Introduction To React, React Bootstrap, and JSX

Introduction: 2
JSX: 2
React: 2
    DOM: 2
    Component Based Architecture: 3
    State Management: 3
    Props: 7
    Routing: 8
    Fetching data: 9
React-Bootstrap: 11
Project: 11

Introduction:

Hello! If you are reading this document, you are probably interested in getting started with React (the best framework BTW) but you may not know how to start or what to learn first. I once shared that pain and have created this tutorial to help you get started. So, what will you learn? Here is a brief overview of the content:

- **JSX**: What is JSX? How is it related to React?
- **React:** I will cover what React is and go over the fundamentals.
- **React-Bootstrap:** React-Bootstrap is a library for React that provides several useful tools, the most important being the intuitive grid system which mitigates the need for an excessive amount of divs, allowing you to focus on your applications core functionality rather than its layout.

JSX:

JSX, or JavaScript XML, is an extension to JavaScript that allows developers to write HTML-like syntax within their JavaScript code. It serves as a convenient and expressive way to define the structure of user interfaces in React applications. JSX resembles HTML, but it is not actual HTML. Instead, it gets transformed into regular JavaScript function calls. JSX makes React code more readable and intuitive by enabling developers to write UI components in a format that closely resembles the final output. It allows for the interpolation of JavaScript expressions, making it easy to dynamically generate content and handle logic within UI components. JSX also seamlessly integrates with JavaScript, enabling developers to utilize the full power of the language to manipulate and manage UI elements. While this tutorial does not show standalone JSX, when we go over React components it will become quite apparent how JSX is implemented.

React:

DOM:

React is a popular JavaScript library for building user interfaces, developed by Facebook. At its core, React utilizes a virtual DOM (Document Object Model) to efficiently manage and update the UI. The virtual DOM is a lightweight, in-memory representation of the actual DOM tree structure. When changes are made to the UI in a React application, React first updates the virtual DOM rather than directly manipulating the browser's DOM. This approach offers significant performance benefits, as React can batch updates and minimize the number of actual DOM manipulations required. Once all updates are processed in the virtual DOM, React compares it with the actual DOM and only applies the necessary changes to the browser, resulting in faster rendering and improved efficiency. The DOM, or Document Object Model, is a programming interface that represents the structure of HTML and XML documents as a tree of nodes. It provides a way for scripts to dynamically access and modify the content, structure, and style of documents. In web development, understanding and efficiently manipulating the DOM is crucial for building dynamic and interactive web applications.

It is not important for you to be an expert on the DOM, but you should know what it is and how React interacts with it from a high level. This will allow you to better understand the system.

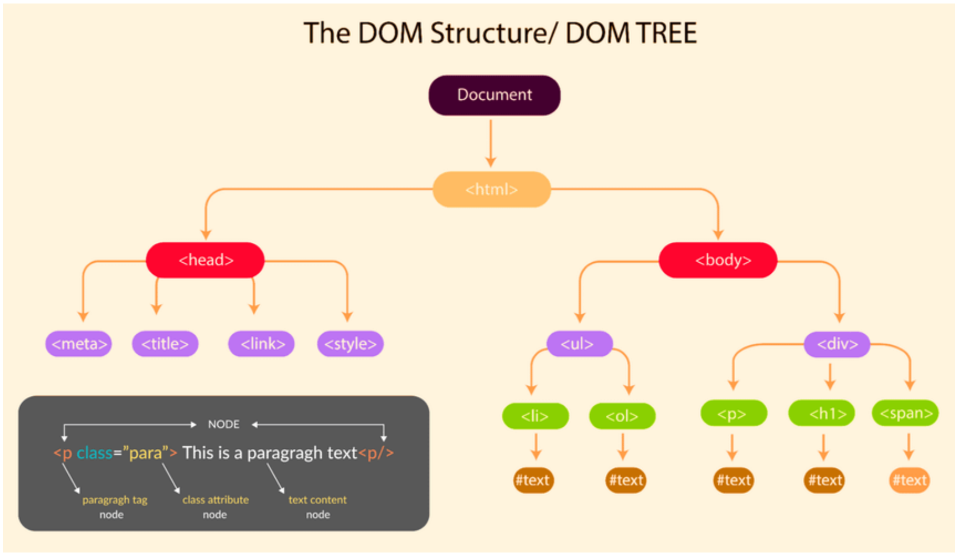
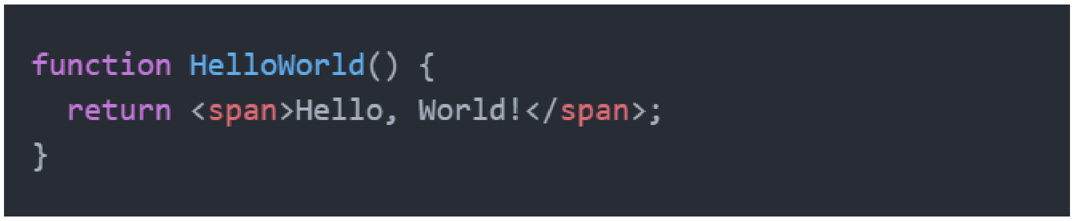


Figure 1: Example DOM Structure [1]

Component Based Architecture:

React is built around the concept of reusable UI components. Components encapsulate both structure and behavior, allowing developers to create modular and maintainable codebases. Components can be composed together to build complex user interfaces.



```
function HelloWorld() {
  return <span>Hello, World!</span>;
}
```

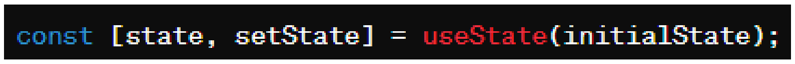
Figure 2: Example of a simple React component [2]

State Management:

React provides a mechanism for managing the state of components. State represents data that can change over time, such as user input or the result of asynchronous operations. React's state management enables developers to update the UI in response to user actions or other events. With React, you won't modify the UI from code directly. For example, you won't write commands like "disable the button", "enable the button", "show the success message", etc. Instead, you will describe the UI you want to see for the different visual states of your component ("initial state", "typing state", "success state"), and then trigger the state changes in response to user input. This is how designers think about UI.

useState():

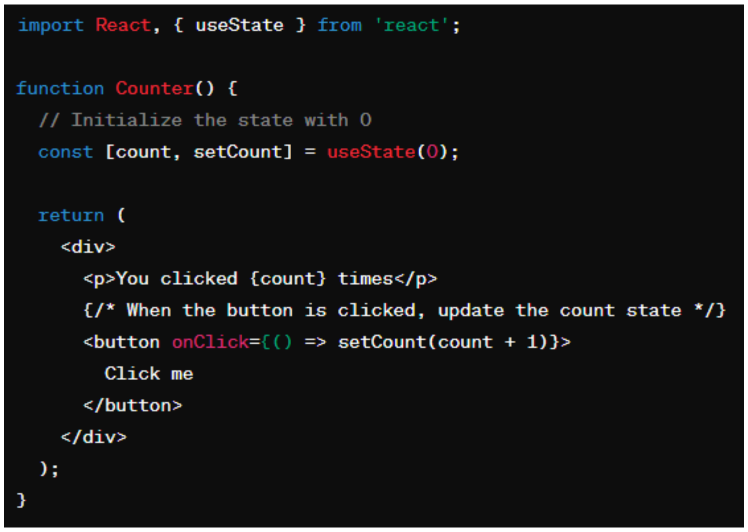
The **useState** hook is a function provided by React that returns a stateful value and a function to update that value. It takes one argument, the initial state value. The syntax for using the **useState** hook is as follows:



```
const [state, setState] = useState(initialState);
```

- **state**: This variable holds the current state value. It's similar to the state property in class components. When you want to access the current state value, you use this variable.
- **setState**: This function is used to update the state. When you call **setState** with a new value, React re-renders the component, and the **state** variable will hold the updated value.
- **initialState**: This is the initial value of the state. It's only used during the first render of the component. The initial state can be a primitive value like a number or string, or it can be an object or an array.

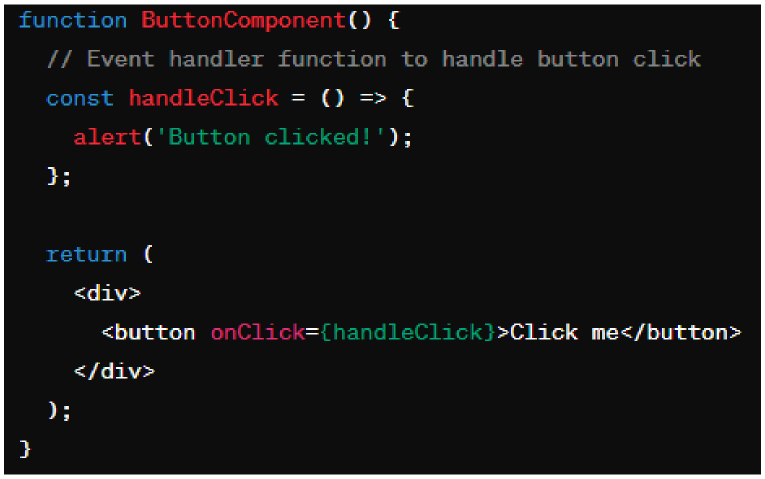
Here is an example of a simple **useState** implementation:



```
import React, { useState } from 'react';

function Counter() {
  // Initialize the state with 0
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      {/* When the button is clicked, update the count state */}
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

Event Handlers:

Event handlers in React are functions that are used to handle user interactions, such as clicking a button, submitting a form, or typing into an input field. These interactions trigger events and React allows developers to define **event handler** functions to respond to these events and update the UI accordingly. To add an **event handler,** you will first define a function and then pass it as a prop to the appropriate JSX tag.



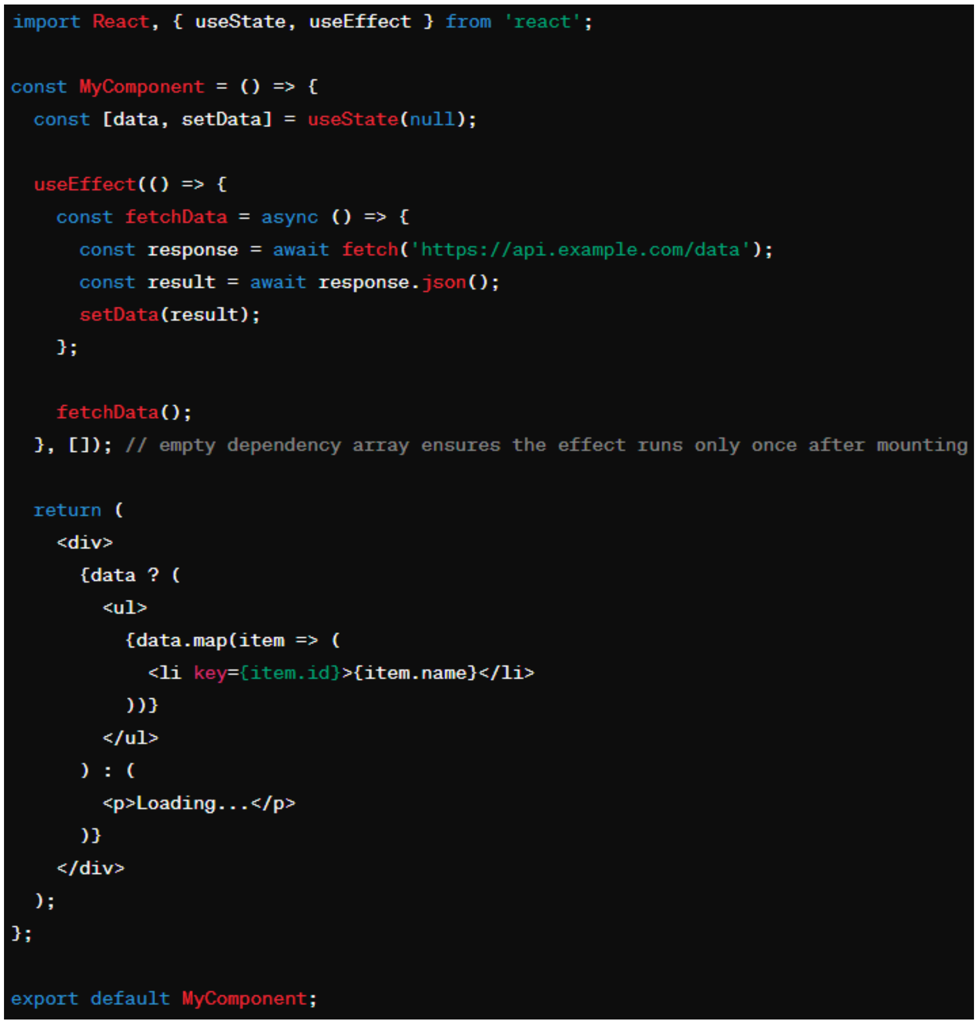
```
function ButtonComponent() {
  // Event handler function to handle button click
  const handleClick = () => {
    alert('Button clicked!');
  };

  return (
    <div>
      <button onClick={handleClick}>Click me</button>
    </div>
  );
}
```

Often your event handlers will do one of two things: update a state variable or send some alert to the user. The complexity of your event handlers is not limited. At some point your event handlers will start to make API calls which will allow for more dynamic and customizable state variables. Below I have listed some of the most common types of event handler JSX tags (i.e. "onClick" or "onChange"):

- onClick
- onChange
- onSubmit
- onKeyDown/onKeyUp/onKeyPress
- onMouseEnter/onMouseLeave
- onFocus/onBlur
- onScroll
- onLoad/onError

UseEffecct():

The useEffect() hook in React is a fundamental tool for managing side effects in functional components. This hook enables developers to perform tasks that are typically performed in class-based lifecycle methods, such as fetching data from APIs, subscribing to events, or updating the DOM, in a functional component. useEffect() takes two arguments: a function representing the side effect to perform, and an optional dependency array. The dependency array is crucial as it dictates when the effect should run. If provided, useEffect() will re-run the effect whenever any value in the dependency array changes, ensuring that the side effect is kept in sync with the component's state. Omitting the dependency array causes the effect to run after every render. A simple use case of useEffect() could involve fetching data from an API when the component mounts, like so:

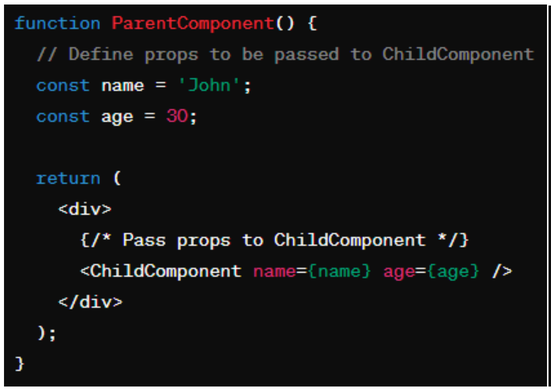


```
import React, { useState, useEffect } from 'react';

const MyComponent = () => {
  const [data, setData] = useState(null);

  useEffect(() => {
    const fetchData = async () => {
      const response = await fetch('https://api.example.com/data');
      const result = await response.json();
      setData(result);
    };

    fetchData();
  }, []); // empty dependency array ensures the effect runs only once after mounting

  return (
    <div>
      {data ? (
        <ul>
          {data.map(item => (
            <li key={item.id}>{item.name}</li>
          ))}
        </ul>
      ) : (
        <p>Loading...</p>
      )}
    </div>
  );
};

export default MyComponent;
```

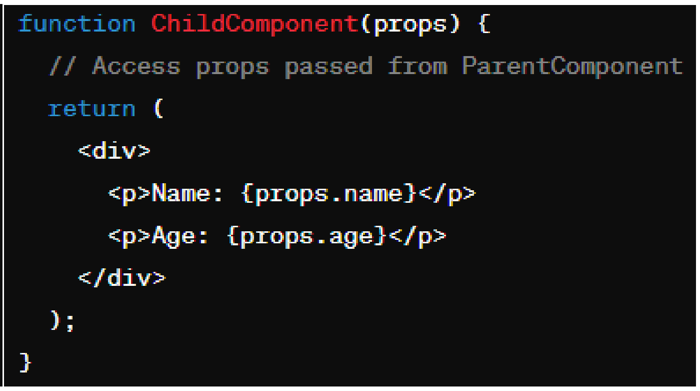
Props:

In React, props (short for "properties") are a mechanism for passing data from a parent component to a child component. They allow you to customize and configure child components, making them more reusable and versatile. Props are similar to function arguments or HTML attributes. They are passed down from parent components to child components as attributes in JSX. Once received by a child component, props are accessed as properties of the **props** object.

Here's a basic example demonstrating the usage of props in React:



```
function ParentComponent() {
  // Define props to be passed to ChildComponent
  const name = 'John';
  const age = 30;

  return (
    <div>
      {/* Pass props to ChildComponent */}
      <ChildComponent name={name} age={age} />
    </div>
  );
}
```



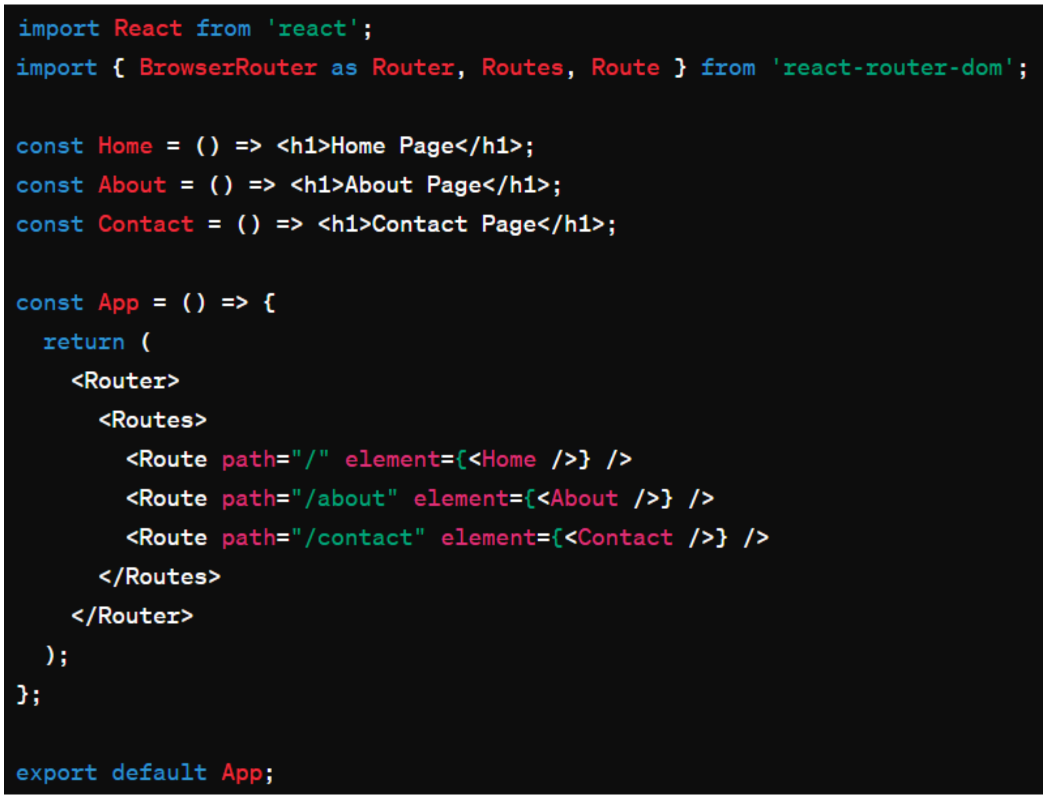
```
function ChildComponent(props) {
  // Access props passed from ParentComponent
  return (
    <div>
      <p>Name: {props.name}</p>
      <p>Age: {props.age}</p>
    </div>
  );
}
```

In this example, the **ParentComponent** passes two props (**name** and **age**) to the **ChildComponent**. These props are accessed within the **ChildComponent** using the **props** object. The **ChildComponent** then renders the received props, displaying the name and age passed from the parent. Props are immutable, meaning that child components cannot modify their props directly. They are intended to be read-only, making components predictable and easier to reason about.

Routing:

Routing in React refers to the process of navigating between different views or pages within a single-page application (SPA). In traditional multi-page applications, navigation involves loading entirely new HTML pages from the server. However, in SPAs built with React, routing enables dynamic content changes without full page reloads, providing a smoother and more interactive user experience. Routing is crucial in React applications as it allows developers to organize their code into separate components representing different views, making the application structure more modular and maintainable. Additionally, routing facilitates deep linking, bookmarking, and browser history manipulation, ensuring better accessibility and usability for users.

In the example below Router, Routes, and Route are imported from react-router-dom. These modules allow us to construct a route tree in our App.js. Each route within this route tree has a specified path. You can make these routes accessible in any manner you would like throughout your web application but within App.js they should be inside of a “Router”.

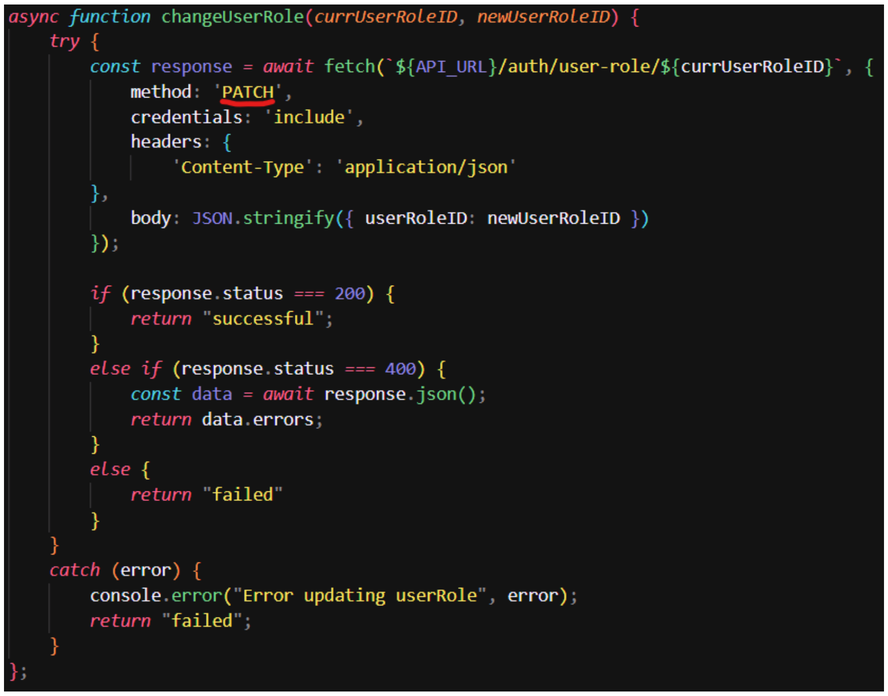


```
import React from 'react';
import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';

const Home = () => <h1>Home Page</h1>;
const About = () => <h1>About Page</h1>;
const Contact = () => <h1>Contact Page</h1>;

const App = () => {
  return (
    <Router>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
        <Route path="/contact" element={<Contact />} />
      </Routes>
    </Router>
  );
};

export default App;
```

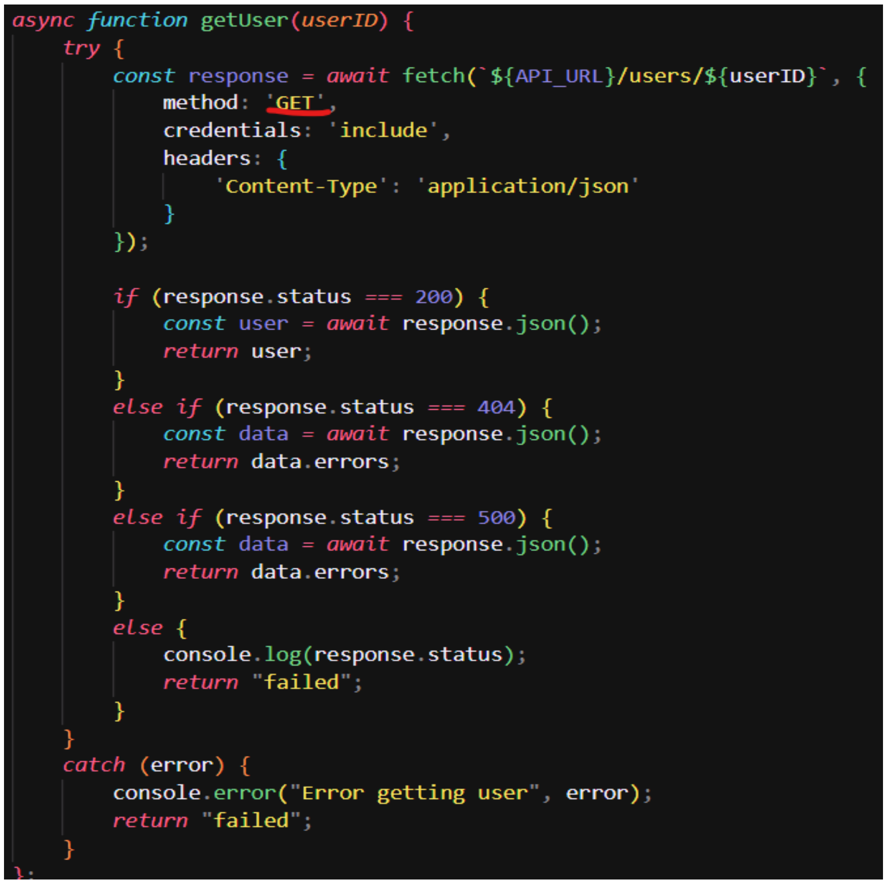
Typically, each page will be constructed elsewhere and then imported into App.js. These pages have the same properties as every other react component and can even contain other react components within them.

Fetching data:

The "fetch" API in JavaScript allows us to make API calls within our web applications. Typically fetch is used to make HTTP requests to the applications server. In your first project you used Node.js to make a RESTful API and then used fetch to make an HTTP request. There are various types of HTTP requests:



```
async function changeUserRole(currUserRoleID, newUserRoleID) {
    try {
        const response = await fetch(`${API_URL}/auth/user-role/${currUserRoleID}`, {
            method: 'PATCH',
            credentials: 'include',
            headers: {
                'Content-Type': 'application/json'
        },
            body: JSON.stringify({ userRoleID: newUserRoleID })
        });

        if (response.status === 200) {
            return "successful";
        }
        else if (response.status === 400) {
            const data = await response.json();
            return data.errors;
        }
        else {
            return "failed"
        }
    }
    catch (error) {
        console.error("Error updating userRole", error);
        return "failed";
    }
};
```



```
async function getUser(userID) {
    try {
        const response = await fetch(`${API_URL}/users/${userID}`, {
            method: 'GET',
            credentials: 'include',
            headers: {
                'Content-Type': 'application/json'
            }
        });

        if (response.status === 200) {
            const user = await response.json();
            return user;
        }
        else if (response.status === 404) {
            const data = await response.json();
            return data.errors;
        }
        else if (response.status === 500) {
            const data = await response.json();
            return data.errors;
        }
        else {
            console.log(response.status);
            return "failed";
        }
    }
    catch (error) {
        console.error("Error getting user", error);
        return "failed";
    }
};
```

React-Bootstrap:

React-Bootstrap is a popular front-end framework that combines the power of React with the versatility of Bootstrap's components. One of its most notable features is the grid system, which provides a flexible and responsive layout structure for building web interfaces. The grid system allows developers to create complex layouts by dividing the page into rows and columns, making it easy to organize content across different screen sizes. This utility becomes especially crucial in modern web development, where responsiveness across various devices is paramount. React-Bootstrap's grid system simplifies the process of creating responsive designs, as it automatically adjusts content placement based on the screen size, ensuring a consistent user experience across desktops, tablets, and smartphones. Whether it's crafting a simple two-column layout or a more intricate multi-column design, the grid system in React-Bootstrap empowers developers to create visually appealing and user-friendly interfaces with ease.

Read the "Layout" section of the react-bootstrap documentation:
https://react-bootstrap.netlify.app/docs/layout/breakpoints

Project:

In the first project you made a single webpage with some text boxes to gather input information. You then sent this information to the backend to be sorted before receiving the sorted information and displaying it to the user. Unfortunately, this was not enough for your client, and they are requesting that more features be added to the website before deployment. They have requested the following:

A second webpage that allows the user to view previously submitted word combinations.

Here are some guidelines/hints to get you started:

- once the second page component is made there should be some type of link on each page to navigate to the other page.
- These pages should be placed inside of a router in your App.js file.
- You will need to store each submitted word combination into a JSON file (on the backend) using POST.
- The second page MUST use the react-bootstrap grid system, feel free to explore the documentation and add other things as well.
- You will need to fetch the submission log before the page renders (using GET) and account for the time that it will take to fetch the information. Hint: use UseEffect()
- Information can be displayed in any manner, table or list, I recommend a scrollable table with 3 columns, but it is up to you.
- You are encouraged to use ChatGPT as much as you need!

Have Fun!!!

Works Cited

[1] A. Anie, “What is DOM: A complete guide to document object model,” *LambdaTest*, Dec. 05, 2023. https://www.lambdatest.com/blog/document-object-model/

[2] D. Pavlutin, “A simple guide to component props in React,” *Dmitri Pavlutin Blog*, Jul. 02, 2021. https://dmitripavlutin.com/react-props/