**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
* Might need to run: npm install react@18 react-dom@18
* Graphical user interface, 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>

**React Architecture: How to Structure and Organize a React Application**

* <https://www.taniarascia.com/react-architecture-directory-structure/>

**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.

**Destructuring props and state**

* Functional Components
  + We can destructure in two ways, one in the parameter, one in the function body
  + Destructuring in parameter
    - Text

      Description automatically generated
    - Text

      Description automatically generated
  + Destructuring in function body
    - Text

      Description automatically generated
    - Text

      Description automatically generated
* Class component
  + We generally destructure props/state in the render method
  + Text

    Description automatically generated
  + Text

    Description automatically generated

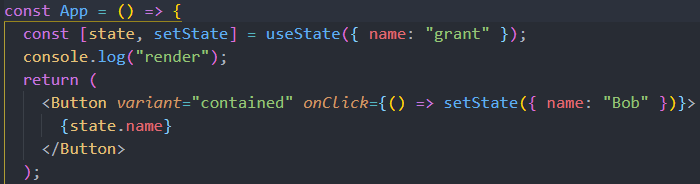
**React DOM/ Virtual DOM/ Re-Rendering**

* 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
  + 1. Update in State
    - Note that a change in state rerenders the component even if the state variable is not displayed.
    - Note that a change in state occurs when prevState === currState evaluates to false. Note the use of triple equals.
      * Ex:
        + Text

          Description automatically generated
        + Console output when the button is clicked many times: 
        + We render once initially.
        + When the button is clicked, we change the state so we rerender.
        + When the button is clicked again, we change the state to obj. But it’s the same obj object we refence everytime we click the button so the value and memory address of the previous state variable and new state variable are the same. Thus, there is no state change, hence no rerender.
      * Ex:
        + Text

          Description automatically generated
        + Console output when the button is clicked many times: Graphical user interface, application

          Description automatically generated
        + We render once initially, creating an obj constant.
        + When the button is clicked, we change the state to that obj constant so we rerender and create a new obj constant with a different memory address (since we created a new instance).
        + Thus, when we click the button, we change the state to that new obj constant which has a different memory address so there is a state change (even though values of the state are the same, the memory addresses are different), causing the component to rerender and to create another new obj constant.
      * Ex:
        + 
        + Console output when the button is clicked many times: Graphical user interface, application

          Description automatically generated
        + We render once initially.
        + When the button is clicked, we change the state to a new obj constant so we rerender
        + When we click the button, we create a new instance of an object. We set the state to take reference this new object instance. Even though it has the same value, the memory address is different since we created a new instance, causing a change in state, causing the component to rerender and to create another new obj instance inside the setState.
  + 2. Update in prop
  + 3. Re-rendering of the parent component
    - A parent component rerendering causes both the parent and its child components to rerender. The parent component is rerendered first, then the child components.
    - Note that a child component rerendering only causes the child component to rerender, and not the parent component to rerender.
* Normal JS variables do not persist through renders when declared inside the component
* Ex: below, the webpage output is always a button with the count state variable’s value and the number 2 below it.
  + Text

    Description automatically generated
* When JS variables are declared outside a component, weird stuff happens. Use useState instead.
* Suppose we want to pass a parent component’s state to a child component. Also suppose we pass a function )that changes the parent component’s state) from the parent to child component. Changing the state of the parent inside the child component via the props will result in the parent component rerendering, which also causes child components to rerender.

**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 called the dependencies array. After any values within the dependencies array (the second parameter) changes, the function (the first parameter) is executed. If the dependencies array includes a state variable that changes, the order of events will be: the state variable will change, followed by a rerender, followed by useEffect function (the first parameter) executing.
* Note: the function is always executed once right after the component is rendered for the first time regardless of the parameters in the array.
* Ex:
  + Text

    Description automatically generated
  + When we render the component for the first time, we get  displayed on the webpage. Right after the component renders, “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 right after the component is rendered for 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.
  + The component renders and displays 4 on the webpage since that is the initial value of the items state variable.
  + After 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. (the following lines are pushed into microtask q)
  + 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. Right after we render for the first time, we run the function inside the useEffect which change the state, causing a second render.
* Ex:
  + Text

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

    Description automatically generated
  + This code generates an infinite loop.
  + We first render the component, its initial state is 0.
  + Right after the component renders, 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.
* Cleaning up side effects
* We can add a return inside the useEffect function. This return itself will also be a function that I will call a cleanup function.
* In general, the cleanup function be called right before the functional parameter to useEffect is executed.
* However, when the component renders for the very first time, the cleanup function is not called.
* Also, the cleanup function will be executed whenever the component gets removed as long as the useEffect function gets called at least once. Since the useEffect function always gets called at least once (right after the component renders for the first time), the cleanup function will always be executed whenever the component gets removed.
* We can put code inside the cleanup function that might cleanup any side effects from the previous execution of the useEffect parameter function, hence the name cleanup.
* Text

  Description automatically generated
* In the above code, after we render the component for the first time, we run the function within the useEffect, but not the cleanup function. Thus, we log “resource change” and then we render the component.
* Note that useEffect changes whenever the component rerenders. In the above code, the component rerenders if the state changes.
* Thus, if we click the users button, we change the value of the resourceType state variable from “posts” to “users”. This change in state causes the component to rerender. Before the component rerenders, useEffect triggers.
* When useEffect is triggered, the return function is executed first so “return from resource change” is logged to the console. Then, the parameter function is executed so “resource change” is logged to the console. Then the component rerenders.
* An example of where this cleanup function might be useful is when we are adding eventListeners.
  + Consider the following code:
  + Text

    Description automatically generated
  + Right after this component renders for the first time, the function within useEffect is executed.
  + The function inside useEffect adds a ‘resize’ eventListener to the window which updates the value of the windowWidth state variable to be whatever the new window inner width is. Not that when the state changes (because the window gets resized), the component is rerendered.
  + However, there is a problem. If we ever remove the App component, there will still be an eventListener for window. This can slow down our application.
  + To resolve this issue, we can add a cleanup function.
  + Text

    Description automatically generated

Component Lifecycle Methods

* When we create components, it goes through many stages of it as shown below.
* Graphical user interface, text, application, email

  Description automatically generated
* React provides us with methods called lifecycle methods that allow to what components should do at different stages of their lifecycle.
* Life cycle methods ONLY exist in the class components, but useEffect Hook can make this kind of method exist in Functional Components. The useEffect Hook sorta acts as componentDidMount, componentDidUpdate, and componentWillUnmount combined.
* Graphical user interface, text, application, email

  Description automatically generated
* We only really need to use: componentDidMount, componentDidUpdate, componentWillUnmount
* Mounting Lifecycle Methods
  + These methods are called when an instance of a component is being created and inserted into the DOM.
  + React has four built-in methods that get called, in the following order, when mounting a component:
  + constructor(props)
    - The constructor(props) method is called before anything else, when the component is initiated, and it is the natural place to set up the initial state , other initial values.
    - We should not do anything that may cause side effects. Ex: do not many any HTTP requests
    - We must call super(props) inside the constructor which will initiate the parent's constructor method and allows the component to inherit methods from its parent (React.Component).
    - Only after we have done super(props) do we have access to this.props.
    - The constructor is the only place we should initialize the state by overwriting this.state fields. In other scenarios, we have to use this.setState.
    - Ex:
      * Text

        Description automatically generated with low confidence
  + static getDerivedStateFromProps(props, state) //RARELY USED
    - This method is called right before rendering the element(s) in the DOM.
    - This method should be used to update the state of a component in response to a change in props.
    - It takes state as an argument, and returns an object that represents the new state of the component.
    - Since this method is static, it does not have access to the ‘this’ keyword. Thus, we cannot call this.setState in this method.
    - We should not cause any side effects in this method such as fetching data.
    - Note we could also return null to make no updates
    - Ex:
      * Graphical user interface, text, application

        Description automatically generated
      * The example below starts with the favorite color being "red", but the getDerivedStateFromProps() method updates the favorite color based on the favcol attribute
  + render()
    - The render() method is required, and is the method that actually outputs the JSX to the DOM.
    - The render method should be a pure method.
    - We should not change the state/interact with the DOM/fetch
    - The lifecycle methods of the children components are executed right after the parent render method.
    - Ex:
      * Graphical user interface

        Description automatically generated with medium confidence
  + componentDidMount()
    - This method is only called once. It is called right after the component and all of its children components have rendered to the DOM.
    - This is where you run statements that requires that the component is already placed in the DOM.
    - This method is where we should cause side effects such as interact with the DOM and fetch data.
    - Ex:
      * Text, letter

        Description automatically generated
  + Consider this following example
    - A picture containing text

      Description automatically generated
    - Text

      Description automatically generated
    - The console will print:
    - Graphical user interface, text, application, email

      Description automatically generated
    - The webpage will be:
    - 
    - Recall the order of mounting lifecycle methods from above.
    - When we create an instance of the App component, the constructor is invoked first. Thus, we set the state to have a field favoriteColor with a value of ‘constructor\_color’. Then we print out ‘Lifecycle A constructor’
    - The getDerivedStateFromProps method is invoked second so we print ‘Lifecycle A getDerivedStateFromProps’. Since we said return {favoriteColor: props.color}, the state now becomes { favoriteColor: "props\_color" }
    - The render method is invoked third so we print ‘Lifecycle A render’. Then we render the JSX onto the webpage and since state is currently { favoriteColor: " props\_color" }, the h1 tag ‘My favorite color is props\_color’ is rendered to the screen.
    - Then the componentDidMount is invoked fourth so we print ‘Lifecycle A componentDidMount’
  + Consider the following example
    - Text

      Description automatically generated
    - Text

      Description automatically generated
    - App and App2 are the same except App2 is a child component of App. As well, App prints ‘Lifecycle A something’ while App2 prints ‘Lifecycle B something’
    - Webpage:
    - Text

      Description automatically generated with medium confidence
    - Color\_b is printed first instead of color\_a because in component A, we rendered rendered App2 first then App’s h1 tag. If we swapped the order, then color\_a would be first. This order has nothing to do with the order of lifecycle methods between parent and class components.
    - Console:
    - Graphical user interface, text, application

      Description automatically generated
    - Recall that componentDidMount() is called right after the component and all of its children components have rendered to the DOM. Thus, before App’s componentDidMount() is called, all of App2 must be rendered to the DOM. That is why ‘Lifecycle A componentDidMount’ appears last in the console.
* Updating Lifecycle Methods
  + These method are called when a component is being rerendered because of changes to either props or state
  + React has five built-in methods that get called, in the following order, when mounting a component:
  + Static getDerivedStateFromProps(props, state)
    - This method is called every time a component is re-rendered.
    - This method should be used when the state depends on the props of the component.
    - We should not cause any side effects
    - This method is invoked in both the mounting and updating phases.
    - Ex:
      * A picture containing text

        Description automatically generated
  + shouldComponentUpdate()//RARELY USED
    - Within this lifecycle method, you can return a boolean  —  true or false — and control whether the component gets rerendered or not respectively (e.g., upon a change in state or props).
    - By default, this function returns true which means all class components will rerender whenever the props/state changes. This method can prevent this default behaviour by returning false.
    - In this method, we can compare the existing state and prop values with the nextProps and nextState values and return true or false to let react know if we should rerender.
    - This method is for performance optimization and should not cause any side effects
    - Ex:
      * A picture containing text

        Description automatically generated
  + Render()
    - If the shouldComponentUpdate method returns true, render causes the component to rerender. If the shouldComponentUpdate method returns false, render is not called.
    - Ex:
      * Graphical user interface, text, application

        Description automatically generated
  + getSnapshotBeforeUpdate(prevProps, prevState) //RARELY USED
    - This method lets you have access to the props and state before the update, meaning that even after the update, you can check what the values were before the update.
    - This method is called right before the changes from the virtual DOM are to be reflected in the DOM.
    - This method returns null or a value. Returned value with be passed as a third parameter to componentDidUpdate() which is the next method we will learn about.
    - If the getSnapshotBeforeUpdate() method is present, you should also include the componentDidUpdate() method, otherwise you will get an error.
    - The value queried from the DOM in getSnapshotBeforeUpdate refers to the value just before the DOM is updated, even though the render method was previously called.
    - Ex:
      * Text

        Description automatically generated
  + componentDidUpdate(prevProps, prevState, snapshot)
    - snap is the value returned from the getSnapshotBeforeUpdate method.
    - This method will be called after the render is finished. Thus we can be sure that the component and all its child components have been properly rendered after the update.
    - This method is executed only once per re-render cycle.
    - We can use side effects such as fetch in this method. It’s good practice to compare the prevProps and prevState values with the currProps and currState values and then decide if we should make the fetch call. Since if we don’t compare, we would be making unwanted requests.
    - Ex:
      * A picture containing text

        Description automatically generated
  + Ex:
    - Text

      Description automatically generated Text

      Description automatically generated
    - When we refresh the page, clear the console, then click the button, we get the following output:
    - Console: A screenshot of a computer

      Description automatically generated with medium confidence
    - The first three methods from A are printed since that is the order of method execution.
    - Recall that whenever a parent component rerenders, all its subsequent child components will re-render, regardless of whether the child components’ props have changed or not.
    - Thus, Lifecycle A prints ‘Lifecycle A render’ and causes App2 to rerender.
    - This causes the next 4 methods from B to print what they need to print.
    - After ‘Lifecycle B render’ is rendered, the changes from the virtual DOM are about to made to the real DOM. This is when the getSnapshotBeforeUpdate is called which is why ‘Lifecycle B getSnapshotBeforeUpdate’ and ‘Lifecycle A getSnapshotBeforeUpdate’ are printed. Why they are in this order, I don’t know. Right after they are printed, the real DOM is changed.
    - Since componentDidUpdate is printed after the entire render process as the real DOM is changed, it makes sense that ‘Lifecycle B componentDidUpdate’ and ‘Lifecycle A componentDidUpdate’ are printed last. Why they are in this order, I don’t know.
  + Ex:
    - Text

      Description automatically generated
    - If we render this app then press the button, the webpage will display:
    - A picture containing text

      Description automatically generated
    - The console will display: 
    - This is because when we click the button, we change the state from { favoriteColor: "constructor\_color\_a" } to { favoriteColor: "new\_color\_a " }. This change in state causes the component to rerender which is why the webpage will display the new color. In the console, we printed out the prevState.favoriteColor in the componentDidUpdate method. The value of the state varaible is { favoriteColor: "new\_color\_a "}, but was previosuly { favoriteColor: "constructor\_color\_a" } which is why constructor\_color\_a was logged to the console.
  + Ex:
    - Text

      Description automatically generated
    - Notice after we click the button, nothing happens since shouldComponentUpdate returns false. Also notice that componentDidUpdate does not print “yes” since it did not update. Note the value of the state variable is still , so there truly was no affect of clicking the button.
* Unmounting Phase method
  + This phase only has one method, componentWillUnmount(), which is invoked when immediately before a component is unmounted and destroyed.
  + In this method, we can perform cleanup tasks such as cancelling network requests, removing event handlers, etc.
  + Ex:
    - Graphical user interface, text, application, email

      Description automatically generated
* Error Handling Phase
  + When there is an error either during rendering, in a life cycle method, or an error is thrown by any child component, these methods are invoked.
  + React has five built-in methods that get called, in the following order, when mounting a component:
  + static getDerivedStateFromError()
    - Whenever an error is thrown in a descendant component, this method is called first, and the error thrown passed as an argument.
    - Whatever value is returned from this method is used to update the state of the component.
    - Graphical user interface, text, application, email

      Description automatically generated
    - Right now, whenever an error is thrown in a descendant component, the error will be logged to the console, console.error(error), and an object is returned from the getDerivedStateFromError method. This will be used to update the state of the ErrorBoundary component i.e with hasError: true.
  + componentDidCatch()
    - The componentDidCatch method is also called after an error in a descendant component is thrown. Apart from the error thrown, it is passed one more argument which represents more information about the error:
    - Graphical user interface, text

      Description automatically generated with medium confidence
    - In this method, you can send the error or info received to an external logging service. Unlike getDerivedStateFromError, the componentDidCatchallows for side-effects.
    - Also, since the ErrorBoundary can only catch errors from descendant components, we’ll have the component render whatever is passed as Children or render a default error UI if something went wrong:
    - Graphical user interface, text, application, email

      Description automatically generated

Error Boundaries

* Error boundaries are React components that catch JS error in their child component tree, log those errors, and display a fall-back UI. Error boundaries are a way to gracefully handle error in application code.
* A class component becomes an Error Boundary by defining either or both of getDerivedStateFromError and componentDidCatch lifecycle methods.
* Ex:
* Text

  Description automatically generated Text

  Description automatically generated
* Because we passed Joker as a heroName, an Error is thrown and our entire application stops. However, we probably don’t want to stop the entire application and instead, just stop the components that aren’t working. Thus, we will create an ErrorBoundary component.
* Text

  Description automatically generated
* We will place this ErroryBoundary component around the Hero components since the Hero components might result in an error.
* Text

  Description automatically generated
* Webpage Output: A picture containing text, clipart

  Description automatically generated
* The placement of the Error Boundary also matters as it controls if the entire app should have the fall-back UI or just the component causing the problem. For example, the Hero component with ‘name’ prop of ‘batman’ should be rendered, but the ErrorBoundary should kick in for the Hero component with ‘name’ prop of ‘Joker’.
* Text

  Description automatically generated Webpage Output: A picture containing logo

  Description automatically generated
* Notice that batman is being rendered and the Error Boundary is working. Ideally, we would have another component that that renders an ErrorBoundary component with a Hero component inside it to make our code more reusable.
* We can also use componentDidCatch to define a Error Boundary.
* Text

  Description automatically generated
* In the above example, we log the error and info to the console, but that is redundant since React already renders them by default.
* Error Boundaries catch errors during rendering, in lifecycle methods, and in constructors of the whole tree below the components. However, they do not catch errors inside event handlers so we would need to use try catch for that.

**React Router**

* Might need to run: npm i react-router-dom
* React Router DOM enables the navigation among views of various components in a React Application, allows changing the browser URL, and keeps the UI in sync with the URL.
* Since we never fetch a new html page, we give the illusion of going to new pages by rendering certain components.
* BrowserRouter, Routes, Route
  + BrowserRouter (often imported a Router): BrowserRouter is a router implementation that uses the HTML5 history API(pushState, replaceState and the popstate event) to keep your UI in sync with the URL. It is the parent component that is used to store all of the other components that want to be included in the routing system.
  + Routes: this is a component that determines where in our router system we want to have routes. It’s a new component introduced in the v6 and a upgrade of the Switch component. The main advantages of Routes over Switch is that Routes are chosen based on the best match instead of being traversed in order.
  + Route: Route is the conditionally shown component that renders some UI when its path matches the current URL. Inside the Route component, we have a path and an element field. The path on the website matches the path in the Route, the component inside the element field will be rendered.
  + Text

    Description automatically generated
  + The Home, About, Pages, and Error components just return a div that has text Home, About, Pages, or Error respectively.
  + Now, when we load our react app which should have a default path of ‘/’, we render the Home component. If we type “/about” in the url as the path, we render the About component. If we type “/page” in the url as the path, we render the Pages component. If we type anything else in the url as the path, we render the Error component.
  + Note that if we didn’t have <Route path="\*" element={<Error />} /> and we tried typing in a random path, we would get an blank white page.
  + Note that the header and footer are always rendered no matter what Route is rendered since the header and footer do not depend any path in order to be rendered.
* Link
  + The user shouldn’t navigate from one page to another by typing in paths in the url.
  + To solve this problem, we can import the Link component from react-router-dom which provides links that can be clicked to navigate to different routes.
  + This is similar to the <a> tag that exists in normal HTML.
  + Inside the Link component, we can say to = “/somepath”. The path we pass in is the path that we want to be directed to when we click the link.
  + Text

    Description automatically generated
  + Notice when we click the different links, the paths in the url changes and the component rendered also changes. Also note that similar to header and footer, the Link components are always rendered since they don’t depend on any path in order to be rendered.
* useNavigate
  + While we can navigate to different routes by changing the url or clicking Links, we might also want to go to a route when a button/div/some other HTML element is clicked.
  + To solve this problem, we can import the useNavigate Hook from react-router-dom.
  + To use the useNavigate Hook, we call useNavigate() which is a function that doesn’t take any parameters and returns a function. This returned function takes in a path as the parameter and navigates towards the certain path when it is executed.
  + Text

    Description automatically generated
  + Notice that we define the function const navigate = useNavigate();
  + Then we pass in a function to the button which returns navigate(“/about”) when it is clicked. Executing navigate(“/about”) will take us to the about page.
* useParams
  + Suppose we want to go to an about page for a specific user. So if we go to “/about/user1”, we would want to get page that corresponds to user1.
  + To do so, we can import the useParams Hook from react-router-dom.
  + useParams allows us to add parameters to certain paths and access those parameters.
  + To add a parameter to a certain path, we add “/:my\_parameter\_name” to the end of the path. 
  + In the example above, the parameter\_name is “my\_id”.
  + Text

    Description automatically generated
  + To access the parameter, we go to the component that is rendered at that path and call the useParmas function. This function returns an object with one property. The object is { my\_parameter\_name : path\_parameter\_that\_user\_passed\_in}. We can destructure this object to get the value associated with the my\_parameter\_name property.
  + Ex:
    - 
    - Text

      Description automatically generated
    - Suppose we go to the path ‘/about/user3’
    - Since that path matches the Route with path = “/about/:my\_id”, we render the component at that Route which is the About component.
    - There, we use the useParmas() function to return an object that contains the value of ‘user3’ at the property with name my\_id.
    - This object can be logged to get: A picture containing diagram

      Description automatically generated
    - We can destructure the object so we now have a constant named my\_id that has a value of ‘user3’. Note that ‘user3’ is a string
    - Now, we can use my\_id in our application such as in the div.
    - Graphical user interface, text, application, chat or text message

      Description automatically generated
  + useParams can be useful to get the id of a certain user, allowing us to make an api request to get information about that certain user.

**Conditional Rendering**

* Inside react, we can’t use if statements.
* Instead, we use conditional rendering when we want to render a component/element based on a condition.
* We will have to use the useState hook to keep track of the condition.
* There are two ways to conditionally render, one with the && logical operator, and one with the ternary operator.
* && logical operator
  + Since the && short circuits, we can put the condition as the first operand and the component as the second operand. If the condition is true, the second operand will be evaluated, and the component will render. If the condition is false, the && operator short circuits so the second operand will never be evaluated, so the component is not rendered.
  + Ex:
  + Text

    Description automatically generated
  + Graphical user interface

    Description automatically generated
  + We have a Toggle component that we want to conditionally render.
  + We use the useState hook to keep track of whether we should render the Toggle component or not.
  + Initially, isToggled is false. Thus, when we render the component, we have to consider the expression {isToggled && <Toggle />}. Since isToggled is false, the && operator short circuits and the Toggle component is never rendered.
  + When we click the button, we change the isToggled state variable from false to true. This change in state causes the App component to rerender, Thus, we have to reconsider the expression {isToggled && <Toggle />}. Since isToggled is true, the && operator continues to evaluate the following operand, evaluating the Toggle component renders it.
* Ternary Operator:
  + The ternary operator behaves much like an if-statement.
  + Text

    Description automatically generated
  + In the example above, we check if isToggled (state variable) is true. If it is, we render the Toggle component. If not, we render the paragraph element.

**List Rendering and Keys**

* When we want to render arrays(lists) in react, we want to repeat some HTML for each item in the array.
* To do this, we can use the map method to add HTML to each item.
* Text

  Description automatically generated
* Webpage Output: Text

  Description automatically generated
* While this code works, it is better practice to refactor the JSX used in the map function into a separate component.
* Text

  Description automatically generated Text

  Description automatically generated
* This is better code since the person component is only responsible for rendering a person and and the App component is only responsible fro rendering the list of persons.
* However, there is actually a problem if we open the console.
* Text

  Description automatically generated
* This error is telling us that each item that was rendered using the map operator should have a prop called ‘key’. Moreover, each item should have a unique value for the ‘key’ prop. The value of this ‘key’ prop does not need to be an id, or a number, it can be whatever we want, as long as each item has a unique value for the ‘key’ prop.
* All we have to change is in the app component to have the key prop as shown below.
* 
* Notice that person.id is unique since each person in our example has a different id. The first person has id 1, the second has id 2, and the third has id 3.
* Note that instead of making the value of the key prop to be person.id, we could make it person.name since each person in our example has a different name.
* Note that the key prop is not accessible in the child component. Thus, the Person component does not have access to the key prop.
* Text

  Description automatically generated
* Webpage output: Text

  Description automatically generated
* Console: Text

  Description automatically generated
* Notice that we destructure the key prop in the Person component. However, this results in nothing being outputted on the webpage and an error in the console since the Peron component (the child component that the person info was passed to) does not have access to the key prop.
* To resolve this issue, we should pass the ‘key’ prob value as another prop.
* Text

  Description automatically generated
* Notice above how we passed a new prop called ‘my\_key’ which also has a value of person.id
* Text

  Description automatically generated
* Notice since we passed in the my\_key prop, we are able to destructure it and use it.
* Why are keys needed?
* Keys allow react to render faster since we can easily identify which items in a list need to be rerendered instead of possibly rerendering the entire list.
* Consider the following example where the list items don’t have keys
  + Graphical user interface, timeline

    Description automatically generated with medium confidence
  + The left is what was rendered before and the right is what was rendered after.
  + React will iterate over both lists at the same time and generates a mutation whenever there is a difference.
  + First, react will compare the first items in each list and see that there is no difference.
  + Then, react will compare the second items in each list and see that there is no difference.
  + Then, react will compare the third items in each list and see that the there is a difference and will insert the third item into the DOM tree.
  + In this case, react does not need to tear down the entire tree and then rebuild it from scratch. It just needs to insert the new item at the end.
* Consider the following example where the list items don’t have keys
  + A screenshot of a computer

    Description automatically generated with medium confidence
  + When comparing these two lists, react will realize the first items are different, the second items are different, and the third items are different.
  + React will mutate every child instead of realizing that it just needs to insert the new Diana list item at the front.
  + This inefficiency can be a problem which can be resolved by keys.
* Consider the following example where the list items have keys
  + Graphical user interface

    Description automatically generated
  + When comparing these two lists, react will realize that the list item with key = ‘3’ is a new item that needs to be inserted at the top. React will also realize the list items with key = ‘ 1’ and key = ‘2’ don’t need to be mutated so the rest of the tree is preserved.
* Generating Keys
  + We could use the item index in the array as its key, but that should only be done if the item in your list do not have a unique id, the list is static and will not change, and the list will never be reordered or filtered. In fact, react uses the index as the key if you don’t specify the key prop.
  + To generate keys, we can use the install an npm package called uuid that will allow us to generate unique id’s.
  + In the cmd, run: npm install uuid
  + Now, in the component we want to use the unique id’s in, we import uuid
  + 
  + To generate a random id, execute uuidv4() which returns a random id.
  + A screenshot of a computer

    Description automatically generated with medium confidence
  + Notice the use of uuidv4() in line 6.
  + Note that each time we execute uuidv4(), we get a different id. If we have to use an id many times, storing that value in a constant seems like the intuitive solution, but that doesn’t work.
  + Ex:
  + Text

    Description automatically generated
  + Webpage output: Text

    Description automatically generated with medium confidence
  + Console output: Text

    Description automatically generated
  + Note that the randomId’s don’t match.

**Event Listeners**

* Event listerens in react is basically the same as in JS. Make sure to use camelCase for events listeners such as ‘onClick’ instead of ‘onclick’. Also make sure to pass in a function such as clickHandler and not a function execution such as clickHandler().
* Functional components:
* Text

  Description automatically generated
* Class components:
* Graphical user interface, text, application

  Description automatically generated
* Note that it is convention for the function (this function refers to the function that will be executed when an event occurs) to include the word ‘handle’. Ex: onClickHandler, clickHandler, handleClick are good names for functions that will handle a button click. renderListHandler is a good name for a function that will render a list upon an event.

**useRef**

* Suppose we tried to count how many times our application renders using the useState Hook, we would be caught in an infinite loop since this Hook itself causes a re-render. The code for this is shown below.
* Text

  Description automatically generated
* The App component renders for the first time with the count state variable initially having a value of 0. After the first render, we then run the useEffect which changes this state to 1. Changing the state causes the component to rerender so the component rerenders. After the component rerenders, the function within useEffect is called once again so we change the state once again, causing the component to rerender.
* To avoid this, we can use the useRef Hook. useRef is a hook that we can import that allows us to store a value (much like useState) and access a DOM element directly.
* useRef(initialValue) is a built-in React hook that accepts one argument as the initial value and returns a reference (aka ref). A reference is an object with only one property called current. The value associated with the property current is called the reference value. The reference value will initially be initalValue which is the value that we passed into the useRef function as a parameter. If no argument is passed to useRef so we see useRef(), the default parameter is null.
* reference.current accesses the reference value, and reference.current = newValue updates the reference value to be newValue.
* Graphical user interface, text, application

  Description automatically generated
* The reference value is persisted (stays the same) between component re-renderings (like useState)
* Updating a reference doesn't trigger a component re-rendering (unlike useState)

Using useRef to store a value

* Ex:Now, we can count the number of times a component is rerendered with the code below.
  + Text

    Description automatically generated
  + The component renders for the first time to display: . This is because the reference value (renderCount.current) was initially set to 1 which was the initial value we passed to the useRef function.
  + After the component renders, the function inside useEffect is executed. The renderCount reference value of 1 is logged to the console. Then we change the reference value by incrementing it by 1. Then we log the renderCount reference value of 2 to the console. Note that useRef is unlike state since the state variable’s value is when it was last rendered. The refence value is not the value when the component was last rendered, but rather what the value currently is. When we finished executing this function within useEffect, the component does not render since no state was changed. Recall that unlike state, a change in the reference value does not trigger a rerender. Note that the reference value as of now is 2 but the refence value as of the last render is 1.
  + Output as of now: Graphical user interface, text, application

    Description automatically generated
  + Now let’s enter one character in the text field, lets say its ‘a’. This event is captured by the onChange event listener which executes the handleOnChange function.
  + The handleOnChange function changes the value state variable to take on the value that was in the text field. In this case, the value state variable now has a value of ‘a’. Since this function changed the state, when the function finishes executing, the component rerenders which causes the following output:
  + Graphical user interface, application

    Description automatically generated
  + This is because when the component rerenders, the current refence value is 2 so we display 2 on the webpage. Since we also changed the value state variable to be ‘a’, ‘a’ appears in the text field.
  + After the component rerenders, the function within useEffect is called again. It logs out the current refence value which is 2. Then it increments the refence value by 1 so it now has a value of 3. The function then logs out the current refence value of 3.
  + Console: 
* Using useRef to reference a DOM element
  + The most common use case for refs in React is to reference a DOM element.
  + Because of how common this use case is every DOM element has a ref property you can use for setting a ref to that element.
  + Text

    Description automatically generated
  + In the example above, we say ref = {inputRef}. In doing so, we set the reference value of inputRef to be the input element.
  + This component has an input field, a div that displays the value of the name state variable, and a button.
  + The user can input text into the input field, which is captured by the onChange event listener which executes the handleOnChange function, which updates the name state variable to be the value of whatever is in the input field. This change in state causes the component to rerender, which changes what is displayed in the div that says ‘My name is {name}’.
  + When the user clicks the button, the click is captured by the onClick event listener which executes the turnGreen function.
  + The turnGreen function uses the value of inputRef.current and recall that we said <input ref = {inputRef}>. Thus, the value of inputRef.current is the input Element. This is the same return type as if we were to do document.getElementById(“hey”). In fact, not only are they the same return type, the Element object returned by inputRef.current is exactly the same as the Element object returned by document.getElementById(“hey”) which is proven by the console logging true to the === statement. Thus, we can change properties for his Element object such as changing its font color to green. Note that this change in the Element does not cause the component to rerender but will cause the text inside the input field to change to green(the user can actually see that the text changed to green).
* Using useRef to store the previous value of state
  + Text

    Description automatically generated
  + In the example above, the value of the name state variable is initialized to be “” and the prevName refence value is initialized to be null.
  + Webpage output: 
  + Right after the component rerenders, the function inside useEffect is executed so the prevName refence value is now the value of the name state variable which is “”. Thus, the prevName refence value changed from null to now be equal to “”.
  + For the following: recall a state variable change will be followed be a rerender which will be followed by the function within useEffect being executed
  + When the user enters a character, say ‘a’, into the input field, the handOnChange function is executed. This causes the name state variable to now have a value of whatever is inside the input field which is ‘a’. Thus, the name state variable now has a value of ‘a’.
  + This change in state causes a rerender so the output is now: This is because the name == ‘a’ so ‘My name is a’ is displayed on the webpage. As well, since the value of the input field is name, the text inside the input field is ‘a’. Recall that the prevName reference value is current “” which is why “used to be” is displayed on the webpage.
  + After the component rerenders, the function within the useEffect is executed. Now, the prevName refence value is the value of the name state variable which is “a”. Thus, the prevName refence value changed from “” to now be equal to “a”.

Things to avoid with useRef

* + While useRef can store a value much like useState. if we can use useState, we should use useState instead. To change values should be done with state, but reading values can be done with useRef if needed.
  + Do not add/remove DOM elements in a component. This can lead to inconsistencies between the actual DOM and the React virtual DOM which is very bad.

**Building Simple Forms**

* Check the code I wrote in the folder called form1. Note that the code is not very good since I tried passing state, variables, and functions from component to component without understanding what really happens.
* Ideas:
  + Use useState to store the text value inside the input fields.
  + When users enter text, update state by using event.target.value to get value inside the text field
  + event.preventDefault allows us to prevent the webpage from refreshing upon form submission so we can do something with the state variables
* Resources:
  + <https://www.youtube.com/watch?v=8hU0I8rY4u4>
  + <https://www.youtube.com/watch?v=7Vo_VCcWupQ>

**Styling Components (CSS)**

* It’s best to use CSS Modules
* CSS Stylesheets
  + Ex:
  + Text

    Description automatically generated
  + Text

    Description automatically generated
  + A picture containing text

    Description automatically generated
  + Th myStyles.css file makes elements with class = ‘primary’ to have an orange font color and elements with class = ‘font-xl’ to have a large font size.
  + In the Stylesheet component, we import this myStyle.css file. We say let className = props.primary ? “primary” : “”. This line gets the value of props.primary (the value passed by the App component). If props.primary is true, the value of className is “primary”. Otherwise, the value of className is “”.
  + We let the h1 element have a class name of className and font-xl.
  + The App component renders the 2 Stylesheet component and sets the ‘primary’ prop to have a value of true for one, and false for the other. Both h1 elements in the Stylesheet component will have a class name of ‘font-xl’ so both render text in large font. Since only the first Stylesheet component has a class name of “primary” (since the value passed to the ‘primary’ props was true), only the first Stylesheet component’s h1 font color is orange, while the other is default black.
  + Note the spelling and capitalization of the className keyword
  + Conflicts can appear with CSS stylesheets
  + CSS class names and animation names are scoped globally by default.
  + Ex:
    - Text

      Description automatically generated
    - Graphical user interface, text, application

      Description automatically generated
    - Text

      Description automatically generated
    - A screenshot of a computer screen

      Description automatically generated with medium confidence
    - Output: Text

      Description automatically generated
    - In this example, myStyle.css makes the font color red for all elements with a class name of red.
    - We have a App component that has renders Component1, Component2, and a div with class name of ‘red’.
    - Component1 renders a div with a classname of ‘red’.
    - Component2 renders a div without any class names.
    - When using stylesheets, CSS class names and animation names are scoped globally. Thus, even though we imported the myStyle.css file in the App component, the styling is not just applied to the div in the App component, but also the div in Component1. This is why the font color of the text ‘Component1’ is also in red.
    - Note that the same thing would happen if we imported the myStyle.css file in Component1. This is because CSS class names and animation names are scoped globally so all other elements with class name of ‘red’ are affected, even if they are in different components.
* Inline styling
  + Text

    Description automatically generated
  + We create an object whose fields are camelCased css stylings and values are strings.
  + We set the h1’s style attribute to have a value of heading which is the object we created above. Now, the h1 element will have the styling.
* CSS Modules
  + A CSS Module is a CSS file in which all class names and animation names are scoped locally by default. Take note of the words scoped locally.
  + Css modules file end in .module.css
  + When we import, we say: import importName from ‘.path/moduleName.module.css’
  + 
  + Unlike CSS stylesheets, CSS module classes and animations are only available within the component where they are used. This resolves the conflict issue that CSS stylesheets have. While CSS module classes and animations are locally scoped, element tags such as div, h1 still have global scope.
  + Ex:
    - Text

      Description automatically generated Graphical user interface, application

      Description automatically generated
    - A screenshot of a computer

      Description automatically generated with medium confidence
    - Text

      Description automatically generated
    - Output: Text

      Description automatically generated
    - In the above example, notice that component1 has a div with a className = ‘red’. We include compoennt1 in the app component which also has a div element with a className = {styles.red}. Since the styling from class names from CSS modules are only applied to the component that it is imported in, only the div inside the App component has red text.
  + Ex:
    - Suppose we continue with the above example but change the CSS module to be the following:
    - Graphical user interface, application

      Description automatically generated
    - Output: Diagram

      Description automatically generated with medium confidence
    - The font color within every div in any component is red. This is because while class names and animation names are scoped locally for CSS modules, elements are still globally scoped. Thus, styling an element such as div {color:red} is still applied to every div element in every component and not just the App component.

**Sass**

* Everything will be the same as normal CSS except the following:
* npm install -s node-sass
* rename .css to .scss (this applies for css modules too so they become modeName.module.scss)
* make sure to change the extension of the scss file when we import it
* Ex:
  + Text

    Description automatically generatedA picture containing text

    Description automatically generated
* Note that the scoping of classnames remains unchanged (it is the same as css)

**Center a div both horizontally and vertically**

* Text

  Description automatically generatedText

  Description automatically generated
* Output: Graphical user interface

  Description automatically generated

**Deployment**

* Heroku
  + Create a new repo on github
  + A screenshot of a computer

    Description automatically generated with medium confidence
  + Add files to repo: public, src, .gitignore, readme.md, package-lock.json, package.json
  + Graphical user interface, application

    Description automatically generated
  + Create a new app on Heroku
  + Graphical user interface, text, application, email

    Description automatically generated
  + Click on the app and go to settings, go to buildpack and paste: <https://github.com/mars/create-react-app-buildpack>
  + Go to deploy, click github icon, enter repo name, and click connect
  + Graphical user interface, text, application, email

    Description automatically generated
  + Click enable automatic deploys
  + Graphical user interface, text, application, email

    Description automatically generated
  + Click deploy branch
* Netlify
  + In cmd: npm run build
  + Drag the build folder into netlify
  + Graphical user interface, application, Word

    Description automatically generated

**Portals**

* React portals provide a way to render components into any DOM node in index.html.
* As of now, all of our components are rendered inside the div with id = ‘root’ in the index.html file. Portal allows us to break out of the DOM tree so we can render a component that is not under the div with id = ‘root’.
* To create a portal, we first need to add a new DOM node which is where we want the component to be rendered to. Generally, we create this new DOM node right underneath the root div in index.html.
* Text

  Description automatically generated
* To create the component that will be rendered in the div with id = ‘portal-root’ (note that we can set the id to be anything), we do the following:
* We need to import: 
* In the render method, instead of returning JSX, we are going to return ReactDOM.createPortal(). ReactDOM.createPortal takes two parameters, the first being the JSX we want the component to render, the second being the DOM node we want to render to.
* Text

  Description automatically generated
* We can add the PortalDemo component to the App component.
* Text

  Description automatically generated
* When we run the server and inspect the webpage, we see the following: Text

  Description automatically generated. Notice that the PortalDemo component is rendered in the div with id = ‘portal-root’ and not in the div with id = ‘root’.
* Applications of Portals
  + Portals are useful when we have a child component that acts as a modal/pop-up. You probably want to review z-index in css for this.
  + Ex: consider the following code that does not use portals
    - Text

      Description automatically generated
    - Text

      Description automatically generated
    - In the above example, in our App component, we render a div that contains a button that toggles the render of the PortalDemo component. Notice that this div has a z-index of 1. Note that the PortalDemo component has a z-index of 1000. Inside the App component, we also render a second div that has a z-index of 2. Even though the z-index of the PortalDemo is the highest, it second div with z-index = 2 is rendered on top of the PortalDemo component as shown below (notice the bright red).
    - Text

      Description automatically generated
  + Ex: consider the following example which fixes the above problem by using portals.
    - The code will be the exact same in the example above except that we add the following DOM node:  to index.html.
    - Then, instead of having the PortalDemo component return just JSX, it returns ReactDOM.creatPortal().
    - Text

      Description automatically generated
    - Now, notice that the output is as follows: Text

      Description automatically generated

**Higher Order Components (HOC)**

* Suppose we want to create a button that displays the number of times it was clicked. The code would look like the following:
* Text

  Description automatically generatedText

  Description automatically generated
* Now suppose we want to add a button that displays the number of times the button was hovered over. The code would look like the following and we would add the HoverCounter component to the App component:
* Text

  Description automatically generated
* Notice how a lot of the code was copy-pasted which is bad design with lots of duplication and no reusability of functionality.
* Having the parent component manage the state and passing down this state and its handler as props to the children might seem like the intuitive approach.
* Diagram, timeline

  Description automatically generated
* However, this solution becomes ineffective when there many be many intermediate components as shown below.
* Diagram, timeline

  Description automatically generated
* Thus, there is a need to share common functionality between components with repeating code.
* To do so, we can use Higher Order Components (HOCs) which is a pattern for reusing component logic.
* We can create a pure function takes a component as an argument and returns a new component. The new component is the HOC. The syntax could look like the following:
* 
* 
* A HOC can return a class/functional component and the HOC itself is a function.
* To create a HOC that adds state to a component, we can create the following component:
* Text

  Description automatically generated
* We created a new function called EnhancedComponent which takes in a component called OriginalComponent as its first parameter and a second parameter called name.
* Inside EnhancedComponent we create a new functional component called NewComponent. This NewComponent has a count state variable, a function incrementCount that updates the count state variable, and returns OriginalComponent but with extra props added to it. Notice that we added the props: count, incrementCount, and name. This means the OriginalComponent has access to these props. Notice name = {name} so the name that was passed as the second parameter to the EnhancedComponent function becomes the value of the name prop.
* We then return NewComponent.
* Notice that this function EnhancedComponent consumes a component called OriginalComponent and a parameter name and returns a functional component NewComponent.
* Now, we can apply this HOC on other components such as the ClickCounter and HoverCounter components.
* Text

  Description automatically generated Text

  Description automatically generated
* Notice how instead of exporting ClickCounter/HoverCounter, we are exporting EnhancedComponent(ClickCounter, “by ClickCounter”)/ EnhancedComponent(HoverCounter, “by HoverCounter”). This means the EnhancedComponent() function is being applied to ClickCounter and HoverCounter to return a HOC. Also “byClickCounter” and “by HoverCounter” are the values of the name parameter that is being passed to the EnhancedComponent function. Note that ClickCounter/HoverCounter receive separate state.
* Notice that ClickCounter/HoverCounter now have new props (count, incrementCount, name) because we applied the function EnhancedComponent on ClickCounter/HoverCounter. We can access those props as we normally would with props.count, props.name, props.incrementCount.
* Now, when we see the following on the webpage: with their functionality.
* Note that we could also implement the above function by returning a class component as shown below:
* Text

  Description automatically generated
* Naming Conventions
  + The name of the function (by function I mean the pure function takes a component as a parameter and returns a new HOC component) describes the functionality that will be added to the parameter component. This name should be in camelCase.
  + The function’s file name should be the same as the function name and should also be in camelCase.
  + The component that is taken in as a parameter to the function is usually called WrappedComponent.
  + The new component that the function returns should be the same as the function name but in PascalCase.
  + Text

    Description automatically generated
* Passing Props to HOCs
* Ex:
  + Suppose we have the following code:
  + Text

    Description automatically generated
  + Text

    Description automatically generated
  + Text

    Description automatically generated
  + Webpage output: 
  + Notice we passed the prop color = ‘green’ to the ClickCounter Component in App.js but it is not showing up. This is because the WithCounter component we return in the withCounter function does not take use any of the props. To do fix this, we need to add {…props} to the return of WithCounter. Thus, change WithCounter from the above to the below to print:
  + Text

    Description automatically generated 
* Consider the following code:
  + Text

    Description automatically generated
  + Text

    Description automatically generated
  + Text

    Description automatically generated
  + Webpage output: 
  + This code is the same as above after we added props to the return of the WithCounter component.
  + Notice that we passed in a prop named ‘name’ in App.js and the withCounter function also takes in ‘name’ as a parameter. The reason the name being rendered is ClickCounter is because that was the prop added last. If we changed the above to the following, the output would be:
  + Text

    Description automatically generated 

**Render Props**

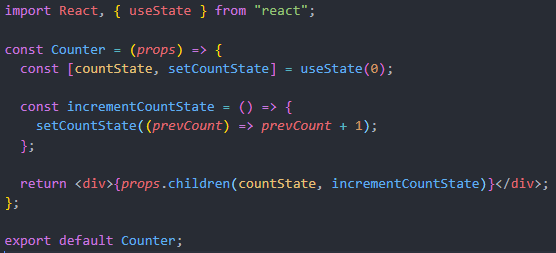
* While HOC is one way to share common functionality between components with repeating code, render props also provides a solution.
* Render prop refers to a technique for sharing code between React components using a prop whose value is a function. We can use a prop whose value is a function to control what is being rendered.
* Text

  Description automatically generated
* Text

  Description automatically generated
* Text

  Description automatically generated
* Text

  Description automatically generated
* In App.js we render a Counter component and pass in the function { (count, incrementCount) => (<ClickCounterTwo count={count} incrementCount={incrementCount} />)} as the value to the render prop.
* In the Counter component, we create a ‘countState’ state variable and a ‘incrementCountState’ function that allows us to change increment the value of the ‘countState’ state variable by 1. This component returns a div with {props.render(countState, incrementCountState)} inside it. Notice props.render(countState, incrementCountState) is a function execution with (). Thus we are executing the function (count, incrementCount) => (<ClickCounterTwo count={count} incrementCount={incrementCount} />) with the ‘countState’ state variable and ‘incrementCountStaet’ state function (that we created in the Counter component) being passed as parameters. This means inside the div we are returning <ClickCounterTwo count={countState} incrementCount={incrementCountState} /> where the value of the ‘count’ prop is the ‘counState’ state variable and the value of the ‘incrementCount’ prop ‘incrementCountState’ state function.
* In ClickCounterTwo, we render a button that is labelled {props.count} with an onclick function of incrementCount. Thus, the value labelled on the button is the ‘countState’ state variable since that was the value passed to the ‘count’ prop. The function that handles the onClick is the ‘incrementCountState’ state function since that was the value passed to the ‘incrementCount’ prop.
* A similar process happens for HoverCounterTwo.
* Note: there will not be a state conflict between ClickCounterTwo and HoverCounterTwo.
* Note: the prop does not need to be named ‘render’, you can call it whatever.
* Note: Instead of passing a function as the value to a ‘render’ prop, we can use pass that function in between parent component tags as a children props.
* Text

  Description automatically generated
* 
* In the App component, notice we are passing a function in between the Counter component tags.
* Inside the Counter component, we are saying props.children(countState, incrementCountState) instead of props.render(countState, incrementCountState). This is the same since props.children is the function {(count, incrementCount) => (<ClickCounterTwo count={count} incrementCount={incrementCount} />)}. Before, we passed in the value {(count, incrementCount) => (<ClickCounterTwo count={count} incrementCount={incrementCount} />)} to the render prop.

**PropTypes**

* PropTypes is a tool in React that lets you validate the structure and type of all your props and if there are any issues they will show up as errors in the console of your browser dev tools.
* In order to start using PropTypes in your project you need to install the PropTypes library. 
* It doesn't matter if you are working with a class or function component, setting up PropTypes works exactly the same. You just need to define a property on the class/function called propTypes. This property will be an object that outlines all the props for the component.
* Make sure to import: import PropTypes from 'prop-types'
* Text

  Description automatically generated Text

  Description automatically generated
* Checking Basic Types
  + The most basic way you can check a prop's type is by checking to see if it is one of the primitive types in JavaScript, such as a boolean, string, object, etc. In order to do this we would use the name of the prop as the key in the propTypes object and the value for that key will be the specific PropType we are checking.
  + Text

    Description automatically generated
  + In the above code we are saying that we expect the name prop of the Component to be of the type string. We also expect the age prop to be a number, the address prop to be an object, and the friends prop to be an array.
  + This is the easiest way to setup PropTypes and is great for catching minor errors, such as, accidentally passing a string instead of a number.
  + 
  + If we create a component with the above code we will get a warning in the console. “Warning: Failed prop type: Invalid prop age of type string supplied to Component, expected number”.
  + This is great since we can now guarantee that the age prop is always a number. You will also notice that there are no warnings for missing props even though we defined PropTypes for address and friends. This is because by default all props are optional. In order to make a prop required we can chain isRequired to the end of the PropType.
  + Text

    Description automatically generated
  + With the above code we have marked our name, age, and address props as required which means now if we do not pass an address for example we will get the following error. “Warning: Failed prop type: The prop address is marked as required in Component, but its value is undefined”
  + Also, here is a list of all the basic PropTypes for checking JavaScript primitives:
  + Text

    Description automatically generated
* React Specific Prop Types
  + On top of being able to check primitive types, you can also check some React specific things as well.
  + Checking For Renderability
  + If you want to simply check to see if a component can be rendered you can use the PropTypes.node check. This will check to see if the prop is a type that React can render. Things such as strings, numbers, elements, and arrays are common renderable types. Objects are not renderable. A renderableProp should be able to do this <div>{prop.renderableProp}</div>.
  + Text

    Description automatically generated
  + Checking For A React Component
  + If you want to just check to see if a prop is a React component you can use PropTypes.element. This is useful for ensuring that a component only ever has exactly one child component, for example.
  + Text

    Description automatically generated
  + Note that having no children/more than 1 child will result in an error. So something like Text

    Description automatically generated with medium confidence will raise a console warning.
  + Checking For A React Component Name
  + Finally, you can check to see if your prop is the name of a React component by using PropTypes.elementType.
  + Text

    Description automatically generated
  + 
* Advanced Type Checking
  + PropTypes also contain a ton of different advanced type checking techniques.
  + Checking For Any Type
  + Sometimes you just want to ensure a prop is required without specifying a specific type. This is where PropTypes.any comes in handy since it will never throw a warning for the type.
  + Text

    Description automatically generated
  + Checking For Multiple Types
  + If you don't like the idea of using any, but also have a prop that could be multiple types you can use PropTypes.oneOfType() to specify which types the prop can be. This just takes an array of valid PropTypes.
  + Text

    Description automatically generated
  + Creating An Enum
  + If you want to ensure that a prop's value is from a specific list you can use PropTypes.oneOf() to define the allowed values. This is perfect for creating an enum.
  + Text

    Description automatically generated
  + Checking Array Types
  + When you have an array where you know the type of the elements, you can use PropTypes.arrayOf() to specify the type of the elements. This works very similar to PropTypes.array, but it also type checks the elements in the array as well.
  + Text

    Description automatically generated
  + Checking Object Types
  + Just like with arrays, you can also check the types of the individual properties of an object with PropTypes.shape(). This is really useful when passing large objects down to a component.
  + Text

    Description automatically generated
  + This above code will only throw a warning if one of the specified properties of address does not follow its PropType. If address has additional properties on it, such as as zip code, that will not cause any warnings. If you did want to throw a warning, though, when the object has properties not specified in the PropTypes you could use PropTypes.exact() instead. It works exactly the same as PropTypes.shape(), but it will throw a warning if the object has properties not defined in the PropTypes.
  + Text

    Description automatically generated

Context

* Consider a react application with the following component structure.
* Diagram

  Description automatically generated
* Suppose we want components A D and F to display a username. The value of this username is maintained as a property in the AppComponent. In order for components A D and F to get this value, the value of the username have to be passed through components B, C, E as well even though they don’t use the value of the username at all. Having middlemen components can make the code very messy.
* This is where context comes into play.
* Context provides a way to pass data through the component tree without having to pass props down manually at every level.
* In the following example, we will pass the value of the username from the App component to the F component.
* The initial code will look like the following to show the tree hierarchy of the components.
* Graphical user interface

  Description automatically generated Graphical user interface

  Description automatically generated with medium confidence Graphical user interface

  Description automatically generated with low confidence Graphical user interface, text

  Description automatically generated
* There are three steps to implement Context.
  + 1. Create the context
  + 2. Provide a context value
  + 3. Consume the context value
* Creating the context
  + Text

    Description automatically generated
  + We create a new .js file for the context.
  + We use the React.createContext() method from react to create a context object.
  + Every context object created with createContext() comes with a Provider and a Consumer component.
  + We need to use them for steps 2 and 3 so we need to export the Provider and Consumer component. When we export, we can change their names to something easier to read.
* Providing a context value
  + Text

    Description automatically generated
  + We need to provide this userContext using the UserProvider component. Only the descendant components of the UserProvider component can consume the context value. Make sure to import the UserProvider at the top.
  + App component is usually a good place to add your Provider since all components are usually descendants of the App component.
  + To initialize the context value that we want the descendent components to consume, we add the ‘value’ prop to the UserProvider component and set its value to whatever we want.
* Consuming a context value
  + Text

    Description automatically generated
  + To consume a context value, we need to use the Consumer component. So in component F, we can include the UserConsumer method and make sure to import it.
  + In between the opening and closing tags of the Consumer component, we need to specify a function. The function gets the context value as its parameter and the function should return JSX. In our example, the value represents the username so we say username is the parameter of the function.
* Now, the webpage output should be: 
* We can also have many contexts. Consider the following example.
* Text

  Description automatically generated
* Text

  Description automatically generated
* Notice that consuming multiple context values is a nightmare. useContext allows you to consumer context values in a more succinct way.

**useContext**

* Note that creating and providing a context remains unchanged, useContext only makes consuming multiple contexts more succinct.
* To consume a context value, we must first import useContext from React.
* Then, we must import the necessary contexts
* Then call the useContext function, passing in the context as its argument. The useContext function returns a context value so we can store that in a constant.
* Text

  Description automatically generated
* Note that the parameter to the useContext function is UserContext and not UserConsumer
* Note that the value of user and channel remains constant from the time after the previous render to the time before the next render. But when the component rerenders, the user and channel constants are reinitialized since these values do not persist through renders such as state.

<https://reactjs.org/docs/context.html#updating-context-from-a-nested-component>

**useReducer**

* useReducer is a hook that is used for state management. It is an alternative to useState which becomes useful when you have to track multiple pieces of state. In fact, useState is built using useReducer.
* useReducer relates to the reduce array method in JS. Recall the reduce array method takes in a reducer function as a parameter. This reducer function has an accumulator and the current item as its parameters and returns a single value.
* Table

  Description automatically generated
* Ex: Suppose we need to create 3 buttons that increment, decrement, and reset a count.
* Text

  Description automatically generated
* To use useReducer, we first need to import it from react: 
* We define a constant object called ACTIONS for good constant style.
* We define initialState which is 0. We define a reduce function that takes in two parameters, state and action. We use the action parameter in a switch statement return different values depending on the what the value of action is.
* To use useReducer, we pass in the reducer function and the initialState as arguments. This useReducer method returns and array of two elements when executed. The first element is the current state so ‘count’ is the current state. The second element is a function that takes an action as its parameter which in our case is ‘dispatch’.
* Dispatch is a void method that does not have a return value. When calling the dispatch function, we pass in an action as an argument. This will execute the reducer function that we passed into useReducer. The reduce function will take the current state and the action passed to the dispatch method as parameters and return the new value of the state.
* useReducer shines when the state is complex (such as when the state is an object and not a string/num).
* Text

  Description automatically generated
* Text

  Description automatically generated
* Notice that in the handleComplete function, we are passing in a object with many properties to the dispatch method.
* Notice in the reducer function, we use the different properties of the action object when determining what to return.
* Notice in the reducer function if action.type === ‘COMPLETE’ , we return a new array that will be the new value of the ‘todos’ state array variable. We map through all the elements in todos if the todo.id === action.id, we toggle the Boolean value of the complete property of the current todo in the map. Graphical user interface, text

  Description automatically generated
* Notice that the return in the reducer function use the spread operator  to return a new object (in terms of both reference and value) that will be an element in the new ‘todos’ state array variable.

**Custom Hooks**

* Custom hooks are reusable JavaScript functions. When you have component logic that needs to be used by multiple components, we can extract that logic to a custom Hook. This is an alternative to HOC and render props.
* By convention, the hook name starts with ‘use’ such as ‘useCounter’ or ‘useFetch’.
* A custom hook can call other hooks if needed.
* Generally, custom hooks don’t return JSX, but it’s possibly they do.
* Ex:
  + Graphical user interface, text, application

    Description automatically generated
  + The logic of fetching data from a url may be needed in other components as well, so we will extract that into a custom Hook.
  + Move the fetch logic to a new file to be used as a custom Hook:
  + Graphical user interface, text, application

    Description automatically generated
  + Now we can use the useFetch hook in any component including the Home component
  + Graphical user interface, text, application, chat or text message, email

    Description automatically generated
* Ex:
  + Suppose we have the following code:
  + A screenshot of a computer

    Description automatically generated with low confidence Text

    Description automatically generated
  + Text

    Description automatically generated
  + Instead of having the above, we could have the following:
  + Text

    Description automatically generated
  + Text

    Description automatically generated
  + Text

    Description automatically generated
  + Output: 
  + Note that there is no state conflict. Changing the count of a component doesn’t change the count of the other component.