface. Breaking the user interface down into components helps with maintainability and code reusability.
* There are **Two Types of Components in React:**
  + **Class Based Components:** Traditional, the OG component.
  + **Function Based Components**: Newer, we’ll only use these.

# **Functional Component Overview**

* React lets you create components based off functions.
  + These are more simple, flexible, and straightforward to write than Class Components. I exclusively use Function Components.
  + With the inclusion of Hooks (read below), function-based components are as powerful as the bulkier class-based components

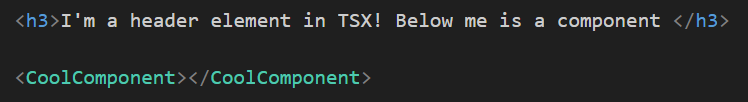
Very typical component structure ^^ It’s the Logic and the View together in one file!

**Rendering React Components**

* To render something means to show/display it. How do we render components in React?
  + Our React components are injected into the root HTML element which is displayed on our page. (This is our **App.tsx**, basically). We never actually leave this one HTML page! **We simply swap components in and out of the user’s view to simulate multiple HTML pages**, which is the point of SPAs in general.
* React uses a **virtual DOM** to map the component’s views to the actual DOM object.
  + This virtual DOM syncs with the actual DOM to improve performance. It’s time consuming and resource intensive to change the actual DOM every time the view needs to change.

**TSX (Typescript XML)**

* TSX is used to **create the template (AKA the view) for React components.** It looks and acts like HTML, but is written in its own syntax. These get injected into the virtual DOM, and then **displayed onto the browser**.
* TSX, unlike HTML, is case sensitive.
  + <lowercase> tags are rendered as HTML elements
  + <Uppercase> tags are rendered as React components
* It’s important to **wrap your entire TSX into a single root tag** because it makes it easier for React to compile your code into the DOM.



**Data Binding**

* In React, we can **inject a variable’s value into a component’s TSX view** with the use of **{curly braces}**. This is called data binding.
  + Through data binding, we can affect what the user sees based on the values of the variables in our TSX file.
* Under the hood: curly braces let us escape from the return() function, and pop back into the {plain ol’ typescript logic} above, which is why we can access variables in this way.

\*made/rendered our first component in HelloReact\*

**Nested Components**

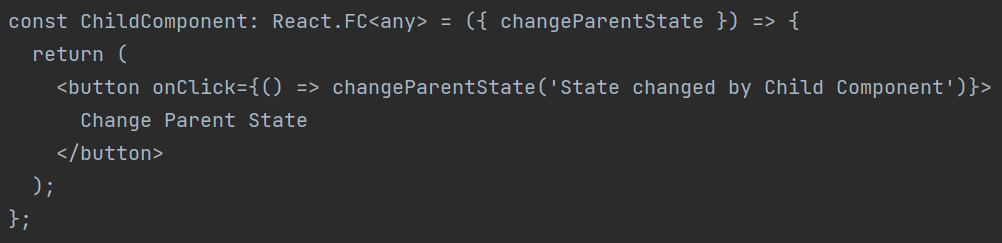
* React allows for components to be **nested inside one another (a component inside a component)**, which can create something like an inheritance hierarchy in Java.
  + AKA a parent child relationship
* Parent components can share data with child components using two objects called **props and state.**
* **React’s data only flows one way,** from the top down (in other words, from parent to child)
  + This is unfortunate, because it means that we can only pass data from parent to child, and the child can’t change the parent’s data unless you do something called “lifting state”. Read below.
* (See ParentComponent/ChildComponent for an example of a nested component)

**Props and State**

* The **State** objects **store information/data within a component.** Like how we have variables in our Java Classes, it’s just stored data we can use in our component.
  + Yes, we can have normal variables, but **a state object is easily passable to child components**, which is why they’re useful.
* To set state in a **function component** you would use the **useState hook** provided by React (more on hooks in function components below).
* **Props (short for properties)** are like arguments/parameters passed into a component. A props object is passed into a child component from a parent component. This is how we can get parent components to **send data** to child components, affecting things like the child component’s view or function behavior.
  + Typically, the props object being sent to the child is the **state** object from the parent.
    - Props are immutable, so a parent component’s state cannot be changed by a child component.
* You can break apart the props object being passed into the component into individual variables using **destructuring**.
  + const {firstName, lastName, city} = person;
  + So now, the values from the prop (person) are now stored in their respective variables - firstName, lastName, city.
* “**Props** is an object with variables sent to a child component from a parent component. This is how we can **pass data from a parent component to a child component**. We include the prop we want to pass down in the **child component’s element tag.**” -slick QC line

**Lifting State**

* As previously mentioned, data can only flow one way in React, and props are read only, so we cannot try to change the state of a parent component.
* However, we can **"Lift State"** by **passing the child component a function as a prop that changes the parent’s state**
  + Lifting state is useful when a common parent/ancestor component is shared by multiple other components that are sharing some value.
  + There are limitations to this. When applications become large and the entire application needs to keep track of state, this approach becomes difficult to manage.
    - However, we will soon learn of technology to manage this problem for us

Here’s an example of lifting state, though we probably won’t use it. Note the function passed in as props  
Also, notice that onClick event handler!

\*parent/child component, and interface in HelloReact\*

**Hooks (QC will ask lots of you about this)**

* Hooks are **functions** in React that allow you to **“hook” into React state/processes** from within **functional components**.
  + There are 4 basic hooks you should know:
* **useState:**
  + Allows you to **store and change state** inside of a function component
    - declare a state variable, and a mutator inside of [square brackets], then useState(value) to set the state to some value
    - use the declared mutator (setXYZ) to change the state
* **useEffect:**
  + Allows you to **perform some logic at specific points** of the applications life
  + It can be used to listen for events or changes in state to perform some logic when that event occurs, such as a button click, or **when a component loads**.
    - Like JS event listeners!! **These are good for event handling**
* **useNavigate:**
  + Allows you to programmatically navigate through your application.
    - It changes your URL endpoint, to hide or render components.
* **useContext: (didn’t get time to cover)**
  + This is used to access values being used within the Context API, which is another way to store state.
    - We'll talk about it more later, but just know that the Context API is a way to store application state in a more flexible way. Think global data sharing, instead of parent to child.

\*Finished hello-react\*

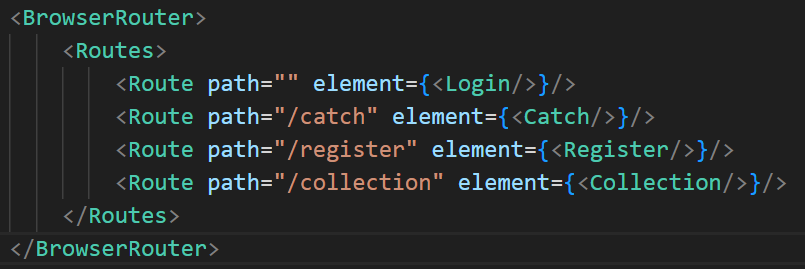
**Routing (QC will most definitely ask about this)**

* In a Single Page Application, **only one page is created,** as we should know by now.
  + However, we can **emulate the experience of browsing multiple pages** to the end user, using **routing.**
  + **“Routing allows for the navigation from one view to another, by changing which components are displayed on the single page.”** -good starting QC line
* Thanks to routing, we can change the user’s view by moving different components in and out. **What components are shown is determined by the URL**
  + , so certain components will be shown when certain URLs are navigated to.
    - /profile might pull up the user’s ProfileComponent
    - /login might pull up the LoginComponent
* We made use of the **react router** to use routing in our applications
* **React Router DOM** is a dependency that provides several solutions for creating a router in your application
  + Install React Router Dom with:
    - npm install react-router-dom
      * Then, you can import the necessary dependencies in your App.tsx

### We need to import **BrowserRouter, (**a component from react-router-dom) in our App.tsx to set up routing. It uses the HTML5 history API to keep the user interface in sync with the URL.

* We start routing within our App.tsx return() method, with the **BrowserRouter** tag.
* The **Routes** tag is used to ensure that only one component gets rendered at a time, exclusively per route.
* Then finally we have **Route** tags, where each Route represents a renderable component
  + **The component to be rendered is nested inside of the Route tag**

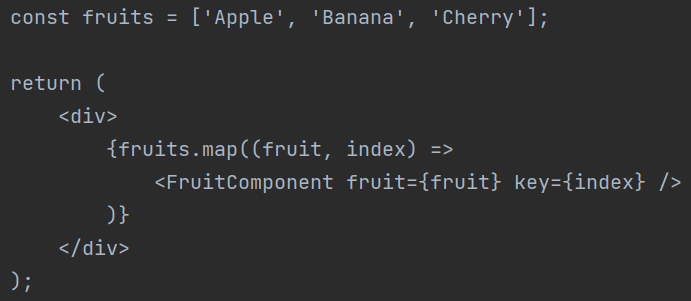
\*It will look a little something like this:



\*started hello-events-routing\*

**Lists and Keys with the .map() function**

* If you want to **render a list of elements** in React, like an Array, you can use the **.map() function**. React recommends that you **provide a unique key for each element** that you are mapping through (but it’s not required).
  + This is useful if you want to do things like render multiple components at once.

  
 Rendering a FruitComponent for every String in the fruits array

* Consider when you search something on amazon (etc) and a number of results pop up. Each of the listed products are styled the same. There is likely a list/key-esque setup that makes one product card render per result returned by your query.
* **BIG PICTURE:** we can use .map to iterate through a list of elements and do something (like render a component) for each of those elements in the list. Kind of like how a for loop works.

\*list/key in the demo\*

**Making HTTP Requests in React**

To create HTTP requests in React, we can use good ol’ fetch, or **Axios** (which is what I do)**.**

# **Axios**

* a promised-based HTTP client for the browser and NodeJS. **This is an easier way than fetch to send HTTP requests**. It is supported by most modern browsers, and comes with the following features:
  + Intercepting and transforming request and response data
  + Automatically parsing JSON response data!!!!
  + Better error handling (a little bit)
  + Built-in protection vs cross-site request forgery
* **Using axios** is straightforward – it uses a verb method, and takes a URL, and an optional body and configuration object
  + Ex: **axios.post(URL, RequestBody, ConfigObject)**
* After the promise object resolves and a response is returned, we can use the same logical methods as with fetch:
  + .then()
  + .catch()
  + .finally()
* We do need to install it to use it!
  + **npm install axios**
* **Axios vs Fetch:**
  + Axios is way less verbose than fetch.
  + Axios automatically translates request/response bodies to and from JSON. This means less code for us to make HTTP requests!
  + Axios can cancel, timeout, and interrupt HTTP requests. Fetch cannot do that as easily.
  + Ultimately it is up to you which to use with React, because they are quite similar in functionality. I just find axios to be way more straightforward.

\*Ben will post pictures of a GET and POST using axios\*

\*Start P1Demo – After seeing Axios, you’ll have 90% of the tools for P1.\*

**\_\_\_The Topics below will be covered later, and won’t be on QC\_\_\_**

**React Application State Management**

* React has an Application State Management problem: **all data in React can only flow one way (unidirectional data flow)**. So sharing state among multiple components can become tricky
  + Remember, state is a built-in React object that lets us store data in a component that we can send to other components, like how we store variables in Java classes that subclasses can access.

I’ll often just store temporary data in some **global data file,** but there are some other possible solutions to this problem:

1. Lifting State: This is usable; however, it becomes too complicated too quickly.
2. Using the Flux Design Pattern through React Redux (modern but complicated)
3. **React Context API**, a built-in centralized data store that comes with React.

**Context API**

* The Context API allows us to **share values between different components without having to explicitly pass a prop** through every level of the tree.
  + This is particularly useful when you have data that needs to be accessed by many components at different nesting levels.
* **Moving to later week, maybe P2 season as this is a big and advanced topic**

**Misc. Topics on the Curriculum------------**

**Higher Order Components**

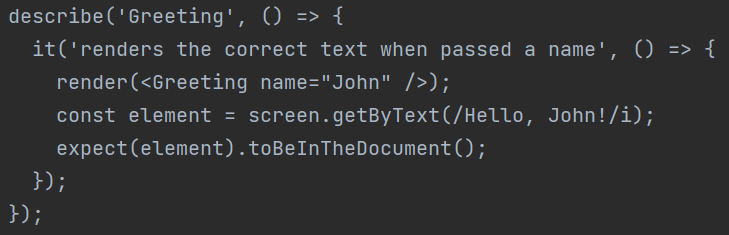
* A Higher-Order Component (HOC) is a design pattern for React components.
* HOCs are **a function that takes a component and returns a new component with additional props** **or behaviors.** 
  + HOCs are used to reuse component logic. They can be used to modify props and component rendering logic.
* **More stuff, or move to later week**

**Refs**

* Dfsdfdsf probably moving to later week

**Jest Overview (moving toward P2 week)**

* Jest is an open-source testing framework designed by Facebook for use with React, but it is also functional with other JS Frameworks
* Jest is popular because:
  + It is lightweight compared to other testing frameworks, yet powerful
  + mocking capabilities (create fake dependent classes to mock API data/user input)
  + capable of testing the DOM easily
* Jest is the default React testing framework, so there's nothing to install
* Uses natural language syntax as much as possible:

  
Testing that the text of the ”Greeting” component is rendered as ”Hello, John!” when passed in the props of name=”John”.

We can use other frameworks to help automate and run the Jest tests for you. Feel free to look into them if front end testing interests you:

* Enzyme
* Selenium
* React Testing Library

**NOT ON QC But Potentially Helpful----------------------------**

**npm run build**

* To build and package your React project you would use npm run build
  + This command will bundle all of your files together, and output a single folder with the final application that you can host statically on an S3 bucket or something. (we’ll talk about S3 in week 7)

Common issues I ran into while learning react:

* React changes so often - to the dismay of programmers worldwide. Make sure when you debug, you’re reading recent material. Things have changed a lot in the last few years.
* “Can’t locate package.json” when running npm start
  + You probably aren’t in your react application when you npm start
* “BREAKING CHANGE: webpack < 5 used to include polyfills for node.js core modules by default. This is no longer the case. Verify if you need this module and configure a polyfill for it.”
  + npm i url --save-dev
* Something about “needs –jsx flag”
  + <https://stackoverflow.com/questions/50432556/cannot-use-jsx-unless-the-jsx-flag-is-provided>

**Common Installs (might be nice to just install these into every new react application)**

* npm install react-router-dom --save
* npm install react-redux
* npm install redux
* npm i --save-dev @types/react-router-dom
* npm i axios
* npm install @reduxjs/toolkit

Revature Serenity Prayer (J. Mockler, 2022)

RevaGods, grant me the serenity to accept the projects I cannot Change. Courage to pass the QCs that I can, and the wisdom to know when it's okay to take a yellow. Living one lecture at a time. Enjoying a new lesson every day. Accepting hardship as a pathway to peace. Taking, as QC does, this weak typing as it is, trusting that I will make all things right if I DO NOT vape on cam.

[11:10 AM] disciple Tyler: proceeds to vape on cam.