Project – 13

React JS

Submitted By:

Nishigandha Patil

INDEX

INTRODUCTION TO REACT

➢ **Introduction:**

- **What is React?**
  React is an open-source JavaScript library used for building user interfaces or UI components, particularly for single-page applications where user interactions are dynamic and frequent. React allows developers to create reusable UI components that update efficiently and seamlessly in response to data changes.

- **Why React?**
  React simplifies the process of building UIs by using a component-based architecture, making it easier to manage complex UIs and their states.

- **React ecosystem overview:**
  React is complemented by a vast ecosystem of tools and libraries such as 'React Router' for routing, 'Redux' for state management, and 'React Native' for building mobile applications.

➢ **Setting Up Your Development Environment:**

- **Installing Node.js and npm:**
  Node.js is a JavaScript runtime environment, and npm is the package manager for Node.js. They are essential for setting up a React development environment.

- **Create React App:**
  Create React App is a tool that sets up a new React project with a pre-configured development environment, allowing you to start building React apps quickly.

- **Project structure overview:**
  Understanding the structure of a typical React project helps you navigate and organize your code effectively.

➢ **Creating React App:**

Open your terminal or command prompt and run the following command to create a new React app using Create React App:

```
> npx create-react-app my-app
```

Once the app is created, navigate into the project directory:

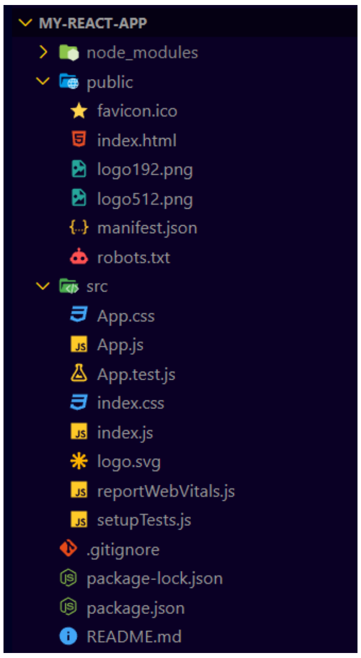
```
> cd my-app
```

Start the development server by running the following command:

```
> npm start
```

You should see a basic React app running in your browser at http://localhost:3000.

➢ Folder Structure:



```
MY-REACT-APP
  > node_modules
  v public
      favicon.ico
      index.html
      logo192.png
      logo512.png
      manifest.json
      robots.txt
  v src
      App.css
      App.js
      App.test.js
      index.css
      index.js
      logo.svg
      reportWebVitals.js
      setupTests.js
    .gitignore
    package-lock.json
    package.json
    README.md
```

- **public/index.html:**

Serves as the entry point for your React application. This file is a static HTML file that contains the basic structure of your web page, and it's where your React app is injected when it's built and served to the browser.

- **src/App.js:**

Serves as the root component of your application. It's where you define the structure and layout of your app, as well as manage its state and behavior.

➢ VS Code Extensions for fast Coding:

1. ES7+ React/Redux/React-Native snippets
2. Auto Rename Tag
3. Thunder Client: To test API.
4. Prettier
5. Live Server

JSX (JAVASCRIPT XML)

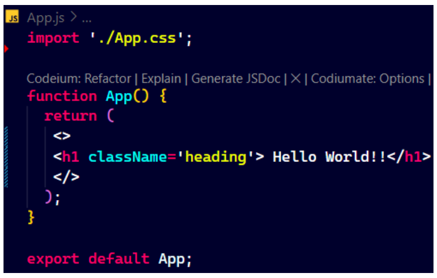
JSX is a Syntax extension for JavaScript that allows you to write HTML-like code within your JavaScript files. It is used to describe what the UI should look like in React applications. JSX syntax resembles HTML but allows you to embed JavaScript expressions within curly braces {}.

Jsx wrapper: < >…</>

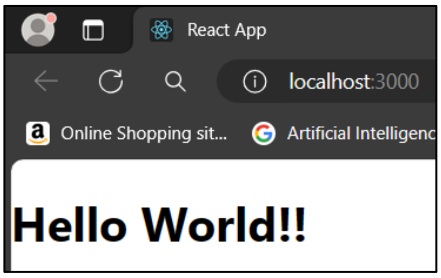
➢ Writing markup with JSX

JSX is stricter than HTML. You have to close tags like <br /> <hr />. Your component also can't return multiple JSX tags. You have to wrap them into a shared parent, like a <div>...</div> or an empty < >...</> wrapper.

Example:



```
App.js > ...
import './App.css';

Codeium: Refactor | Explain | Generate JSDoc | X | Codiumate: Options |
function App() {
  return (
    <>
    <h1 className='heading'> Hello World!!</h1>
    </>
  );
}

export default App;
```

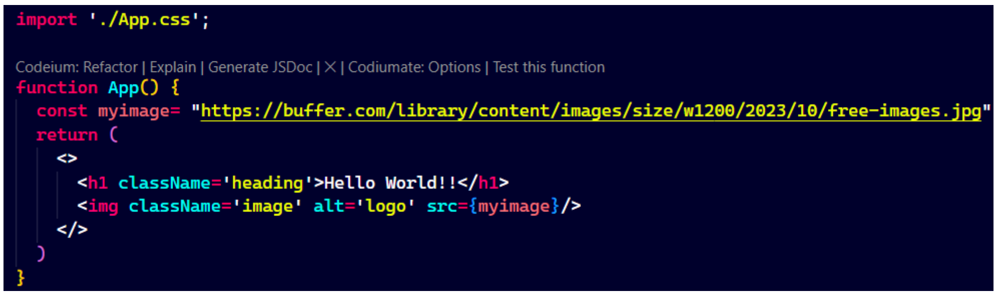


Hello World!!

➢ Adding styles and Displaying Data:

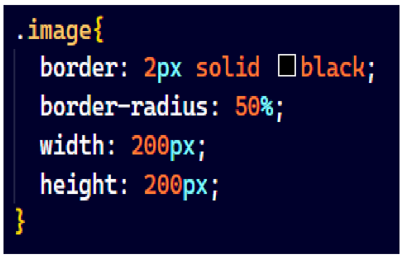
In React, you specify a CSS class with **className**. It works the same way as the HTML class attribute:

App.js:



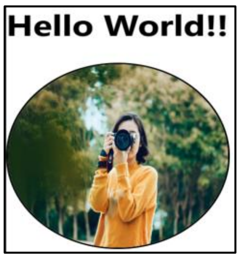
```
import './App.css';

Codeium: Refactor | Explain | Generate JSDoc | X | Codiumate: Options | Test this function
function App() {
  const myimage= "https://buffer.com/library/content/images/size/w1200/2023/10/free-images.jpg"
  return (
    <>
      <h1 className='heading'>Hello World!!</h1>
      <img className='image' alt='logo' src={myimage}/>
    </>
  )
}
```

App.css:



```
.image{
  border: 2px solid black;
  border-radius: 50%;
  width: 200px;
  height: 200px;
}
```

Output:



Hello World!!

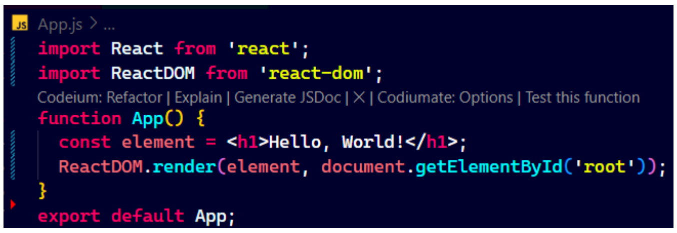
REACT DOM

In React, the ReactDOM library is used to interact with the DOM (Document Object Model) of a web page. It provides methods for rendering React elements into the DOM, updating them, and handling events. ReactDOM is essential for integrating React components with the web page's structure and behavior.

Installation: npm install react react-dom

Rendering React Elements:

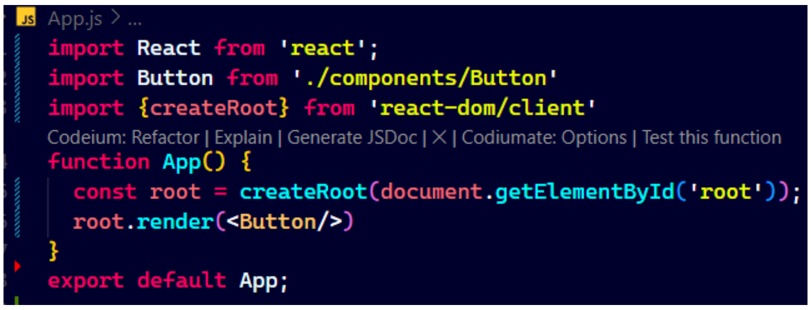
The **ReactDOM.render()** method is used to render React elements into the DOM. It takes two arguments: the React element to render and the DOM node where the element should be rendered.



```
App.js > ...
import React from 'react';
import ReactDOM from 'react-dom';
Codeium: Refactor | Explain | Generate JSDoc | X | Codiumate: Options | Test this function
function App() {
  const element = <h1>Hello, World!</h1>;
  ReactDOM.render(element, document.getElementById('root'));
}
export default App;
```

Note: ReactDOM.render is no longer supported in React 18. Your app will behave as if it's running React 17.

The **createRoot** function is a new API introduced in React 18 to create a new root-level render. It's intended to replace the existing ReactDOM.render() method for creating the root of a React application.



```
App.js > ...
import React from 'react';
import Button from './components/Button'
import {createRoot} from 'react-dom/client'
Codeium: Refactor | Explain | Generate JSDoc | X | Codiumate: Options | Test this function
function App() {
  const root = createRoot(document.getElementById('root'));
  root.render(<Button/>)
}
export default App;
```

In this example:

- We import **createRoot from react-dom**.
- We create a new root using createRoot, passing the DOM element where we want to render the application.
- We call the render method on the root, passing the root component of our application (<Button/> in this case) to render it into the DOM.

If you need to remove a React component from the DOM, you can use the **root.unmount()** method. It takes the DOM node containing the component as an argument and removes the component from the DOM.

COMPONENTS

React apps are made out of components. A component is a piece of the UI (user interface) that has its own logic and appearance. A component can be as small as a button, or as large as an entire page.

React components are JavaScript functions that return markup.

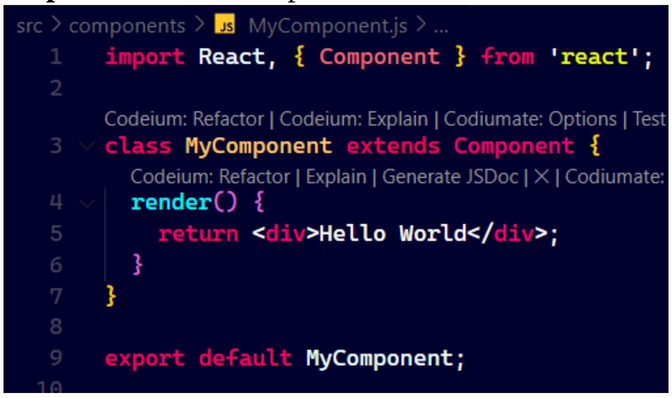
There are two types of components:

1. **Class Based Components**
   Class components are ES6 classes that extend from the React. Component class. They have more features than functional components, such as lifecycle methods and local state management.

   Step 1: Create **components** folder in src.

   Step 2: Create a file - MyComponent.js. Type **rcc** for snippets.
   Note: React component names must always start with a capital letter.

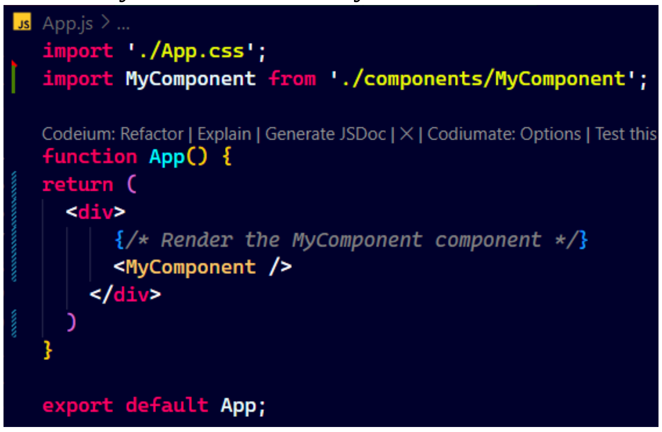
   Step 3: Write the component code:



src > components > MyComponent.js > ...
Codeium: Refactor | Codeium: Explain | Codiumate: Options | Test
Codeium: Refactor | Explain | Generate JSDoc | X | Codiumate:

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

class MyComponent extends Component {
  render() {
    return <div>Hello World</div>;
  }
}

export default MyComponent;
```

   Step 4: Use the component
   You can now import and use the MyComponent component in other files within your project. For example, you might import and render MyComponent within your **App.js** file or any other file where you want to include it.



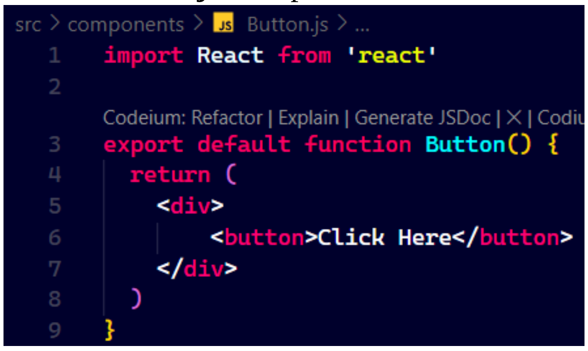
App.js > ...
Codeium: Refactor | Explain | Generate JSDoc | X | Codiumate: Options | Test this

```
import './App.css';
import MyComponent from './components/MyComponent';

function App() {
  return (
    <div>
      {/* Render the MyComponent component */}
      <MyComponent />
    </div>
  )
}

export default App;
```

2. **Function Based Components**

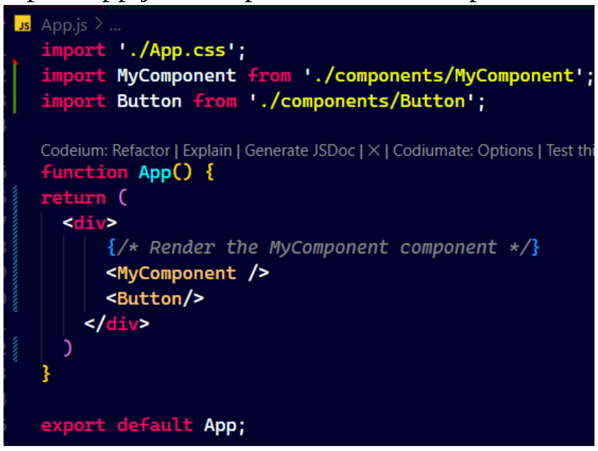
Functional components are JavaScript functions that take props (short for properties) as an argument and return React elements. They are simple and lightweight, making them easy to understand and maintain.

Create **Button.js** Component → Write code. Type **rfc** for snippets.



```
src > components > Button.js > ...
import React from 'react'

Codeium: Refactor | Explain | Generate JSDoc | X | Codiu
export default function Button() {
  return (
    <div>
        <button>Click Here</button>
    </div>
  )
}
```

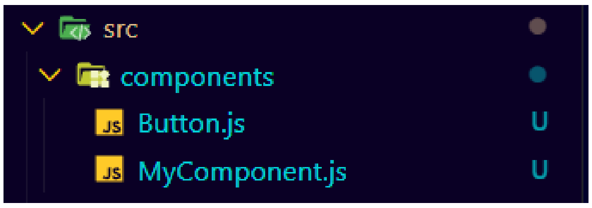
Open App.js → import **Button** Component.



```
App.js > ...
import './App.css';
import MyComponent from './components/MyComponent';
import Button from './components/Button';

Codeium: Refactor | Explain | Generate JSDoc | X | Codiumate: Options | Test thi
function App() {
return (
  <div>
      {/* Render the MyComponent component */}
      <MyComponent />
      <Button/>
    </div>
  )
}

export default App;
```

The **export default** keywords specify the main component in the file.

Folder Structure:



```
src
  components
    Button.js        U
    MyComponent.js   U
```

➢ **Stateful and Stateless Components:**

1. **Stateful Components:**

Stateful components, also known as "class components," are components that manage their own state using the setState method. They are defined using ES6 classes and extend the React.Component class. Stateful components have their own internal state, which can be modified throughout the component's lifecycle.

2. **Stateless Components:**

Stateless components, also known as "functional components," are components that do not manage their own state. They are defined using JavaScript functions and typically accept props as input and return JSX as output. Stateless components are primarily concerned with presenting UI based on the data they receive as props.

- **Choosing Between Stateful and Stateless Components:**
  Use “stateful components” when you need to manage complex state or lifecycle methods (e.g., componentDidMount, componentDidUpdate, etc.).

  Use “stateless components” for simple UI components that do not require state management or lifecycle methods. They are easier to understand, test, and maintain.

  In modern React development, functional components are preferred over class components due to their simplicity and performance benefits.

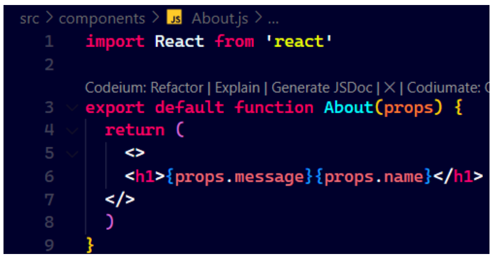
- **Components lifecycle method:**
  A component's lifecycle refers to the different stages a component goes through from its creation to its destruction. These stages are represented by various lifecycle methods which you can override in your component classes. With the introduction of React Hooks, functional components can also utilize lifecycle methods using hooks like **useEffect.**

  Component lifecycle in React, divided into three main phases: mounting, updating, and unmounting.

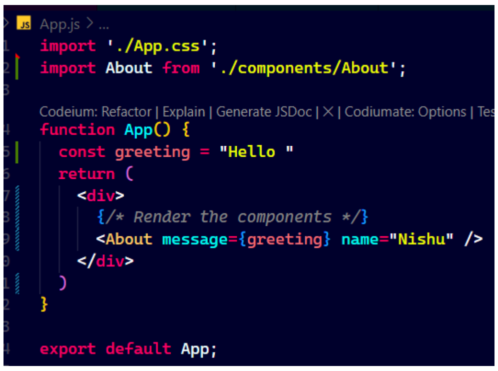
PROPS AND STATES

1. **Props:**
   props (short for properties) are a mechanism for passing data from a parent component to a child component. They are a fundamental part of React's component model and are used to make components reusable and modular.

   Example:


src > components > About.js > ...
Codeium: Refactor | Explain | Generate JSDoc | X | Codiumate:

```
import React from 'react'

export default function About(props) {
  return (
    <>
    <h1>{props.message}{props.name}</h1>
  </>
  )
}
```

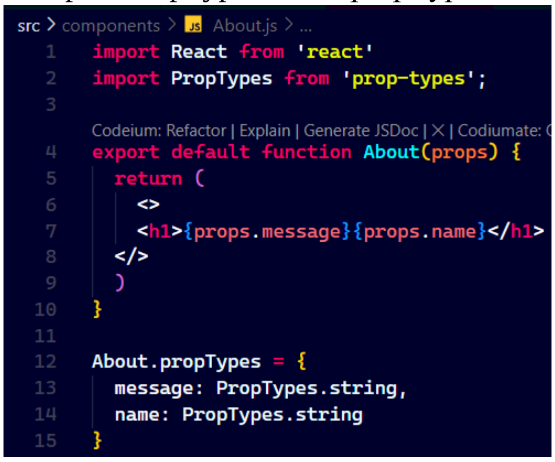

App.js > ...
Codeium: Refactor | Explain | Generate JSDoc | X | Codiumate: Options | Tes

```
import './App.css';
import About from './components/About';

function App() {
  const greeting = "Hello "
  return (
    <div>
      {/* Render the components */}
      <About message={greeting} name="Nishu" />
    </div>
  )
}

export default App;
```

In this example, the **message** prop is passed dynamically based on the value of the **greeting** variable.

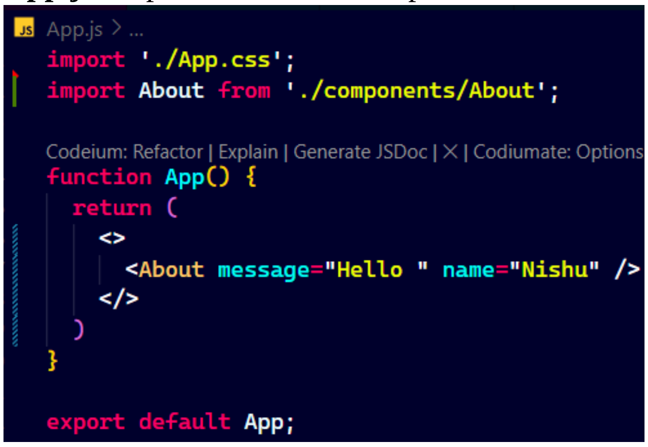
➢ **Prop types:** Used to specify the expected types of props.
PropTypes is a feature provided by React to validate the props passed to a component, helping developers catch bugs and enforce data type constraints in their applications.

Example:
> Import PropTypes from ‘prop-types’:


src > components > About.js > ...
Codeium: Refactor | Explain | Generate JSDoc | X | Codiumate:

```
import React from 'react'
import PropTypes from 'prop-types';

export default function About(props) {
  return (
    <>
    <h1>{props.message}{props.name}</h1>
  </>
  )
}

About.propTypes = {
  message: PropTypes.string,
  name: PropTypes.string
}
```

App.js: import the **About** Component.


App.js > ...
Codeium: Refactor | Explain | Generate JSDoc | × | Codiumate: Options

```
import './App.css';
import About from './components/About';

function App() {
  return (
    <>
      <About message="Hello " name="Nishu" />
    </>
  )
}

export default App;
```

➢ **Available PropTypes:**

PropTypes provides a range of validators to specify the types of props:

- **PropTypes.string:** Validates that the prop is a string.
- **PropTypes.number:** Validates that the prop is a number.
- **PropTypes.boolean:** Validates that the prop is a boolean.
- **PropTypes.array:** Validates that the prop is an array.
- **PropTypes.object:** Validates that the prop is an object.
- **PropTypes.func:** Validates that the prop is a function.
- **PropTypes.element:** Validates that the prop is a React element.
- **PropTypes.node:** Validates that the prop can be a React node (string, number, element, etc.).

2. States:

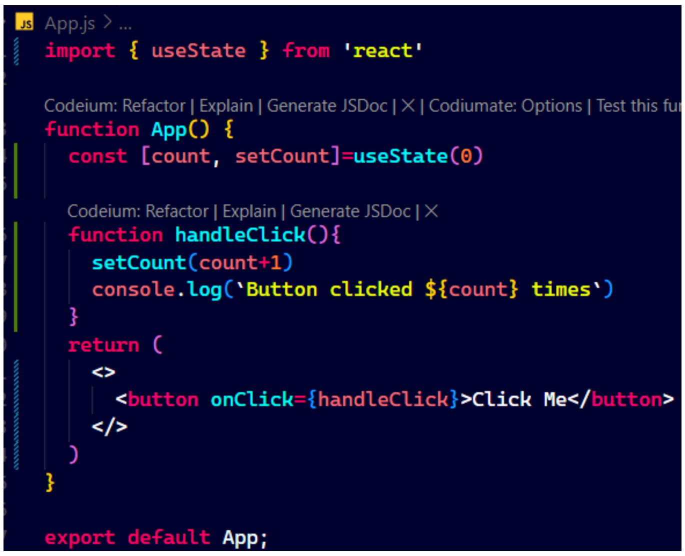
React introduced a feature called "Hooks" in React 16.8, which allows you to use state and other React features without writing a class. With the **useState** hook, you can add state to functional components.

Initializing State with useState Hook:

To use state, you import the **useState** hook from React and call it within the component function. It returns an array with two elements: the current state value and a function to update that value.

State is used to manage component-specific data that can change over time. setState is a method used to update the state of a component.

Example:



```
App.js > ...
import { useState } from 'react'

Codeium: Refactor | Explain | Generate JSDoc | X | Codiumate: Options | Test this fu
function App() {
  const [count, setCount]=useState(0)

  Codeium: Refactor | Explain | Generate JSDoc | X
  function handleClick(){
    setCount(count+1)
    console.log(`Button clicked ${count} times`)
  }
  return (
    <>
      <button onClick={handleClick}>Click Me</button>
    </>
  )
}

export default App;
```

In this example:
We import **useState** hook from React.
Inside the App function, we call **useState(0)** to declare a state variable named count initialized with the value 0.
When the button is clicked, the setCount function is called with the new count value, updating the state and triggering a re-render.

You can call useState multiple times in a single component to manage multiple independent state variables. Each call to useState creates a separate state variable and its associated setter function.

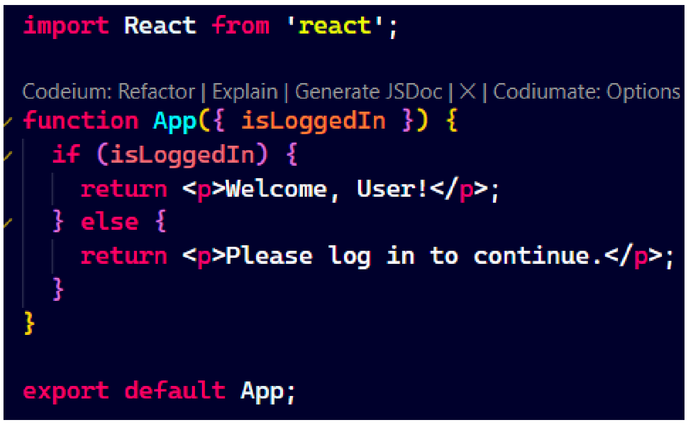
.

CONDITIONAL RENDERING

In React, there is no special syntax for writing conditions. Instead, you'll use the same techniques as you use when writing regular JavaScript code. For example, you can use an **if statement** to conditionally include JSX.

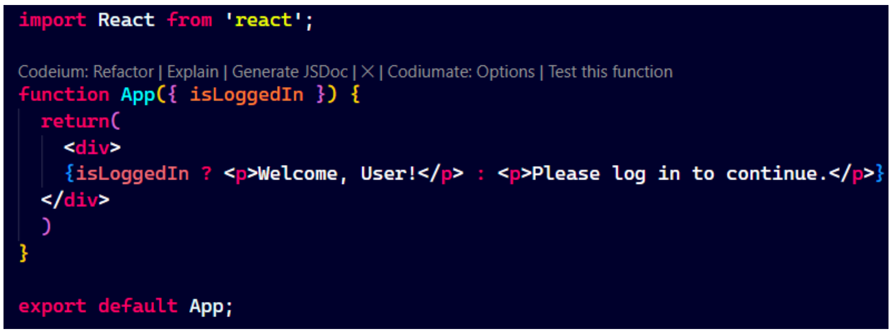
Conditional rendering in React allows you to render different content based on certain conditions. This is useful for displaying different UI elements or content based on the state of your application or the properties passed to your components.

Example 1: using if else



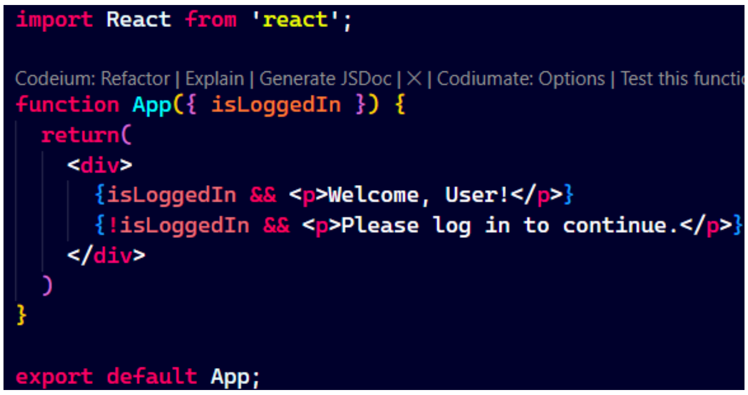
```
import React from 'react';

Codeium: Refactor | Explain | Generate JSDoc | × | Codiumate: Options
function App({ isLoggedIn }) {
  if (isLoggedIn) {
    return <p>Welcome, User!</p>;
  } else {
    return <p>Please log in to continue.</p>;
  }
}

export default App;
```

Example 2: using Ternary Operator



```
import React from 'react';

Codeium: Refactor | Explain | Generate JSDoc | × | Codiumate: Options | Test this function
function App({ isLoggedIn }) {
  return(
    <div>
    {isLoggedIn ? <p>Welcome, User!</p> : <p>Please log in to continue.</p>}
  </div>
  )
}

export default App;
```

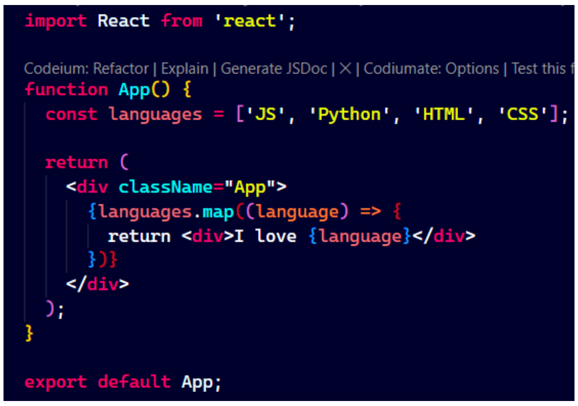
Example 3: using Logical && Operators



```
import React from 'react';

Codeium: Refactor | Explain | Generate JSDoc | × | Codiumate: Options | Test this functi
function App({ isLoggedIn }) {
  return(
    <div>
      {isLoggedIn && <p>Welcome, User!</p>}
      {!isLoggedIn && <p>Please log in to continue.</p>}
    </div>
  )
}

export default App;
```

➢ Rendering lists:

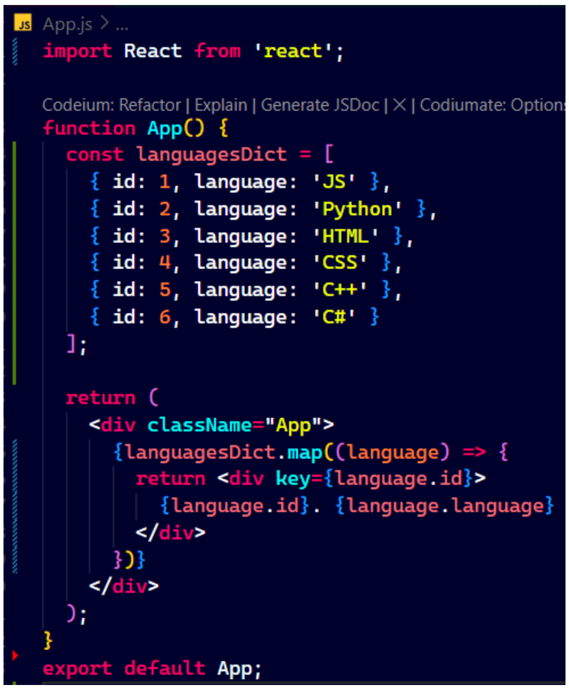
You will rely on JavaScript features like for loop and the array map() function to render lists of components.



```
import React from 'react';

Codeium: Refactor | Explain | Generate JSDoc | X | Codiumate: Options | Test this f
function App() {
  const languages = ['JS', 'Python', 'HTML', 'CSS'];

  return (
    <div className="App">
      {languages.map((language) => {
        return <div>I love {language}</div>
      })}
    </div>
  );
}

export default App;
```

➢ Keys

With keys React keep record of elements. This ensures that if an item is updated or removed, only that item will be re-rendered instead of the entire list.



```
App.js > ...
import React from 'react';

Codeium: Refactor | Explain | Generate JSDoc | X | Codiumate: Option
function App() {
  const languagesDict = [
    { id: 1, language: 'JS' },
    { id: 2, language: 'Python' },
    { id: 3, language: 'HTML' },
    { id: 4, language: 'CSS' },
    { id: 5, language: 'C++' },
    { id: 6, language: 'C#' }
  ];

  return (
    <div className="App">
      {languagesDict.map((language) => {
        return <div key={language.id}>
          {language.id}. {language.language}
        </div>
      })}
    </div>
  );
}
export default App;
```

REACT ROUTER

React Router is a popular routing library for React applications. It allows you to handle routing and navigation in your single-page applications (SPAs) by mapping URLs to different components, enabling the creation of multi-page experiences within a single HTML page.

Installations: npm install react-router-dom

Basic Usage:

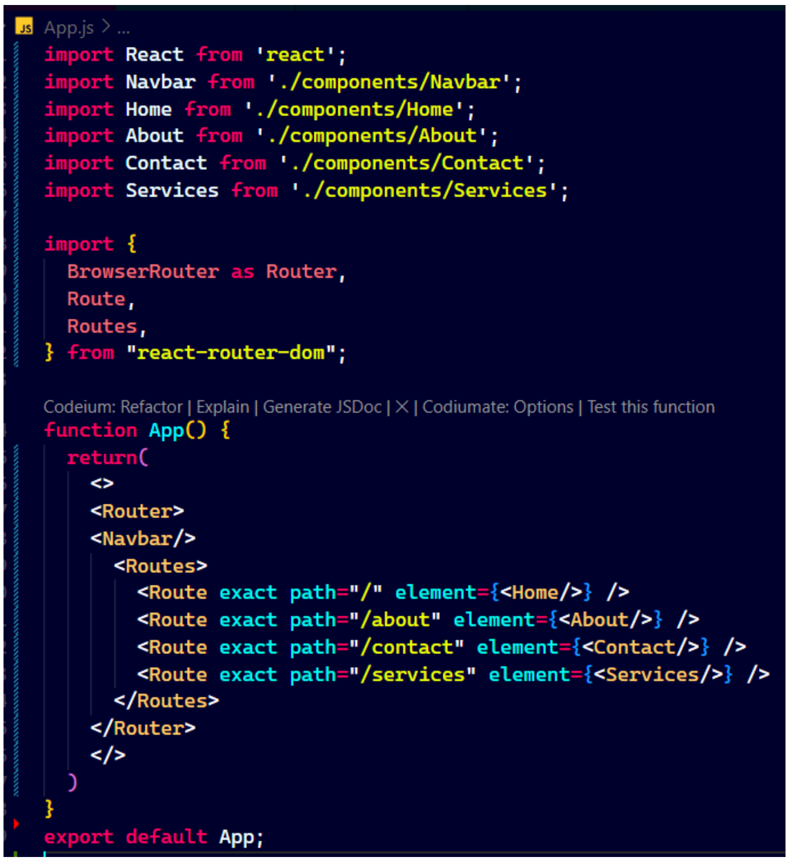
React Router provides several components to help you define your application's routes. The most commonly used ones are BrowserRouter, Route, and Link.

1. **BrowserRouter:**
   The BrowserRouter component wraps your entire application and provides the routing functionality.

2. **Route:**
   The Route component renders a specific component when the path matches the current URL.

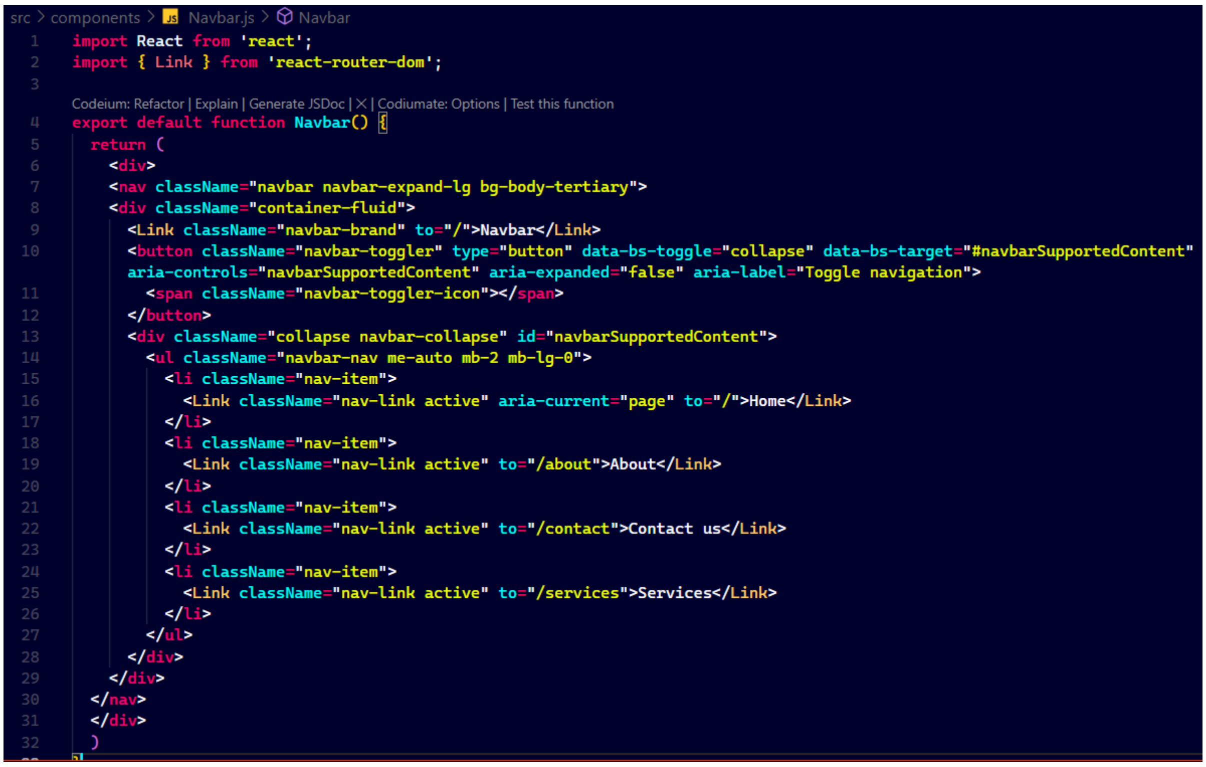
3. **Link:**
   The Link component provides navigation links in your application. It renders an anchor tag (<a>) with the appropriate href attribute.

Example:



```
import React from 'react';
import Navbar from './components/Navbar';
import Home from './components/Home';
import About from './components/About';
import Contact from './components/Contact';
import Services from './components/Services';

import {
  BrowserRouter as Router,
  Route,
  Routes,
} from "react-router-dom";

Codeium: Refactor | Explain | Generate JSDoc | X | Codiumate: Options | Test this function
function App() {
  return(
    <>
    <Router>
    <Navbar/>
      <Routes>
        <Route exact path="/" element={<Home/>} />
        <Route exact path="/about" element={<About/>} />
        <Route exact path="/contact" element={<Contact/>} />
        <Route exact path="/services" element={<Services/>} />
      </Routes>
    </Router>
    </>
  )
}
export default App;
```

Navbar.js:


src > components > Navbar.js > Navbar
Codeium: Refactor | Explain | Generate JSDoc | X | Codiumate: Options | Test this function

```
import React from 'react';
import { Link } from 'react-router-dom';

export default function Navbar() {
  return (
    <div>
    <nav className="navbar navbar-expand-lg bg-body-tertiary">
    <div className="container-fluid">
      <Link className="navbar-brand" to="/">Navbar</Link>
      <button className="navbar-toggler" type="button" data-bs-toggle="collapse" data-bs-target="#navbarSupportedContent"
      aria-controls="navbarSupportedContent" aria-expanded="false" aria-label="Toggle navigation">
        <span className="navbar-toggler-icon"></span>
      </button>
      <div className="collapse navbar-collapse" id="navbarSupportedContent">
        <ul className="navbar-nav me-auto mb-2 mb-lg-0">
          <li className="nav-item">
            <Link className="nav-link active" aria-current="page" to="/">Home</Link>
          </li>
          <li className="nav-item">
            <Link className="nav-link active" to="/about">About</Link>
          </li>
          <li className="nav-item">
            <Link className="nav-link active" to="/contact">Contact us</Link>
          </li>
          <li className="nav-item">
            <Link className="nav-link active" to="/services">Services</Link>
          </li>
        </ul>
      </div>
    </div>
  </nav>
  </div>
  )
```

React Router is a powerful tool for managing client-side routing in React applications, allowing you to create SPAs with multiple views and navigation between them without a full page reload.

➢ Events:

In React, events are actions triggered by users interacting with your application, such as clicking a button, typing in an input field, or scrolling.

Every HTML attribute in React is written in **camelCase** syntax. Event is also an attribute. Hence, written in camelCase.

Event Handling: You attach event handlers to JSX elements using camelCase naming convention, like onClick, onSubmit, onChange, etc.

Event Handlers: Event handlers are functions that are called when an event is triggered. These functions typically receive an event object as a parameter, which contains information about the event, such as the target element and the type of event.

Updating State: In many cases, when an event is triggered, you might want to update the state of your React component. You can do this by calling setState within your event handler

REACT HOOKS

Functions starting with **use** are called Hooks**. useState** is a built-in Hook provided by React. React Hooks are functions that enable functional components to use state and other React features without writing a class. They were introduced in React 16.8 as a way to simplify and enhance state management, side effects, and other features in functional components.

Hooks are more restrictive than other functions. You can only call Hooks at the top of your components (or other Hooks). If you want to use useState in a condition or a loop, extract a new component and put it there.

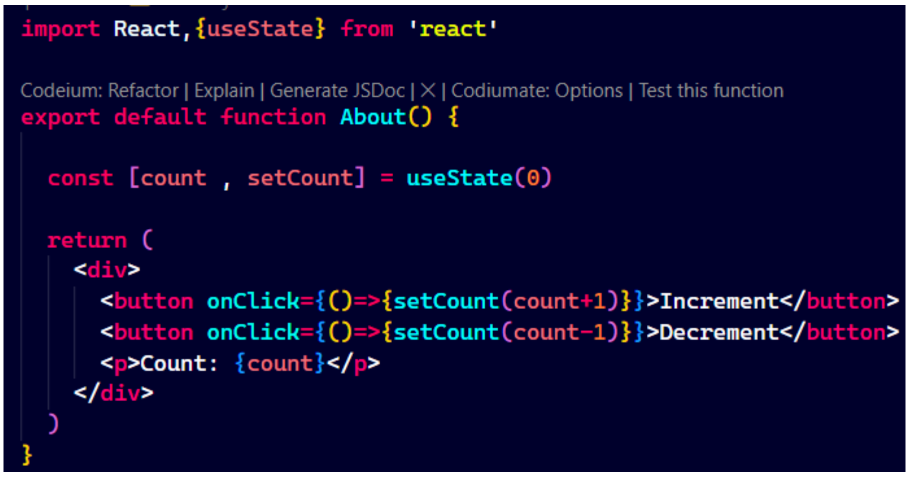
➢ **Hook Rules:**

- You must import hook first.
- You must import it from react.
- Hooks can only be called in React Function Components.
- Hooks cannot be conditional.
- Hooks cannot work in React Class Components.
- Hooks can only be called at the top level of a component, meaning it can't be called from inside a block, i.e. {}. So, can't be called inside if, loops or any block.

➢ **Types of Hooks:**

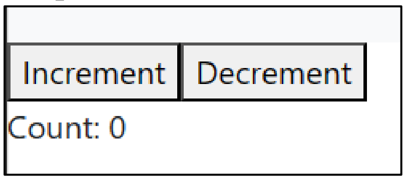
1. **useState Hooks:**

   useState is a Hook that allows functional components to manage local state. It returns a stateful value and a function to update that value.



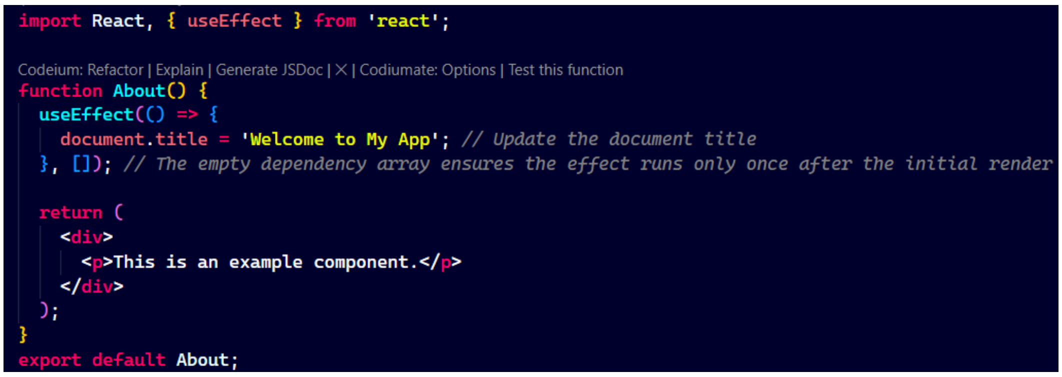
```
import React,{useState} from 'react'

Codeium: Refactor | Explain | Generate JSDoc | × | Codiumate: Options | Test this function
export default function About() {

  const [count , setCount] = useState(0)

  return (
    <div>
      <button onClick={()=>{setCount(count+1)}}>Increment</button>
      <button onClick={()=>{setCount(count-1)}}>Decrement</button>
      <p>Count: {count}</p>
    </div>
  )
}
```

Output:



```
Count: 0
```

2. **useEffect Hook:**
useEffect is a Hook that allows functional components to perform side effects in function components.

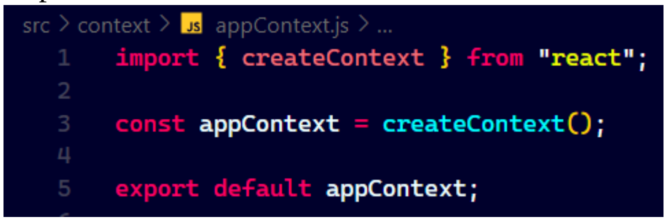


```
import React, { useEffect } from 'react';

Codeium: Refactor | Explain | Generate JSDoc | X | Codiumate: Options | Test this function
function About() {
  useEffect(() => {
    document.title = 'Welcome to My App'; // Update the document title
  }, []); // The empty dependency array ensures the effect runs only once after the initial render

  return (
    <div>
      <p>This is an example component.</p>
    </div>
  );
}
export default About;
```

3. **useContext Hook:**
React Context is a way to manage state globally.
It can be used together with the useState Hook to share state between deeply nested components more easily than with useState alone.

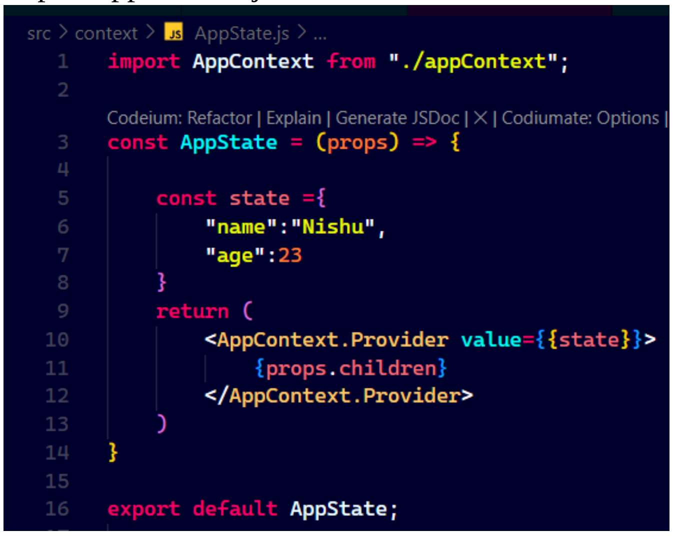
It's a hook so first we need to import it from react and initialize it.

Create a file: appContext.js → Type following code.
Import createContext:



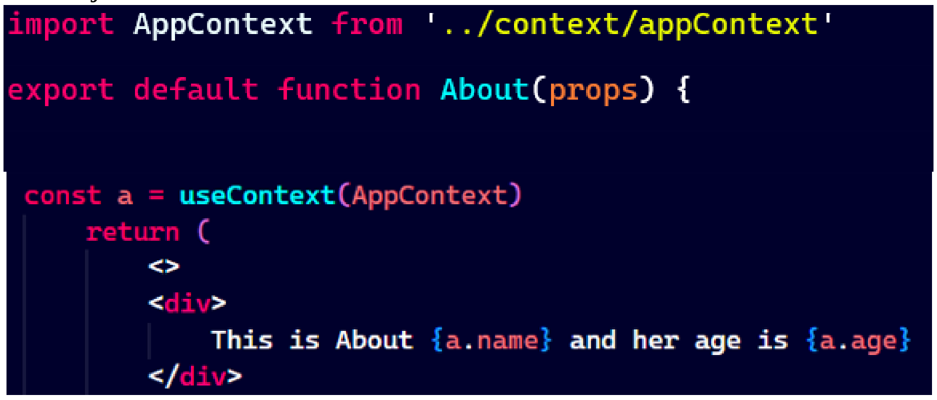
```
src > context > appContext.js > ...
import { createContext } from "react";

const appContext = createContext();

export default appContext;
```

Create another file: AppState.js
import appContext.js here



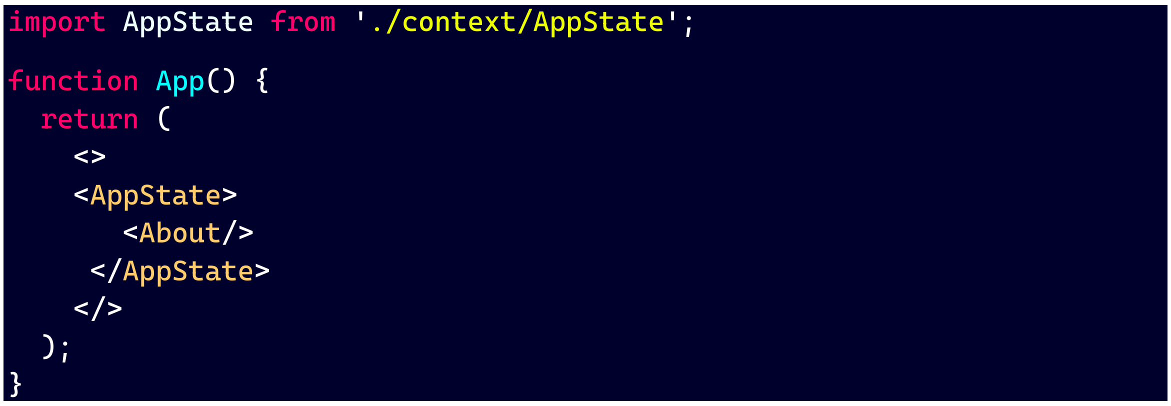
```
src > context > AppState.js > ...
import AppContext from "./appContext";

Codeium: Refactor | Explain | Generate JSDoc | X | Codiumate: Options |
const AppState = (props) => {

    const state ={
        "name":"Nishu",
        "age":23
    }
    return (
        <AppContext.Provider value={{state}}>
            {props.children}
        </AppContext.Provider>
    )
}

export default AppState;
```

Use Context Hook:
About.js:



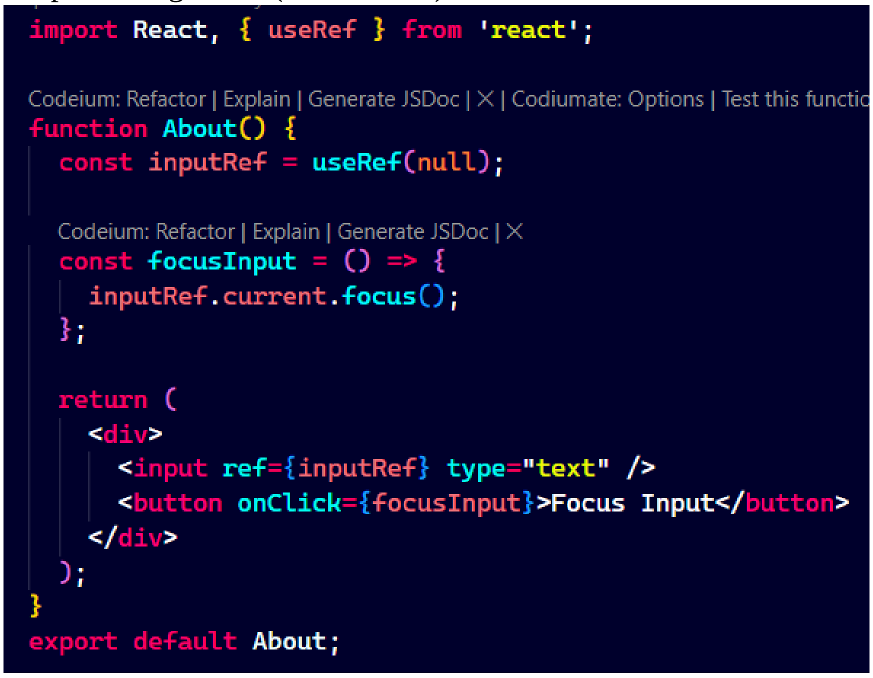
```
import AppContext from '../context/appContext'

export default function About(props) {


  const a = useContext(AppContext)
      return (
          <>
          <div>
              This is About {a.name} and her age is {a.age}
          </div>
```

App.js (main file): Wrap it with <AppState>



```
import AppState from './context/AppState';

function App() {
  return (
    <>
    <AppState>
       <About/>
     </AppState>
    </>
  );
}
```

4. **useRef Hook:**
useRef is a Hook that returns a mutable ref object whose current property is initialized to the passed argument (initial value).

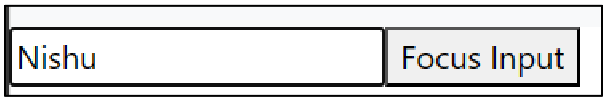


```
import React, { useRef } from 'react';

Codeium: Refactor | Explain | Generate JSDoc | X | Codiumate: Options | Test this functio
function About() {
  const inputRef = useRef(null);

  Codeium: Refactor | Explain | Generate JSDoc | X
  const focusInput = () => {
    inputRef.current.focus();
  };

  return (
    <div>
      <input ref={inputRef} type="text" />
      <button onClick={focusInput}>Focus Input</button>
    </div>
  );
}
export default About;
```

In this example:

We use the **useRef** hook to create a mutable ref object and initialize it with null.
We attach the **inputRef** to the input element using the ref attribute, allowing us to access and manipulate the input DOM node directly.
We define a **focusInput** function that uses **inputRef.current.focus()** to programmatically focus on the input element when the button is clicked.
When the "Focus Input" button is clicked, the focusInput function is called, and it sets focus on the input element using the ref.

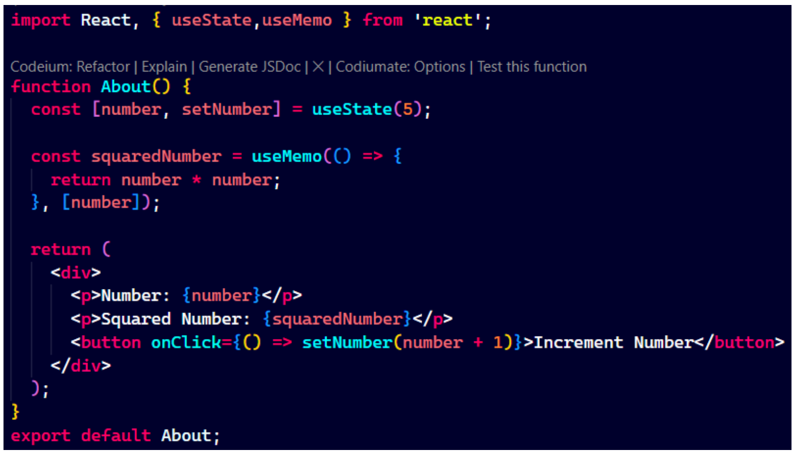
Output:



5. **useMemo Hook and useCallback Hook:**
useMemo and useCallback are Hooks that optimize performance by memoizing values and functions, respectively.

Memoization:
It is an optimization technique used in programming to speed up the execution of functions by caching the results of expensive function calls and returning the cached result when the same inputs occur again.
In the context of React, the term "memoized" refers to the process of caching the result of a function component so that the component is only re-rendered when its dependencies change.
Memoization is particularly useful when working with computationally expensive calculations, expensive component rendering, or when dealing with components that re-render frequently but have the same output given the same inputs.

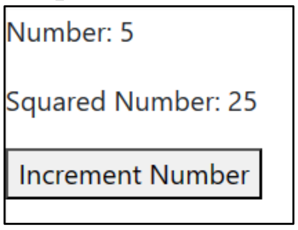
useMemo:



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

Codeium: Refactor | Explain | Generate JSDoc | X | Codiumate: Options | Test this function
function About() {
  const [number, setNumber] = useState(5);

  const squaredNumber = useMemo(() => {
    return number * number;
  }, [number]);

  return (
    <div>
      <p>Number: {number}</p>
      <p>Squared Number: {squaredNumber}</p>
      <button onClick={() => setNumber(number + 1)}>Increment Number</button>
    </div>
  );
}
export default About;
```

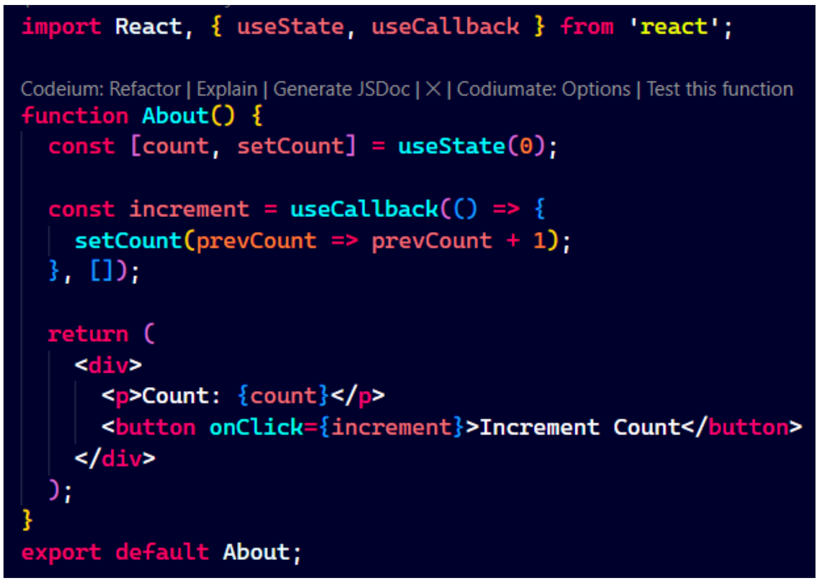
The useMemo hook takes a function as its first argument that computes the value to be memoized and an array of dependencies as its second argument. If any of the dependencies change, the memoized value will be recalculated.

Output:



```
Number: 5

Squared Number: 25
```

useCallback:



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

Codeium: Refactor | Explain | Generate JSDoc | × | Codiumate: Options | Test this function
function About() {
  const [count, setCount] = useState(0);

  const increment = useCallback(() => {
    setCount(prevCount => prevCount + 1);
  }, []);

  return (
    <div>
      <p>Count: {count}</p>
      <button onClick={increment}>Increment Count</button>
    </div>
  );
}
export default About;
```

We use the **useCallback** hook to create a memoized callback function increment.
The increment function increments the count state value by 1 using the functional form of setCount.
The empty dependency array [] ensures that the increment function is only created once and doesn't change between re-renders.
The increment function is passed as a callback to the button's onClick event to increment the count when the button is clicked.

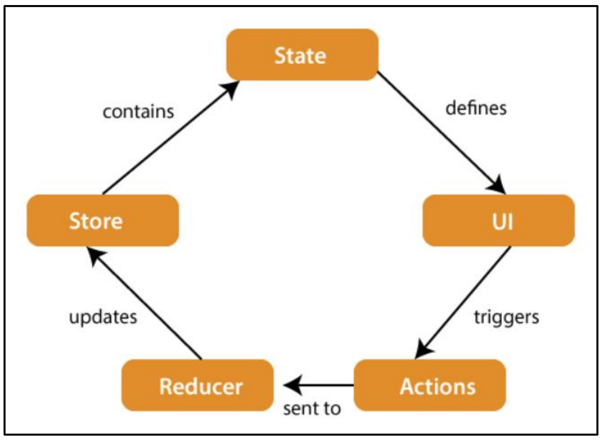
REDUX

Redux is a state managing library used in JavaScript apps. It simply manages the state of your application or in other words, it is used to manage the data of the application.

Uses: It makes easier to manage state and data. As the complexity of our application increases.

Why redux?
When a JavaScript application grows big, it becomes difficult for the user to manage its state. Redux solves this problem by managing application's state with a single global object called "Store". It makes testing very easy.
Provides consistency throughout the application.

Building Parts of Redux:



1. **Action:**
   Actions are payloads of information that send data from your application to the Redux store. They are plain JavaScript objects and must have a **type** property that indicates the type of action being performed.
   Action Creators: these are the function that creates actions.
   So, actions are the information (Objects) and action creator are functions that return these actions.

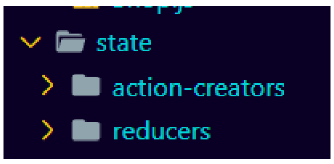
2. **Reducers:**
   Actions only tell what to do, but they don't tell how to do, so reducers are the pure functions that take the current state and action and return the new state and tell the store how to do. There can be multiple reducers. Each reducer is responsible for managing a specific slice of the application state.

3. **Store:**
   The store is the object which holds the state of the application.
   Functions associated with Store:
   - **createStore():** To create a store.
   - **dispatch(action):** To change the state.
   - **getState():** for getting current state of store.

Installation: npm i redux react-redux redux-thunk

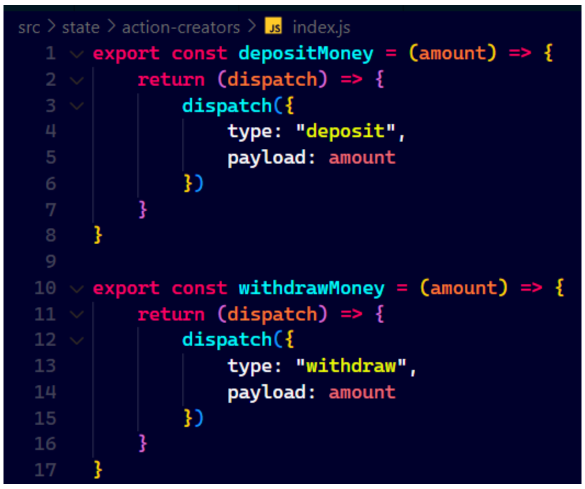
➢ To implement Redux in your code, you'll typically follow these steps:

1. **Define your application's state:** Identify the data that needs to be managed by Redux and define the initial state of your application.



```
state
  action-creators
  reducers
```

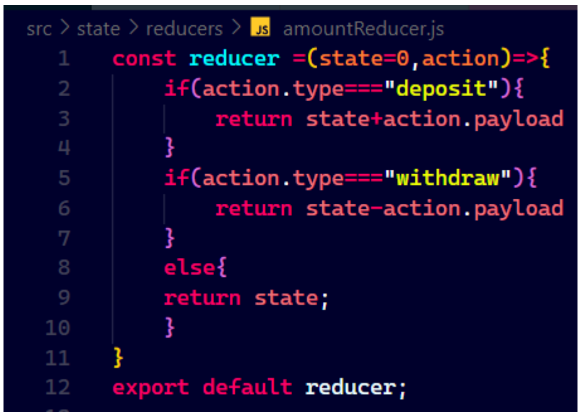
2. **Create actions:**



src > state > action-creators > index.js

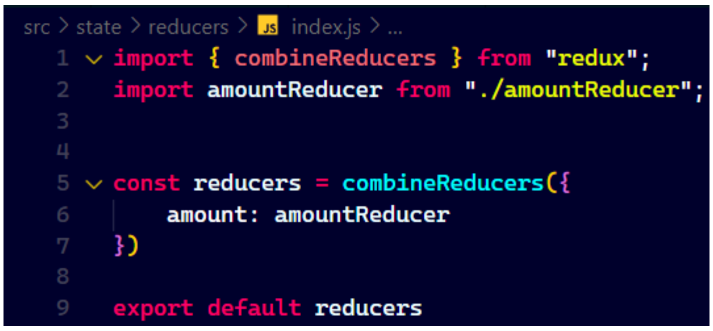
```
export const depositMoney = (amount) => {
    return (dispatch) => {
        dispatch({
            type: "deposit",
            payload: amount
        })
    }
}

export const withdrawMoney = (amount) => {
    return (dispatch) => {
        dispatch({
            type: "withdraw",
            payload: amount
        })
    }
}
```

3. **Create reducers:**



src > state > reducers > amountReducer.js

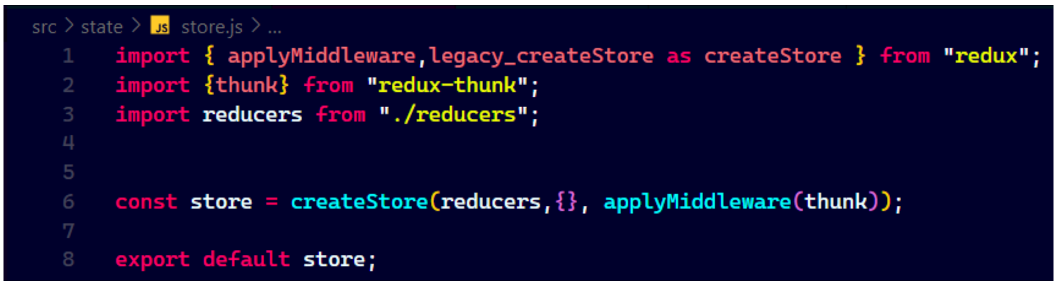
```
const reducer =(state=0,action)=>{
    if(action.type==="deposit"){
        return state+action.payload
    }
    if(action.type==="withdraw"){
        return state-action.payload
    }
    else{
    return state;
    }
}
export default reducer;
```



src > state > reducers > index.js > ...

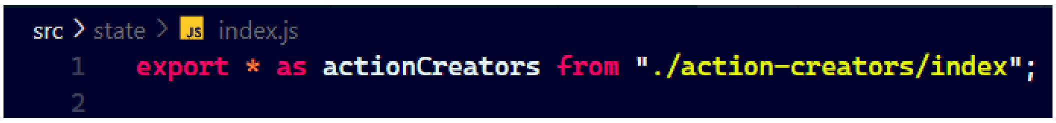
```
import { combineReducers } from "redux";
import amountReducer from "./amountReducer";


const reducers = combineReducers({
    amount: amountReducer
})

export default reducers
```

4. **Create the Redux store:** The store brings actions and reducers together. It holds the complete state tree of your application and allows you to dispatch actions to change the state.



src > state > store.js > ...

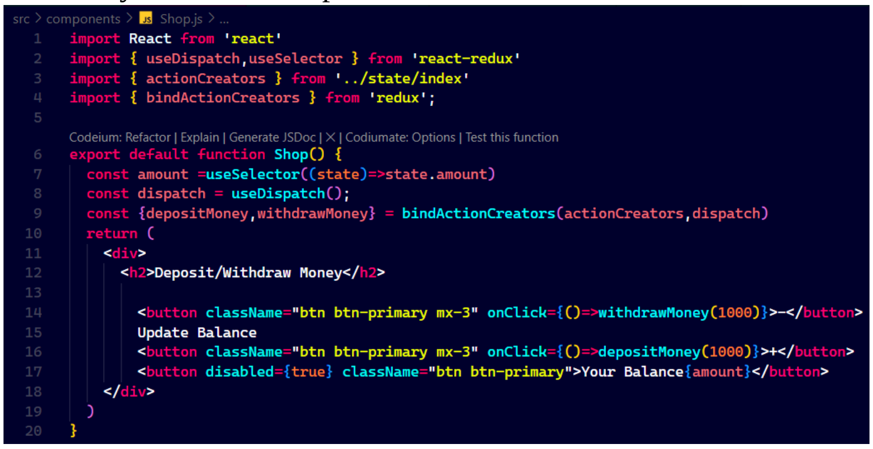
```javascript
import { applyMiddleware,legacy_createStore as createStore } from "redux";
import {thunk} from "redux-thunk";
import reducers from "./reducers";


const store = createStore(reducers,{}, applyMiddleware(thunk));

export default store;
```



src > state > index.js

```javascript
export * as actionCreators from "./action-creators/index";
```

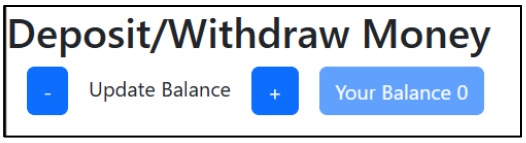
5. **Connect Redux to your components:** Use the connect function from react-redux to connect your React components to the Redux store and access the state and actions.



src > components > Shop.js > ...

```javascript
import React from 'react'
import { useDispatch,useSelector } from 'react-redux'
import { actionCreators } from '../state/index'
import { bindActionCreators } from 'redux';

Codeium: Refactor | Explain | Generate JSDoc | × | Codiumate: Options | Test this function
export default function Shop() {
  const amount =useSelector((state)=>state.amount)
  const dispatch = useDispatch();
  const {depositMoney,withdrawMoney} = bindActionCreators(actionCreators,dispatch)
  return (
    <div>
      <h2>Deposit/Withdraw Money</h2>

        <button className="btn btn-primary mx-3" onClick={()=>withdrawMoney(1000)}>-</button>
        Update Balance
        <button className="btn btn-primary mx-3" onClick={()=>depositMoney(1000)}>+</button>
        <button disabled={true} className="btn btn-primary">Your Balance{amount}</button>
    </div>
  )
}
```

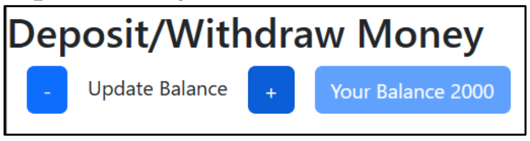
Output:



Deposit/Withdraw Money

- Update Balance + Your Balance 0

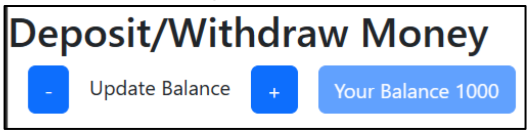
depositMoney Function:



Deposit/Withdraw Money

- Update Balance + Your Balance 2000

withdrawMoney Function:



Deposit/Withdraw Money

- Update Balance + Your Balance 1000