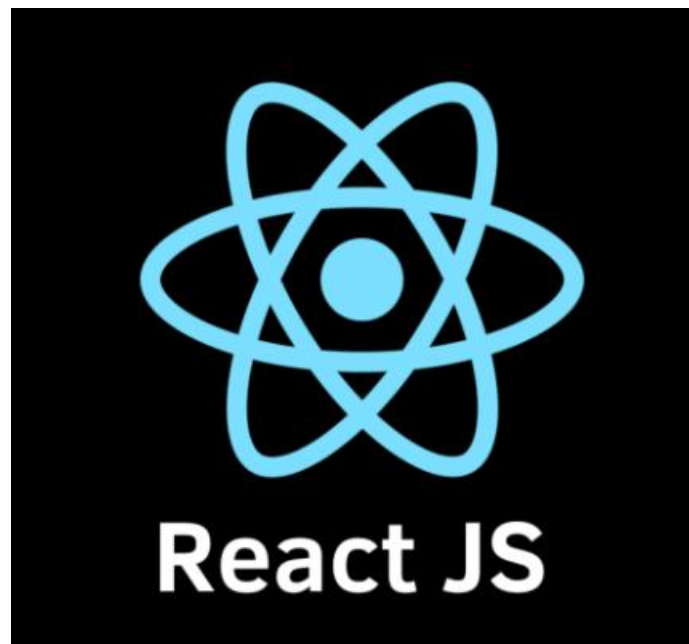




Project- 13

React.JS



Submitted BY:

Manaswi M. Patil



INDEX

SR.NO.	TOPICS	PG.NO.
1.	Introduction	3
2.	Installation	4
3.	JSX	8
4	ES6	10
5.	React DOM	13
6.	Components	14
7.	Props and States	16
8.	Conditional Rendering	18
9.	React Router	19
10.	React Hooks	21
11.	Redux	30



1.Introduction

ReactJS is a **JavaScript** library used to build User Interfaces(UI).ReactJS, commonly referred to as React, is an open-source JavaScript library developed and maintained by Facebook. React was designed to make it easier to develop UI components that can efficiently update and render in response to data changes.

Here are some key aspects of ReactJS:

1. **Component-Based Architecture:** React follows a component-based architecture, where the UI is broken down into independent, reusable components. Each component can have its own state, which allows for efficient management of data and UI updates.
2. **Virtual DOM (Document Object Model):** React uses a virtual DOM to optimize the rendering process. Instead of directly manipulating the browser's DOM, React creates a lightweight, in-memory representation of the DOM. When there are changes in the data, React calculates the most efficient way to update the virtual DOM and then applies those changes to the actual DOM.
3. **JSX (JavaScript XML):** React uses JSX, a syntax extension for JavaScript that allows developers to write HTML-like code within their JavaScript files. JSX makes it more intuitive to describe the structure of UI components and their relationships.
4. **Unidirectional Data Flow:** React follows a unidirectional data flow, meaning that data flows in a single direction—from parent components to child components. This helps in maintaining a clear and predictable data flow, making it easier to understand and debug code.
5. **Reusability and Composability:** React promotes the creation of reusable components, making it easier to build and maintain large-scale applications. Components can be composed together to create complex user interfaces.
6. **React Router:** React Router is a library commonly used with React for implementing client-side routing in single-page applications. It allows developers to define different "routes" in the application, enabling navigation between views or pages without a full-page reload.

React is often used in combination with other tools and libraries, such as Redux for state management, and it has a vibrant community of developers contributing to its ecosystem.



2. Installation

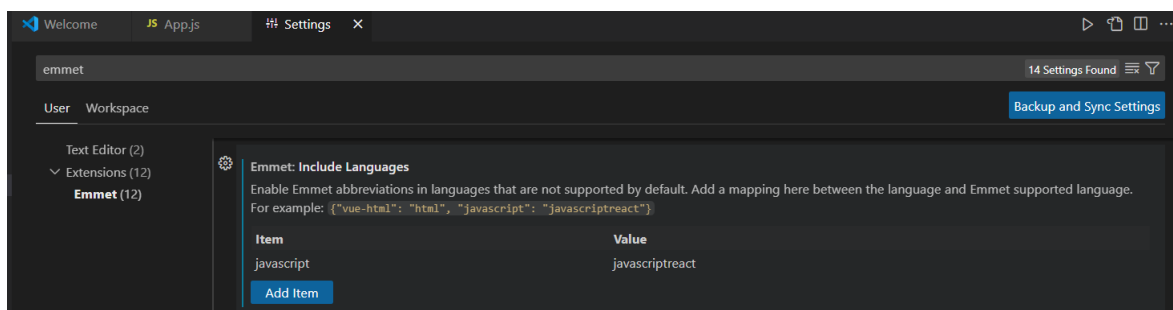
To Create react app

- Need to install Node.js → download from this link → You can download it from → <https://nodejs.org/en/> .
- Need code editor → vs code.
- **Install vs code extensions like :**

1. **Thunder Client:** Thunder Client is an HTTP client extension for Visual Studio Code. It allows you to send HTTP requests and view responses directly from within your code editor. When working with React applications, you might need to interact with APIs, and Thunder Client makes it easy to test and debug API endpoints without leaving your coding environment.
2. **ES7 React/Redux/GraphQL/React-Native snippets:** This extension provides shortcuts, also known as snippets, for common React, Redux, GraphQL, and React Native code patterns. When building React applications, you often find yourself writing repetitive boilerplate code. This extension helps you speed up development by providing shortcuts for common tasks like creating components, defining Redux actions and reducers, and more.
3. **Live Server:** Live Server is a simple development server for static and dynamic websites. It automatically reloads the web page whenever you save changes to your HTML, CSS, or JavaScript files. When developing React applications, Live Server can be used to quickly preview your changes in the browser without manually refreshing the page after each code modification.
4. **Prettier:** Prettier is a code formatter that automatically formats your code according to predefined rules. It helps maintain consistency in your codebase by enforcing a consistent code style across your project. When working with React, Prettier ensures that your JSX syntax is formatted correctly, making your code more readable and maintainable.
5. **Bracket Pair Colorizer:** This extension colorizes matching brackets in your code to make it easier to identify the scope of code blocks. When writing React components with JSX syntax, it's common to have nested elements and multiple levels of indentation. Bracket Pair Colorizer visually highlights matching pairs of brackets (parentheses, curly braces, square brackets) to help you understand the structure of your code more easily.



6. **Auto Rename Tag:** Auto Rename Tag is a handy extension that automatically renames closing HTML tags when you rename opening tags and vice versa. When working with JSX in React, you often need to rename components or HTML elements. Auto Rename Tag simplifies this process by automatically updating corresponding closing tags whenever you rename an opening tag, saving you time and reducing the risk of errors.
7. **Emmet:** Emmet is a powerful toolkit for web developers that significantly speeds up HTML and CSS workflow. It allows you to write HTML and CSS code using abbreviations and shortcuts, which are then expanded into full HTML and CSS code snippets. Emmet is incredibly useful when working with JSX in ReactJS . because its dealing with nested components and complex structures.
8. Go to setting and type emmete→ emmete include language→add→ok.



9. Babel:

Babel is a JavaScript compiler that allows developers to use next-generation JavaScript syntax (ES6/ES7 and beyond) in current environments, including older browsers and environments that may not yet support the latest JavaScript features. In ReactJS development, Babel is often used as part of the build process . JSX and ES6/ES7 code into plain JavaScript that browsers can understand. Babel allows developers to take advantage of modern JavaScript features and JSX syntax without worrying about browser compatibility issues, enabling them to write cleaner and more expressive code in React applications.

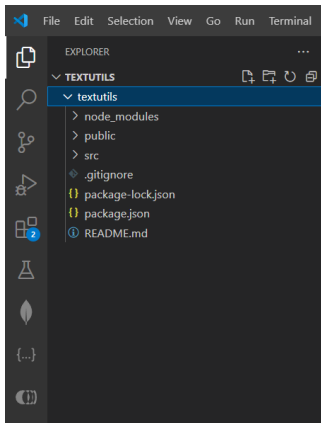
10.Strict mode in react:

In React, "strict mode" is a development mode feature that helps catch common mistakes and potential issues in your application code. When you enable strict mode, React provides additional runtime checks and warnings to help you identify and fix problems early in the development process. It is primarily meant for development environments and is not intended for production use.

To create React App:



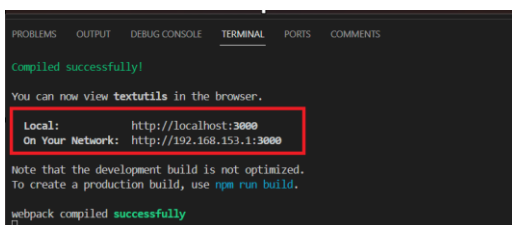
- Open terminal → `mkdir textutils` → `cd textutils`
- `npx create-react-app textutils`
- `npx create-react-app react-project-ai`
- Open vs code editor → open your folder which you created recently.
- Once the app is created, navigate into the project directory → `cd textutils` in project terminal
- Start the development server by running the following command → `npm start`
- You will get dependencies folder which already created .



Public folder: The **public** folder contains static assets like HTML files, images, and other resources that don't need to go through the build process. When you build your React application, these files are copied directly into the build output directory.

src folder: The **src** folder contains the source code of your React application. This is where you'll write your JavaScript/JSX code, including components, styles, and other application logic.

- After clicking on below link you will reached your react app to view updates which made in project window.



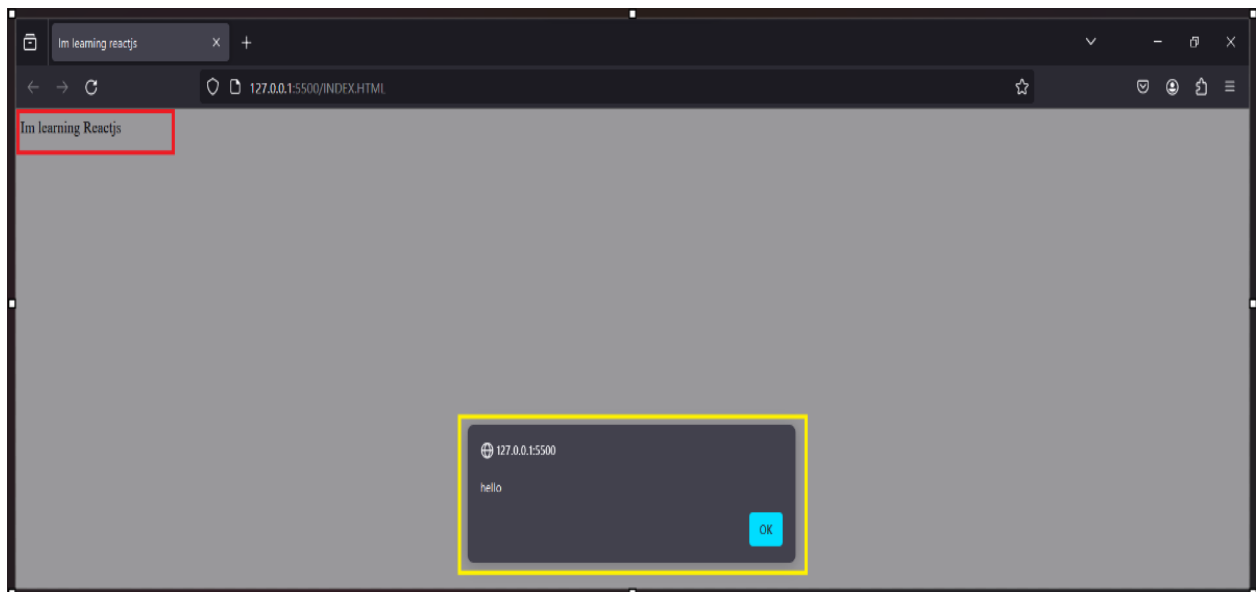
- Open with react app → <http://192.168.153.1:3000> .
- Open with browser → <http://localhost:3000> .

Simple demonstration with React:



```
EXPLORER
JS REFRESHER
INDEX.HTML
INDEX.HTML X
1 <!DOCTYPE html>
2 <html lang="en">
3 <head>
4   <meta charset="UTF-8">
5   <meta name="viewport" content="width=device-width, initial-scale=1.0">
6   <title>Im learning reactjs</title>
7   <script>
8     let a=34;
9     console.log(a)
10    document.addEventListener("click",function click(){
11      console.log("clicked")
12      alert("hello");
13    });
14  </script>
15 </head>
16 <body>
17   im learning reactjs
18 </body>
19 </html>
```

Just click on output window → click on displaying text → will get “hello” output.





3.React JSX

JSX (JavaScript XML) is a syntax extension for JavaScript often used with React. Provides a way to easily write HTML in react. It provides a more concise and readable way to write React components compared to using plain JavaScript to create and manipulate the DOM. JSX resembles XML or HTML syntax, making it easier to visualize the structure of the user interface.

1. **Syntax:** JSX allows you to write XML/HTML-like code within JavaScript files. It looks similar to HTML but is embedded directly within JavaScript.
2. **Embedding Expressions:** You can embed JavaScript expressions within JSX using curly braces `{}`. This allows you to dynamically include values or expressions within the JSX.
3. **Attributes and Props:** JSX uses HTML-like attributes to define properties (props) for React elements. Props are used to pass data from a parent component to a child component.
4. **JSX Elements:** JSX elements can represent HTML tags, React components, or fragments. JSX expressions are often assigned to variables or used directly within the `ReactDOM.render()` method.
5. **JavaScript Expressions:** JSX allows the use of any valid JavaScript expression within curly braces `{}`. This includes variables, calculations, and function calls.
6. **Children Elements:** JSX can have nested elements, and child elements are specified within the opening and closing tags of the parent element.
7. **Class vs. className:** JSX uses `className` instead of `class` to define CSS classes. This is because `class` is a reserved keyword in JavaScript.
8. **Self-Closing Tags:** Similar to HTML, self-closing tags can be used in JSX for elements without any content e.g. `
tag`.
9. **Comments:** JSX supports JavaScript-style comments within curly braces.
10. **Javascript wrapper function:** `<> </>`

JSX is a syntax extension that simplifies the creation and manipulation of React elements. It provides a more declarative and expressive way to describe the structure of a user interface in JavaScript code.

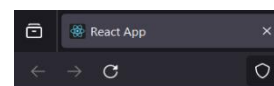


11. **Coding in JSX:** Earlier we had to make an HTML element or append it into existing ones with methods like createElement() / appendChild()
12. If get error in closing tag then add '/' to close that tag → errors will removed.

Working on React with JSX:

```
# App.css JS App.js ×
src > JS App.js > App
1 import './App.css'; // Import CSS file
2
Codeium: Refactor | Explain | Generate JSDoc | ×
3 function App() {
4   const myimage = "https://images.freeimages.com/images/large-previews/48d/marguerite-1372118.jpg";
5   return (
6     <>
7     <h1 className='heading'>Good Morning!</h1>
8     <img className='image' alt='Logo' src={myimage} />
9     </>
10  );
11 }
12
13 export default App;
14
```

```
# App.css × JS App.js
src > # App.css > .image
1 .image{
2   border: 2px solid black;
3   border-radius: 50%;
4   width: 200px;
5   height: 200px;
6 }
```



Good Morning!





4.ES6

ES6 stands for ECMAScript ,which is the sixth edition of the ECMAScript standard. ECMAScript is the scripting language specification upon which JavaScript is based. ES6 introduced several new features and enhancements to the language, making JavaScript more powerful, expressive, and developer-friendly. It was a major update to the language and brought many improvements over the previous version (ES5).

Some key features introduced in ES6 include:

1. **Arrow Functions:** A concise syntax for writing function expressions, making function definitions more compact and expressive.

```
// ES5 function
function add(x, y) {
  return x + y;
}

// ES6 arrow function
const add = (x, y) => x + y;
```

2. **Template Literals:** A more flexible way to work with strings, allowing for multi-line strings and variable interpolation.

```
// ES5
var message = "Hello, " + name + "!";

// ES6 template literal
const message = `Hello, ${name}!`;
```

3. **Let and Const Declarations:** **let** and **const** were introduced for variable declarations, providing block-scoping for **let** and constant values for **const**.

Let is a modern javascript keyword.

```
// ES5
var x = 10;

// ES6
let x = 10;
const PI = 3.14;
```

4. **Destructuring Assignment:** Allows for easily extracting values from arrays or objects and assigning them to variables.

Old way:



```
const person = { name: "John", age: 30 };
const name = person.name;
const age = person.age;
```

New way:

```
const person = { name: "John", age: 30 };
const { name, age } = person;

console.log(name, age); // Output: John 30
```

5. **Ternary operator:** The ternary operator is a concise way to write conditional statements in a single line.

```
// Old way using if-else
const x = 10;
let result;

if (x > 5) {
  result = "Greater than 5";
} else {
  result = "5 or less";
}

// New way using the ternary operator
const x = 10;
const result = x > 5 ? "Greater than 5" : "5 or less";

console.log(result); // Output: Greater than 5
```

6. **Spread Operator:** The spread operator (...) is used to expand elements of an array or properties of an object.

```
// Old way concatenating arrays
const arr1 = [1, 2, 3];
const arr2 = [4, 5, 6];
const combinedArray = arr1.concat(arr2);

// New way using spread operator
const arr1 = [1, 2, 3];
const arr2 = [4, 5, 6];
const combinedArray = [...arr1, ...arr2];

console.log(combinedArray); // Output: [1, 2, 3, 4, 5, 6]
```

7. **Classes:** A more convenient and structured way to create constructor functions and work with object-oriented programming concepts.



8. **Modules:** A standardized way to organize and import/export code, promoting better code organization and reusability.

```
// ES6 modules
// module1.js
const myModule = {};

// module2.js
import myModule from './module1';
```

9. The **.map()** method is a higher-order function in JavaScript used to iterate over an array and create a new array by applying a provided function to each element in the original array. It does not mutate the original array.

```
// Old way using a for loop
const numbers = [1, 2, 3, 4, 5];
const squaredNumbers = [];

for (let i = 0; i < numbers.length; i++) {
  squaredNumbers.push(numbers[i] * numbers[i]);
}

// New way using .map()
const numbers = [1, 2, 3, 4, 5];
const squaredNumbers = numbers.map((number) => number * number);

console.log(squaredNumbers); // Output: [1, 4, 9, 16, 25]
```

These are just a few examples of the features introduced in ES6. The subsequent ECMAScript editions (such as ES7, ES8, etc.) .



5.React DOM

ReactDOM is a package in React that provides DOM-specific methods that can be used at the top level of your web app. It's the glue between React and the actual DOM, enabling React components to be rendered into the browser DOM.

➤ `npm install react react-dom`

Here are some commonly used methods provided by **ReactDOM**:

1. **render()**: This is the most commonly used method in **ReactDOM**. It renders a React element into the DOM in the supplied container and returns a reference to the root `ReactComponent` instance.
2. **hydrate()**: This is similar to **render()**, but it's used for rehydrating server-rendered HTML into a React application. It's typically used in server-side rendering (SSR) setups.
3. **unmountComponentAtNode()**: This method removes a React component from the DOM and cleans up its event handlers and state.
4. **createPortal()**: This method allows rendering a React component into a DOM node that exists outside the DOM hierarchy of the parent component.
5. **findDOMNode()**: This method returns the underlying DOM node of a React component instance. However, its usage is discouraged in newer versions of React, and the **ref** API should be used instead.

These methods are used to interact with the DOM in a React application.

React Render HTML

- **ReactDOM.render()** function is used to render HTML to the web page. This function is part of the ReactDOM library, which is specifically designed for rendering React components into the DOM (Document Object Model).
- This function takes two arguments, HTML content which you want to show on page and HTML element where you want to put the HTML content(first argument).
- The **ReactDOM.render()** function is then used to render the **MyComponent** into the HTML document. The first argument passed to **ReactDOM.render()** is the React component, and the second argument is the target HTML element where the component should be rendered.



6.React Components

React components are the building blocks of a React application. They are reusable, self-contained pieces of code that represent parts of a user interface. React follows a component-based architecture, where the entire UI is broken down into smaller, modular components. Each component can have its own state, props, and lifecycle methods, making it easier to manage and update the user interface.

There are two main types of React components:

1. Functional Components/ Stateless Components :

- Stateless components, also known as functional components, are JavaScript functions that return JSX.
 - These components do not have their own state and do not manage any data internally.
 - Stateless components are simple and lightweight as they only receive data via props and render it.
 - They are primarily used for representing UI elements, presentational components, or reusable component fragments.
 - Stateless components are easy to test, understand, and reason about due to their simplicity.
- Create any component → Type **rfc** in editor window for snipes.

Functional components are simpler and more concise. They are stateless and don't have their own internal state or lifecycle methods. Functional components are primarily used for presenting UI elements.

2. Class Components/Stateful Components:

- Stateful components, also known as class components, are JavaScript classes that extend `React.Component` or `React.PureComponent`.
- These components have the ability to hold and manage local state data.
- Stateful components can change their internal state using the `setState()` method provided by React.



- They typically contain business logic, handle user interactions, and manage data fetching and manipulation.
 - Stateful components are used when a component needs to maintain and update its own state over time, such as handling form input, managing UI state, or fetching data from an API.
- Create components folder in src.
 - Create a file as Component.js.
 - React component names must always start with a capital letter.
 - Write the component code
 - Type rcc for snippets.
 - Class components are more feature-rich and can manage their own state, handle lifecycle methods, and contain additional logic. With the introduction of React Hooks, functional components can now also manage state and lifecycle methods, reducing the need for class components.

Rendering a Component:

We made a component, now we want to render/use it. Syntax for using a component is:

`<Component/>`

- **Note:** File name must start with uppercase letter.
- File name and function name shouldn't same. It should be a different.



7.Props and States

- **Props:** When we want to use multiple content at once then must use props means property attribute.
- React.js has a feature like 'hot-reloading', whenever you make changes in the code it will automatically update the output.

We can import the same component in different files and use it, but maybe in different files some changes in the component are needed.

React.js, props (short for properties) serve as a mechanism for passing data from parent components to child components.

1. Passing Props from Parent to Child:

- In the parent component, you define the child component and pass data to it as props.
- Props are passed as attributes to the child component when it's being used in the parent component's JSX.

2. Receiving Props in Child Component:

- Inside the child component, you access the props passed from the parent via the **props** object.

3. Rendering Props:

- You can render the values of props directly within the child component's JSX using curly braces {}.

4. Dynamic Props:

- Props can be dynamic, meaning their values can change based on state or other factors in the parent component. Whenever the parent component's state or props change, React automatically re-renders the child component with the updated props.

5. Default Props:

- You can define default values for props in case they're not provided by the parent component. This ensures that your component doesn't break due to missing props.

By passing data through props, React enables a unidirectional data flow from parent to child, which helps in creating predictable and maintainable component-based architectures.

PropTypes: PropTypes is a package that allows you to declare the intended types of the props passed to a component. It's a helpful tool for both development and documentation purposes as it helps catch bugs early and provides clear expectations for component usage.

1. Primitive Types:

- **PropTypes.string:** A string.
- **PropTypes.number:** A number.



- **PropTypes.boolean:** A boolean.
 - **PropTypes.symbol:** A symbol.
 - **PropTypes.func:** A function.
 - **PropTypes.object:** An object.
2. **Composite Types:**
- **PropTypes.array:** An array.
 - **PropTypes.node:** A React node (string, number, element, array, fragment, etc.).
 - **PropTypes.element:** A React element.
 - **PropTypes.instanceOf(ClassName):** An instance of a specific class.
 - **PropTypes.oneOf([val1, val2]):** One of the specified values.
 - **PropTypes.oneOfType([type1, type2]):** One of the specified types.
 - **PropTypes.arrayOf(type):** An array of a specific type.
 - **PropTypes.objectOf(type):** An object with values of a specific type.
 - **PropTypes.shape({ key: PropTypes.type }):** An object with specific keys and types.
3. **Special Types:**
- **PropTypes.any:** Any data type.
 - **PropTypes.func.isRequired:** A required function.
 - **PropTypes.oneOfType([type1, type2]).isRequired:** A required prop with one of the specified types.
 - **PropTypes.shape({ key: PropTypes.type }).isRequired:** A required object with specific keys and types.

State:

- **State** is a built-in feature of React components used for managing internal component data.
- Unlike props, state is mutable and can be modified using the **setState()** method provided by React. State is local to a component and cannot be accessed or modified by other components.
- State is typically used for managing dynamic data that may change over time, such as user input, UI state, or data fetched from an API.
- When state changes, React automatically re-renders the component to reflect the updated state. State should be initialized in the constructor of a class component using **this.state** or using the **useState()** hook in functional components.
- Updating state asynchronously may lead to unexpected behavior, so it's important to use the functional form of **setState()** when depending on the current state.



8.React Conditional Rendering

Conditional rendering in React refers to the ability to render different elements or components based on certain conditions. This is a fundamental aspect of building dynamic user interfaces where what the user sees can change based on various factors like user interaction, data availability, or authentication status.

Techniques for Conditional Rendering:

1. **Using if-else statements:** You can use regular JavaScript conditional statements inside your JSX to conditionally render different components.
2. **Using ternary operator:** The ternary operator provides a concise way to conditionally render elements.
3. **Using logical && operator:** This is useful for rendering a component only if a condition is true.
4. **Using logical || operator:** This is useful for rendering a default component when a condition is false.

Conditional Rendering with State:

Conditional rendering often involves managing state to keep track of conditions that affect what gets rendered. State can be managed using React's built-in **useState** hook or class-based component state.

React Lists

Lists are a fundamental part of many UIs, displaying data in a structured manner. In React, rendering lists involves mapping over an array of data to generate a collection of React elements.

Rendering Lists:

The `map()` function is commonly used to iterate over arrays and generate a list of elements based on the array items.

- Items is an array containing the data to be rendered.
- `map()` iterates over each item in the array.
- Each item is transformed into a `` element with its content.

Key Prop: When rendering lists in React, it's important to assign a unique key prop to each item. This helps React identify which items have changed, been added, or removed efficiently.

Dynamic Lists: Lists in React can be dynamic, meaning they can change based on user interaction, data fetching, or other events. By updating the underlying data and re-rendering the component, React efficiently handles dynamic lists.



9.React Router

React Router is a popular routing library for React applications. It enables navigation and routing within single-page applications (SPAs) by mapping URLs to React components.

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.

- Installation: First, you install React Router in your project using npm or yarn.
- `npm install react-router-dom`

Key Concepts:

- **BrowserRouter:** Provides client-side routing using HTML5 history API.
- **Route:** Renders a UI component based on the URL.
- **Link/NavLink:** Provides navigation links to different routes.
- **Switch:** Renders the first matching route exclusively.
- **Route Parameters:** Allows dynamic route matching.
- **Nested Routes:** Enables nested route structures.
- ❖ Importing Components: You import the necessary components from React Router, such as BrowserRouter, Route, Switch, Link, or NavLink, depending on your requirements.
 - Defining Routes: You define your application's routes using the Route component. Each Route component specifies a URL path and the component to render when that path matches the current URL.
 - Navigation Links: You use the Link or NavLink component to create navigation links within your application. These components generate anchor tags (`<a>`) with the appropriate URLs, allowing users to navigate between different routes.
 - Rendering Routes: React Router dynamically renders the appropriate component based on the current URL and the defined routes. It uses a matching algorithm to determine which route to render, considering factors like exact matching, partial matching, and route hierarchy.
 - Handling Dynamic Routes: React Router supports dynamic routes with URL parameters. You can access these parameters within your components and use them to fetch data or render dynamic content.
 - Navigating Programmatically: You can navigate between routes programmatically using methods provided by React Router, such as `history.push()` or `history.replace()`. This allows you to navigate based on user actions or application logic.



- **Handling NotFound Routes:** You can define a special route for handling 404 errors or routes that do not match any of the defined routes. This ensures a graceful fallback for unexpected URLs.
- **Additional Features:** React Router provides additional features like nested routing, route guards, query parameters, code splitting, server-side rendering, and more, depending on your project's requirements.
- **Testing and Debugging:** You can test and debug your application's routing behavior using tools like React Developer Tools or browser developer tools. These tools allow you to inspect the current route, view route parameters, and debug navigation issues.

React Events

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

Arguments in events: We can't pass arguments just like that, it will give syntax error. First, we need to put the whole function in arrow function.

React Event Object: Event handler can be provided to the function like this.

Using inline event handlers in JSX: You can define event handlers directly within JSX using arrow functions or regular function declarations. This approach is convenient for simple event handling.

Using class methods: In class components, you can define event handlers as class methods. This approach is useful when you need to access component state or props within the event handler.

Passing parameters to event handlers: If you need to pass additional data to an event handler, you can use arrow functions or the **bind()** method.



10. React Hooks

React Hooks are functions that enable functional components to use React features. They allow you to use state and other React features without writing a class. Some commonly used React Hooks include `useState`, `useEffect`, `useContext`, `useRef`, `useReducer`, `useCallback`, and `useMemo`.

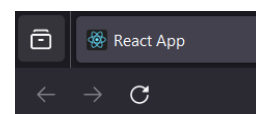
What is a Hook?

A Hook in React is a special function that allows you to use React features in functional components. They let you "hook into" React state and lifecycle features from functional components. Hooks allow for code reuse, better organization, and improved readability by separating concerns and making components more modular.

React `useState` Hook:

- `useState` is a Hook that allows functional components to add state variables. It returns a stateful value and a function to update that value, providing a way to manage state in functional components. It's commonly used to store and update component state.
- React Hooks provide a way to use React features in functional components, promoting code reuse and better organization.

```
JS App.js x
src > JS App.js > ExampleComponent
1  import React, { useState } from 'react';
2
3  Codeium: Refactor | Explain | Generate JSDoc | X
4  const ExampleComponent = () => {
5    // Define a state variable named 'count' and a function to update it
6    const [count, setCount] = useState(0);
7
8    // Event handler to increment the count
9    Codeium: Refactor | Explain | X
10   const incrementCount = () => {
11     setCount(count + 1); // Update the 'count' state
12   };
13
14   return (
15     <div>
16       <h2>Counter</h2>
17       <button onClick={incrementCount}>Increment</button>
18       <button onClick={incrementCount}>Decrement</button>
19       <p>Count: {count}</p>
20     </div>
21   );
22 };
23 export default ExampleComponent;
24
```



Counter

Increment Decrement

Count: 0

React `useContext`:

`useContext` is a Hook that allows functional components to consume context provided by a `Context.Provider` higher up in the component tree without using props. It simplifies the process of passing data through the component tree without having to manually pass props at every level.



React useRef:

useRef is a Hook that returns a mutable ref object whose current property can hold a reference to a DOM element or any mutable value. It's commonly used to access DOM elements or to persist values between renders without causing a re-render.

React useReducer:

useReducer is a Hook that is an alternative to useState. It's used for managing more complex state logic where state transitions depend on the previous state and actions dispatched to the reducer function. It's similar to the reducer concept in Redux.

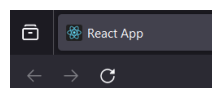
React useCallback:

useCallback is a Hook that returns a memoized callback function. It's useful for optimizing performance by preventing unnecessary re-renders in child components that rely on functions passed down from parent components. It memoizes the function so that it's only recreated if its dependencies change.

React useMemo:

useMemo is a Hook that memoizes the result of a function and returns the memoized value. It's used to optimize performance by caching the result of expensive computations so that they're only recalculated when necessary. It memoizes values based on a dependency array, and it's useful for avoiding unnecessary recalculations in functional components.

```
1  App.js  X
src > JS App.js > _
2  import React, { useState, useMemo } from 'react';
3
4  Codeium: Refactor | Explain | Generate JS/Doc | X
5  const ExampleComponent = () => {
6    const [number, setNumber] = useState(0);
7
8    // Expensive calculation function
9    Codeium: Refactor | Explain | X
10   const calculateSquare = (num) => {
11     console.log('calculating square...');
12     return num * num;
13   };
14
15   // Memoize the expensive calculation using useMemo
16   const squaredNumber = useMemo(() => calculateSquare(number), [number]);
17
18   return (
19     <div>
20       <h2>useMemo</h2>
21       <p>Number: {number}</p>
22       <p>Squared Number: {squaredNumber}</p>
23       <button onClick={() => setNumber(number + 1)}>Increment</button>
24     </div>
25   );
26 }
27
28 export default ExampleComponent;
```



useMemo

Number: 12

Squared Number: 144

Increment

UseLocation:

- import useLocation from the react-router-dom library.
- use the useLocation hook to get the current location object.
- The useLocation hook allows you to access the current URL location in your components. This can be useful for various purposes, such as conditionally rendering components based on the current URL, accessing query parameters, or tracking user navigation.



React Class

1. **Class Declaration:** The class keyword is used to declare a new class named MyClassComponent, which extends the Component class from the react module.
2. **Constructor:** The constructor method is called when an instance of the class is created. It is used to initialize the component's state and bind event handlers.
3. **State:** this.state is used to initialize the component's state. In this example, the state has a property called count with an initial value of 0.
4. **Event Handling:** The incrementCount method is a custom method that updates the component's state when called. It is attached to a button's onClick event.
5. **Render Method:** The render method is required for every React class component. It returns the JSX that defines the component's UI. The JSX can access the component's state and props.
6. **Export:** The class component is exported to make it available for use in other parts of the application.

To use this class component in another file, you would typically import it and render it within a parent component or directly using ReactDOM.render().

React Forms

React Forms are used to handle user input within React applications. Forms allow users to input data through various elements like text fields, checkboxes, radio buttons, etc. React provides a way to manage form state and handle form submission efficiently.

Controlled Components:

In React, form elements such as <input>, <textarea>, and <select> typically maintain their state in the component's state by updating the state with the onChange event. These components are referred to as controlled components.

Uncontrolled Components:

Uncontrolled components allow form elements to maintain their own state without being controlled by React's state. They are generally used when integrating React with non-React libraries or for handling form input less verbosely.

Form Libraries:

React also supports integration with form libraries such as Formik, React Hook Form, and Redux Form, which provide additional features like form validation, error handling, and easier state management.

React Memo



`React.memo()` is a higher-order component (HOC) provided by React that's used for optimizing functional components by memoizing them. It's similar to the concept of `shouldComponentUpdate()` in class components, but for functional components.

How `React.memo()` Works: When you wrap a component with `React.memo()`, React will memoize the result of rendering that component. If the props of the component remain the same between renders, React will reuse the previously rendered result, thus preventing unnecessary re-renders.

When to use `React.memo`:

- Use `React.memo` for functional components that render the same result given the same props.
- It's particularly useful for optimizing functional components that render large trees or are rendered frequently.

When not to use `React.memo`:

- Avoid using `React.memo` on components that frequently change their props or have frequent re-renders, as it may negate its performance benefits.

Note: `React.memo` only shallowly compares props. If props contain complex data structures, it's important to ensure that their reference doesn't change unnecessarily to benefit from memoization.

These concepts are essential for building robust and efficient React applications. They enable developers to create dynamic user interfaces, manage application routing, and optimize performance effectively.

React CSS Styling:

React CSS Styling offers multiple methods for styling components.

React provides several methods for styling components:

1. **Inline Styles:** You can apply styles directly to individual components using the `style` attribute.
2. **CSS Files:** You can import CSS files and apply styles using class names.
3. **CSS Modules:** CSS Modules allow scoping CSS locally to a component.
4. **CSS-in-JS Libraries:** Libraries like `styled-components` or `Emotion` enable writing CSS directly inside JavaScript files.

These approaches offer flexibility and cater to different preferences and project requirements.

React SASS Styling: To use Sass (Syntactically Awesome Stylesheets) with React, you'll typically set up your project to compile Sass files into regular CSS files, which you can then import into your React components.



1. **Install Sass:** First, you need to install Sass in your project. You can do this using npm → **npm install node-sass** .
2. **Create Sass files:** Create your Sass files with the **.scss** extension. You can organize your Sass files however you like, but it's common to have a separate folder for styles.
3. **Import Sass files into your components:** You can import Sass files directly into your React components.
4. **Compile Sass:** You need to compile your Sass files into regular CSS files. This can be done using build tools like Webpack, Parcel, or Gulp, or through scripts in your **package.json**.
5. **Run your project:** After setting up Sass and compiling your styles, you can run your React project as usual.

Why saas?

- Sass allows you to define variables that can be reused throughout your stylesheets. This makes it easy to maintain consistent styles across your application by centralizing color schemes, font sizes, spacing, and other style properties.
- Sass allows you to break your stylesheets into smaller, modular files called partials. You can then import these partials into your main stylesheet, making it easier to organize and maintain your styles.
- Sass provides built-in functions and operators that allow for more complex and dynamic styles. This includes mathematical operations, color manipulation, string interpolation, and more.
- Saas is a efficient ways of write style.

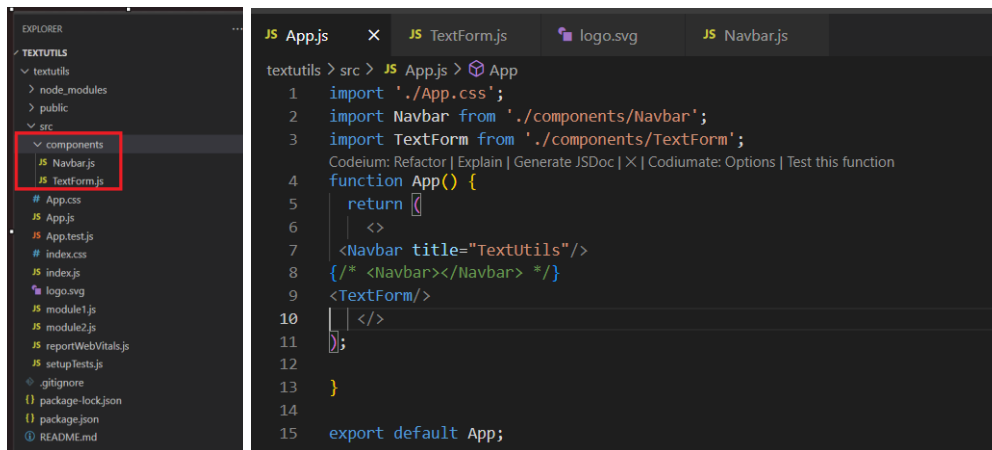
Step by step application making process:

- In project window. ->cd textutils->npm start.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS  COMMENTS
Compiled successfully!
You can now view textutils in the browser.
Local:      http://localhost:3000
On Your Network: http://192.168.153.1:3000
Note that the development build is not optimized.
To create a production build, use npm run build.
webpack compiled successfully
```

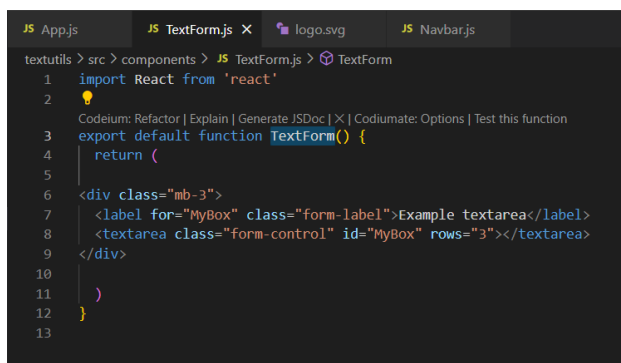
- Open with react app→ <http://192.168.153.1:3000> .
- Open with browser→ <http://localhost:3000> .

Step-1:



The image shows the VS Code Explorer on the left with the 'components' folder expanded, highlighting 'Navbar.js' and 'TextForm.js'. The main editor shows 'App.js' with the following code:

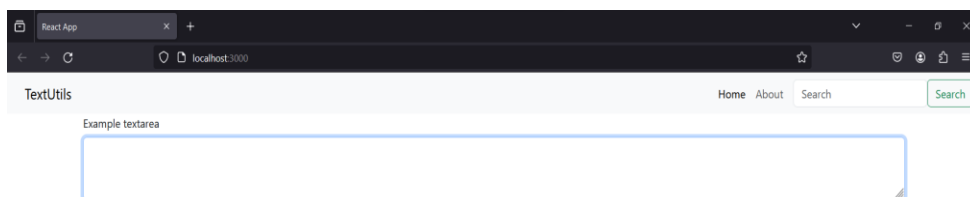
```
1 import './App.css';
2 import Navbar from './components/Navbar';
3 import TextForm from './components/TextForm';
4 function App() {
5   return (
6     <>
7     <Navbar title="TextUtils"/>
8     { /* <Navbar></Navbar> */ }
9     <TextForm />
10    </>
11  );
12 }
13
14 export default App;
```



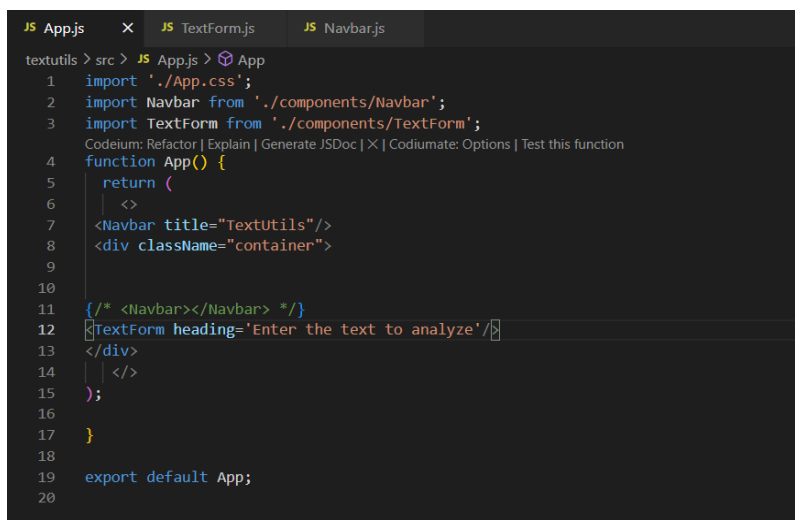
The image shows the VS Code editor with 'TextForm.js' open, displaying the following code:

```
1 import React from 'react'
2
3 export default function TextForm() {
4   return (
5
6     <div class="mb-3">
7       <label for="MyBox" class="form-label">Example textarea</label>
8       <textarea class="form-control" id="MyBox" rows="3"></textarea>
9     </div>
10
11   )
12 }
13
```

Output will display like this:



Step-2:



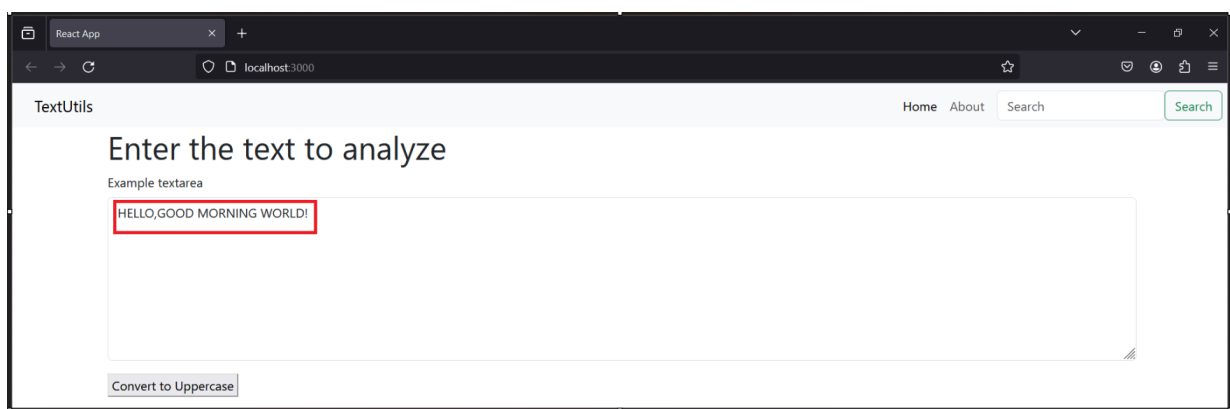
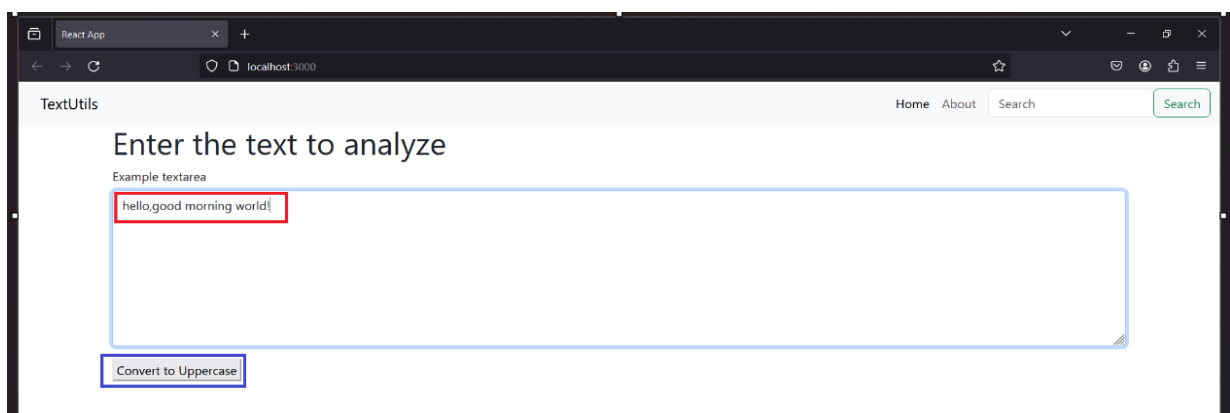
The image shows the VS Code editor with 'App.js' open, displaying the following code:

```
1 import './App.css';
2 import Navbar from './components/Navbar';
3 import TextForm from './components/TextForm';
4 function App() {
5   return (
6     <>
7     <Navbar title="Textutils"/>
8     <div className="container">
9
10      { /* <Navbar></Navbar> */ }
11      <TextForm heading='Enter the text to analyze' />
12    </div>
13    </>
14  );
15 }
16
17 export default App;
```



```
App.js | TextForm.js | NavBar.js
textutils > src > components > TextForm.js > handleOnChange
1 import React, {useState} from 'react'
2
3 export default function TextForm(Props) {
4   const handleClick={() => {
5     console.log("uppercase was clicked "+ text);
6     let newText= text.toUpperCase();
7     setText(newText)
8   }}
9   const handleChange=(event) => {
10    console.log("on change");
11    setText(event.target.value);
12  }
13  const [text, setText]=useState("Enter text here");
14  return (
15    <div>
16      <h1>{Props.heading}</h1>
17      <div class="mb-3">
18        <label for="myBox" class="form-label">Example textarea</label>
19        <textarea class="form-control" value={text} onChange={handleChange} id="myBox" rows="8"></textarea>
20      </div>
21      <button className="btn btn-primary" onClick={handleClick}>Convert to Uppercase</button>
22    </div>
23  )
24 }
25
26
```

Output will display like this:

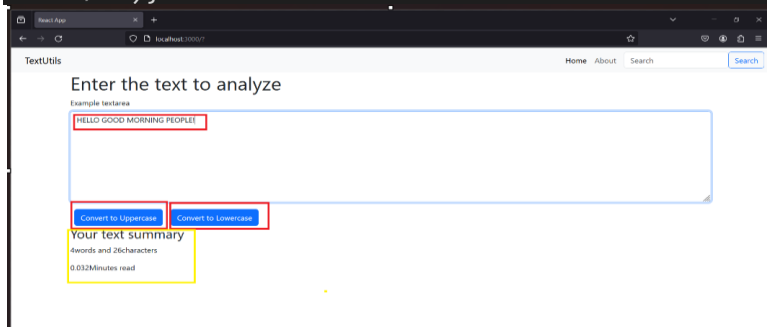


Step-3: TextForm.js

```
import React, {useState} from 'react'
export default function TextForm(Props) {
  const handleClick={() => {
    console.log("uppercase was clicked "+ text);
    let newText= text.toUpperCase();
    setText(newText)
  }}
  const handleLoClick={() => {
    console.log("lowercase was clicked "+ text);
```



```
    let newText= text.toLowerCase();
    setText(newText)}
    const handleChange=(event)=>{
      console.log("On Change");
      setText(event.target.value);}
    const [text, setText]=useState("");
    return (
      <>
      <div className='container'>
      <h1>{Props.heading}</h1>
      <div class="mb-3">
        <label for="MyBox" class="form-label">Example textarea</label>
        <textarea class="form-control" value={text} onChange={handleChange}
id="MyBox" rows="8"></textarea>
      </div>
      <button Class= "btn btn-primary mx-2" onClick={handleUpClick}>Convert to
Uppercase</button>
      <button Class= "btn btn-primary mx-2" onClick={handleLoClick}>Convert to
Lowercase</button>
      </div>
      <div className="contaner">
        <h3>Your text summary</h3>
        <p>
          {text.split(" ").length}words and {text.length}characters</p>
        <p>{0.008*text.split(" ").length }Minutes read</p>
        </div>
      </> )}
```



Step-4:

```
JS App.js x JS TextForm.js JS About.js JS NavBar.js
textutils > src > JS App.js > ...
1 import './App.css';
2 // import About from './components/About';
3 import NavBar from './components/NavBar';
4 import TextForm from './components/TextForm';
5 import React, {useState} from 'react';
6 Codium: Refactor | Explain | Generate JSDoc | X | Codiumate: Options | Test this function
7 function App() {
8   // const [mode, setMode]=useState('light');
9   return (
10     <>
11     <NavBar title="TextUtils"/>
12     <div className="container my-3">
13     { /* <NavBar></NavBar> */ }
14     <TextForm heading='Enter the text to analyze below'/>
15     { /* <About> */ }
16     </div>
17     </>
18   );
19 }
20 export default App;
```

```
JS App.js JS TextForm.js JS About.js JS NavBar.js
textutils > src > components > JS TextForm.js > ...
1 import React, {useState} from 'react';
2 Codium: Refactor | Explain | Generate JSDoc | X | Codiumate: Options | Test this function
3 export default function TextForm(Props) {
4   Codium: Refactor | Explain | Generate JSDoc | X
5   const handleUpClick=()=>{
6     console.log("uppercase was clicked "+ text);
7     let newText= text.toUpperCase();
8     setText(newText)
9   }
10   Codium: Refactor | Explain | Generate JSDoc | X
11   const handleLoClick=()=>{
12     // console.log("lowercase was clicked "+ text);
13     let newText= text.toLowerCase();
14     setText(newText)
15   }
16   Codium: Refactor | Explain | Generate JSDoc | X
17   const handleClearClick=()=>{
18     let newText= '';
19     setText(newText)
20   }
21   Codium: Refactor | Explain | Generate JSDoc | X
22   const handleChange=(event)=>{
23     // console.log("On Change");
24     setText(event.target.value);
25   }
26   Codium: Refactor | Explain | Generate JSDoc | X
27   const handleCopy=()=>{
28     // console.log("im a copy");
29     var text = document.getElementById("myBox");
30     text.select();
31     text.setSelectionRange(0,9999);
32     navigator.Clipboard.writeText(text.value);
33   }
34 }
```



```
JS App.js JS TextForm.js JS About.js JS Navbar.js
textutils > src > components > JS TextForm.js > ...
2 export default function TextForm(Props) {
28   const handleExtraSpaces = () => {
29     let newText = text.split(/[ ]+/);
30     setText(newText.join(" "))
31   }
32   const [text, setText] = useState('');
33   return (
34     <>
35     <div className='container'>
36     <h1>{Props.heading}</h1>
37     <div class="mb-3">
38       <label for="MyBox" class="form-label">Example textarea</label>
39       <textarea class="form-control" value={text} onChange={handleOnChange} style={{}} id="MyBox" rows="8"></textarea>
40     </div>
41     <button Class= "btn btn-primary mx-2" onClick={handleUpClick}>Convert to Uppercase</button>
42     <button Class= "btn btn-primary mx-2" onClick={handleLoClick}>Convert to Lowercase</button>
43     /* <button Class= "btn btn-primary mx-2" onClick={handleReverse}>Reverse Text</button> */
44     <button Class= "btn btn-primary mx-2" onClick={handleClearClick}>Clear Text</button>
45     <button Class= "btn btn-primary mx-2" onClick={handleCopy}>Copy Text</button>
46     <button Class= "btn btn-primary mx-2" onClick={handleExtraSpaces}>Remove Extra Spaces</button>
47   </div>
48   <div className="contaner">
49     <h3>Your text summary</h3>
50     <p>
51       {text.split(" ").length}words and {text.length}characters</p>
52     <p>{0.008*text.split(" ").length }Minutes read</p>
53     <h2>Preview</h2>
54     <p>{'Enter something in the title box to preview here'}</p>
55   </div>
56   </>
57 }
```

React App

localhost:3000

TextUtils

Home

Search

Search

Enter the text to analyze below

Example textarea

Convert to Uppercase

Convert to Lowercase

Clear Text

Copy Text

Remove Extra Spaces

Your text summary

1words and 0characters

0.008Minutes read

Preview

Enter something in the title box to preview here



11.React Redux

Make folder ReduxApp→open with vs code→open terminal→npx-create-react-app.

Step 1: Create a React Application

If you haven't already, start by creating a new React application using Create React App. You can do this by running the following command in your terminal: → npx create-react-app my-redux-app

Replace **my-redux-app** with the name of your application.

Step 2: Install Redux

Next, navigate to your project directory and install Redux and React Redux as dependencies:

cd my-redux-app→ npm install redux react-redux

Step 3: Set Up the Redux Store

Create a directory called **redux** inside the **src** directory of your project. Inside the **redux** directory, create a file called **store.js**:

Step 4: Define Reducers

Inside the **redux** directory, create a directory called **reducers**. In this directory, you can define your reducers. Reducers specify how the application's state changes in response to actions sent to the Redux store.

Step 5: Combine Reducers

Create a file called **index.js** inside the **reducers** directory to combine all your reducers:

Step 6: Connect Redux to React

In your **src** directory, you can connect Redux to your React application by wrapping your main component with the Redux **Provider** and passing the Redux store as a prop:

Step 7: Use Redux in Components

Now you can use Redux in your React components by connecting them to the Redux store using the **connect** function from **react-redux**.

This are the building parts of react:

1. **Action Types:** Constants that define the types of actions that can be dispatched.
2. **Action Creators:** Functions that create and return action objects.
3. **Reducers:** Pure functions that specify how the application's state changes in response to actions.
4. **Store:** The object that brings the actions, reducers, and application state together.



```
JS actions.js X JS index.js JS App.js JS actionTypes.js
src > JS actions.js > ...
1 // actions.js
2
3 Codeium: Refactor | Explain | Generate JSDoc | X
4 export const deposit = (accountType, amount) => {
5   return {
6     type: 'DEPOSIT',
7     payload: {
8       (property) amount: any
9     },
10   };
11 };
12
13 Codeium: Refactor | Explain | Generate JSDoc | X
14 export const withdraw = (accountType, amount) => {
15   return {
16     type: 'WITHDRAW',
17     payload: {
18       accountType,
19       amount
20     },
21   };
22 };
```

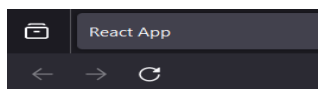
```
JS actions.js JS index.js X JS App.js JS
src > components > JS index.js
1 // index.js
2
3 import React from 'react';
4 import ReactDOM from 'react-dom';
5 import { Provider } from 'react-redux';
6 import store from './store';
7 import App from './App';
8
9 ReactDOM.render(
10   <Provider store={store}>
11     <App />
12   </Provider>,
13   document.getElementById('root')
14 );
15
```

```
JS actions.js JS index.js JS App.js JS actionTypes.js JS reducers.js JS shop.js
src > JS App.js > ...
1
2
3 import React from 'react';
4 import { useSelector, useDispatch } from 'react-redux';
5 import { deposit, withdraw } from './actions';
6
7 Codeium: Refactor | Explain | Generate JSDoc | X
8 const App = () => {
9   const checking = useSelector(state => state.checking);
10   const savings = useSelector(state => state.savings);
11   const dispatch = useDispatch();
12
13   return (
14     <div>
15       <h1>Bank Details</h1>
16       <h2>Checking: {checking}</h2>
17       <h2>Savings: {savings}</h2>
18       <button onClick={() => dispatch(deposit('checking', 1000))}>Deposit to Checking</button>
19       <button onClick={() => dispatch(withdraw('checking', 500))}>Withdraw from Checking</button>
20       <button onClick={() => dispatch(deposit('savings', 1000))}>Deposit to Savings</button>
21       <button onClick={() => dispatch(withdraw('savings', 100))}>Withdraw from Savings</button>
22     </div>
23   );
24 };
25 export default App;
```

```
JS actions.js JS index.js JS App.js JS actionTypes.js X
src > components > JS actionTypes.js > ...
1 export const INCREMENT = 'INCREMENT';
2 export const DECREMENT = 'DECREMENT';
```

```
JS actions.js JS index.js JS App.js JS actionTypes.js JS reducers.js JS shop.js
src > JS reducers.js > ...
1
2 const initialState = {
3   checking: 0,
4   savings: 0
5 };
6
7 Codeium: Refactor | Explain | Generate JSDoc | X
8 const bankReducer = (state = initialState, action) => {
9   switch (action.type) {
10     case 'DEPOSIT':
11       return {
12         ...state,
13         [action.payload.accountType]: state[action.payload.accountType] + action.payload.amount
14       };
15     case 'WITHDRAW':
16       return {
17         ...state,
18         [action.payload.accountType]: state[action.payload.accountType] - action.payload.amount
19       };
20     default:
21       return state;
22   }
23 };
24 export default bankReducer;
```

```
JS actions.js JS index.js JS App.js JS actionTypes.js JS reducers.js JS shop.js X
src > components > JS shop.js > @Shop
1 import React from 'react';
2 import { useDispatch, useSelector } from 'react-redux';
3 import { actionCreators } from './state/index';
4 import { bindActionCreators } from 'redux';
5 Codeium: Refactor | Explain | Generate JSDoc | X
6 export default function Shop() {
7   const amount = useSelector(state => state.amount);
8   const dispatch = useDispatch();
9   const { depositMoney, withdrawMoney } = bindActionCreators(actionCreators, dispatch);
10   return (
11     <div>
12       <h2>Deposit/Withdraw Money</h2>
13       <button className="btn btn-primary mx-3" onClick={() => withdrawMoney(1000)}>Withdraw 1000</button>
14       <div>Update Balance</div>
15       <button className="btn btn-primary mx-3" onClick={() => depositMoney(1000)}>Deposit 1000</button>
16       <button className="btn btn-primary mx-3" yourBalance={amount}>Your Balance</button>
17     </div>
18   );
19 }
```

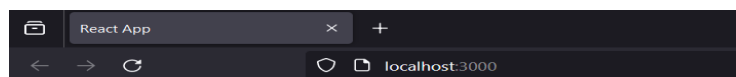


Bank Details

Checking: 1000

Savings: 0

Deposit to Checking



Bank Details

Checking: 1000

Savings: 1000

Deposit to Checking Withdraw from Checking Deposit to Savings Withdraw from Savings



Purpose of React and Redux:

- **React:** React is a JavaScript library for building user interfaces. It provides a declarative and component-based approach to building UIs, making it easier to create complex UIs by breaking them down into smaller, reusable components. React manages the UI layer of your application and provides tools for handling component state and lifecycle.
- **Redux:** Redux is a predictable state container for JavaScript apps. It provides a centralized store to manage the state of your entire application, making it easier to manage and update state across different parts of your application. Redux follows the principles of unidirectional data flow, which helps to maintain a predictable state and makes it easier to debug and test your application.

Redux and API integration

1. **Fetching Data from an API:** Redux can be used to manage the state of data fetched from APIs. Typically, you would dispatch actions to request data from the API, update the Redux store with the received data, and then use selectors to access this data from your components.
2. **Redux Thunk or Redux Saga:** To handle asynchronous actions, such as fetching data from an API, you can use middleware like Redux Thunk or Redux Saga.
 - Redux Thunk allows you to write action creators that return functions instead of plain objects, which enables you to perform asynchronous operations such as API requests.
 - Redux Saga is a more powerful alternative that uses ES6 generator functions to manage side effects like API calls in a more structured and composable way.
3. **Action Types and Action Creators:** Define action types to represent different stages of the API request (e.g., REQUEST, SUCCESS, FAILURE).
 - Create action creators to generate actions for initiating API requests, handling successful responses, and handling errors.
4. **Reducers:** Write reducers to update the state of your application in response to actions dispatched by the action creators.
 - Manage the state related to API requests (e.g., loading state, error state, fetched data) in your Redux store.
5. **Selectors:** Use selectors to extract data from the Redux store and pass it as props to your components.
 - Selectors allow you to derive data from the store state in a memoized and efficient way, helping to prevent unnecessary re-renders in your components.
6. **Middleware:** Middleware can be used to intercept actions before they reach the reducers, enabling you to perform additional logic such as logging, transforming actions, or handling side effects.