INDEX

01 What is React.js? Explain the Virtual DOM and its significance in React

React.js

React.js is a JavaScript library for building user interfaces (UIs). It follows a component-based approach, where complex UIs are broken down into smaller, reusable components. Each component specifies how a part of the UI should look and behave. React uses a declarative paradigm, meaning you describe what you want the UI to look like, and React handles the updates when the data or state changes.

Example:

```
function Profile() {
  return (
    <img
      src="https://i.imgur.com/MK3eW3As.jpg"
      alt="Katherine Johnson"
    />
  );
}

export default function Gallery() {
  return (
    <section>
      <h1>Amazing scientists</h1>
      <Profile />
      <Profile />
      <Profile />
    </section>
  );
}
```

Virtual DOM

React JS Virtual DOM is an in-memory representation of the **DOM**. **DOM** refers to the **Document Object Model** that represents the content of XML or HTML documents as a tree structure so that the programs can be read, accessed, and changed in the document structure, style, and content.

What is DOM

DOM stands for 'Document Object Model'. In simple terms, it is a structured representation of the HTML elements that are present in a webpage or web app. DOM represents the entire UI of your application. The DOM is represented as a tree data structure. It contains a node for each UI element present in the web document.

Disadvantages of DOM

Every time the DOM gets updated, the updated element and its children have to be rendered again to update the UI of our page. For this, each time there is a component update, the DOM needs to be updated and the UI components have to be re-rendered.

Example:

```
// Simple getElementById() method
document.getElementById('some-id').innerValue = 'updated value';
```

Virtual DOM

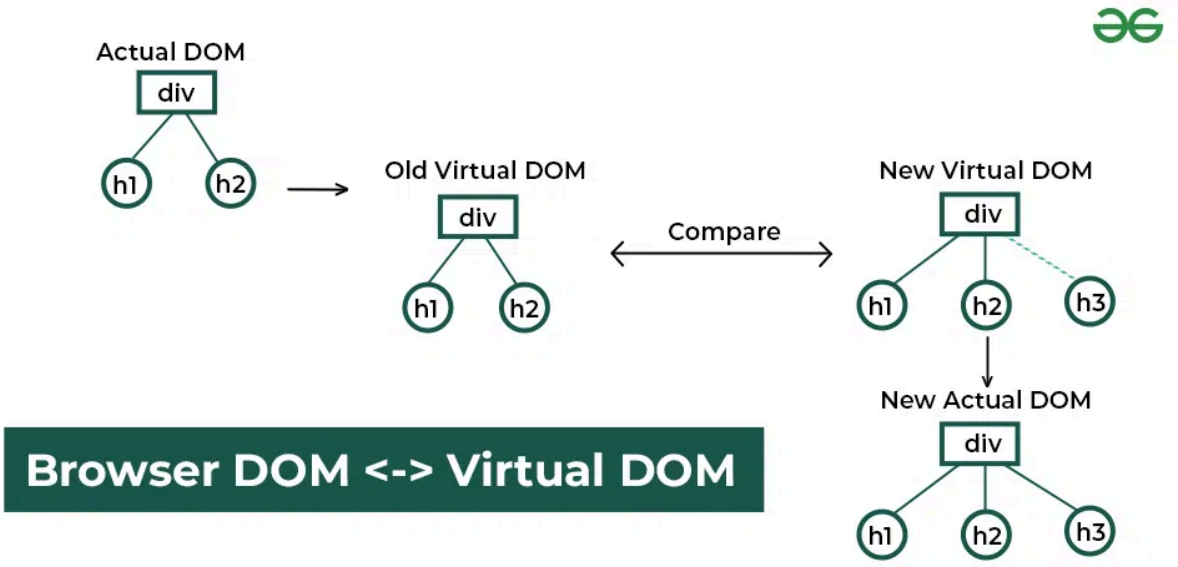
The Virtual DOM is a key concept in React that optimizes UI rendering performance. It's an in-memory representation of the actual DOM (Document Object Model), which is the browser's way of representing the web page structure. Here's how it works:

Component State Changes: When the state of a component in your React application changes (due to user interaction, data updates, etc.), React doesn't directly manipulate the real DOM.

Virtual DOM Update: Instead, it creates a new, updated virtual DOM tree that reflects the changes in the component's state. This update is lightweight and efficient.

Diffing: React then compares the previous virtual DOM tree with the new one to identify the minimal set of changes required in the real DOM. This process is called "diffing."

Actual DOM Updates: Finally, React applies only the necessary changes to the real DOM, ensuring an efficient update process. This minimizes the number of DOM manipulations, which can be slow in browsers.



Significance of Virtual DOM in React

The Virtual DOM offers several benefits for React applications:

- **Improved Performance:** By avoiding unnecessary DOM manipulations, React can update the UI efficiently, especially for complex UIs with frequent state changes.
- **Declarative Programming:** The virtual DOM allows you to focus on describing the desired UI state, and React handles the efficient updates behind the scenes.
- **Predictability:** Since React manages the DOM updates, you can be more confident that the UI will reflect the component's state consistently.
- **Batching:** React can batch multiple state updates and DOM changes together, further optimizing performance.

Overall, the Virtual DOM is a fundamental concept in React that contributes to its efficiency, ease of development, and overall user experience.

02 What is JSX in React? How does it differ from regular JavaScript syntax?

What is JSX ?

JSX stands for JavaScript XML. JSX is basically a syntax extension of JavaScript.

React JSX helps us to write HTML in JavaScript and forms the basis of React Development. Using JSX is not compulsory but it is highly recommended for programming in React as it makes the development process easier as the code becomes easy to write and read.

JSX

- **Looks similar to HTML:** JSX uses opening and closing tags to define elements, attributes, and content, resembling HTML syntax. This makes it easier for developers familiar with HTML to pick up React.
- **Embed JavaScript Expressions:** Within JSX, you can embed JavaScript expressions using curly braces `{}`. This allows you to dynamically generate content or attributes based on component state, props, or other variables.
- **Creates React Elements:** Although JSX looks like HTML, it's not directly converted into HTML. Instead, it's transformed during compilation into React elements, which are lightweight objects that represent the UI.

Example of JSX

Here is a simple example of JSX:

```
const element = <h1>Hello, world!</h1>;
```

This JSX code is equivalent to the following JavaScript code using React's createElement function:

```
const element = React.createElement('h1', null, 'Hello, world!');
```

How Does JSX Differ from Regular JavaScript Syntax?

1. **XML-Like Tags**:

JSX: Uses XML-like tags to define elements.

```
const element = <div>Hello, world!</div>;
```

JavaScript: Uses functions and objects to define elements.

```
const element = React.createElement('div', null, 'Hello, world!');
```

2. **HTML Attributes**:

JSX: Allows setting attributes directly on elements, similar to HTML.

```
const element = <img src="logo.png" alt="Logo" />;
```

JavaScript: Requires passing attributes as the second argument in the `createElement` function.

```
const element = React.createElement('img', { src: 'logo.png', alt: 'Logo'
});
```

3. **Embedding Expressions**:

JSX: Allows embedding JavaScript expressions inside curly braces `{}`.

```
const name = 'John';
const element = <h1>Hello, {name}!</h1>;
```

JavaScript: Requires concatenation or other means to include variables in strings.

```
const name = 'John';
const element = React.createElement('h1', null, `Hello, ${name}!`);
```

4. **Class and For Attributes**:

JSX: Uses `className` instead of `class` and `htmlFor` instead of `for` to avoid conflicts with JavaScript reserved words.

```
const element = <label htmlFor="input">Input</label>;
const element = <div className="container"></div>;
```

JavaScript: No such conflicts because it's purely function calls and objects.

```
const element = React.createElement('label', { htmlFor: 'input' },
'Input');
const element = React.createElement('div', { className: 'container' });
```

5. **Self-Closing Tags**:

JSX: Uses self-closing tags for elements without children, similar to HTML.

```
const element = <img src="logo.png" alt="Logo" />;
```

JavaScript: Requires `createElement` calls for each element, specifying `null` for no children.

```
const element = React.createElement('img', { src: 'logo.png', alt: 'Logo'
});
```

Transpilation of JSX

JSX is not valid JavaScript, so it cannot be interpreted by browsers directly. Instead, it needs to be transpiled into regular JavaScript using tools like Babel. Babel transforms JSX code into `React.createElement` calls, which the browser can then execute.

JSX is a syntax extension for JavaScript that allows you to write HTML-like code within your JavaScript files, making it easier to create and visualize the structure of your React components. It differs from regular JavaScript syntax in its use of XML-like tags, embedding expressions, handling of attributes, and requirement for transpilation to standard JavaScript.

03 Explain React class components with an Example

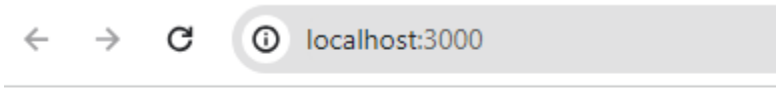
React class components are a traditional way to define reusable UI elements in React applications. They extend the built-in `React.Component` class and provide a clear separation of concerns between component structure (render method), state management, and lifecycle hooks.

Example 1: Create Class Component in React

Create a React app and edit the **App.js** as:

```
// Filename App.js

import React from "react";

class App extends React.Component {
  render() {
    return <h1>Mallareddy University</h1>;
  }
}

export default App;
```

Output:



Mallareddy University

Once a component is declared, it can be used in other components. Program to demonstrate the use of class components in other components.

Example 2: Using Class Components in React

This example demonstrate the creation and use of class component Sample.

```
// Filename - App.js

import React from "react";

class Sample extends React.Component {
  render() {
    return <h1>CSE department in Mallareddy University</h1>;
  }
}

class App extends React.Component {
  render() {
    return <Sample />;
  }
}

export default App;
```

Output:

04. Explain Functional components with an Example

Functional Component is one way to create components in a React Application. React.js Functional Components helps to create UI components in a Functional and more concise way. In this article, we will learn about functional components in React, different ways to call the functional component, and also learn how to create the functional components. We will also demonstrate the use of hooks in functional components

How do ReactJS functional components work?

Functional components receive input data through props, which are objects containing key-value pairs. Once the component receives props, it processes them and returns a JSX element that describes the component's structure and content. When the component is rendered, React creates a virtual DOM tree that represents the current state of the application. If the component's props or state change, the tree is updated accordingly, and the component is re-rendered.

Why use ReactJS functional components?

ReactJS functional components offer several benefits over class components, including:

- **Simplicity**: Functional components are simpler and easier to read than class components, making them ideal for small to medium-sized projects.
- **Performance**: Functional components are faster than class components because they don't use the **this** keyword, which can slow down rendering.
- **Testability**: Functional components are easier to test because they are stateless and don't rely on lifecycle methods.
- **Reusability**: Functional components can be reused across multiple projects, making them a great choice for building component libraries

Eample of Functinal Component

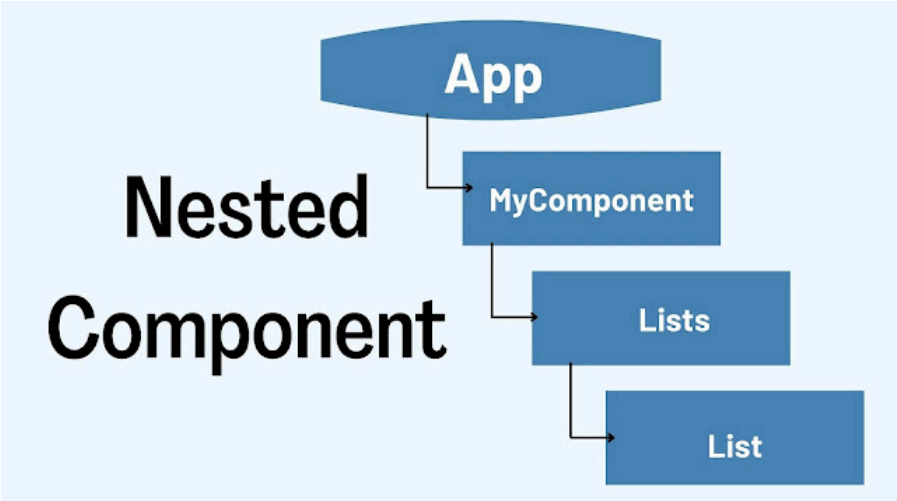
```
import React from 'react';

// Define a functional component
function Greeting(props) {
  return <h1>Hello, {props.name}!</h1>;
}
```

```
// Use the functional component in an App component
function App() {
  return (
    <div>
      <Greeting name="Alice" />
      <Greeting name="Bob" />
      <Greeting name="Charlie" />
    </div>
  );
}

export default App;
```

ReactJS functional components offer several benefits, including simplicity, performance, testability, and reusability. You can create a functional component by writing a JavaScript function that returns a JSX element, and use it in your project by importing it and passing props to it. With the right tools and techniques, you can ensure that your functional components are high-quality, reliable, and easy to maintain.

Nested Functional Components

In React, a nested component refers to a component that is used or rendered inside another component. This concept allows you to build complex UIs by composing simpler, reusable components. Nesting components is a key part of building modular and maintainable code in React.



Example:

app.js file

```
import React from 'react';
import OuterComponent from './OuterComponent';

function App() {
  return (
    <div>
      <h1>Nested Components Example</h1>
      <OuterComponent />
    </div>
  );
}

export default App;
```

OuterComponent.js

```
import React from 'react';
import InnerComponent from './InnerComponent';

function OuterComponent() {
  return (
    <div>
      <h2>This is the Outer Component</h2>
      <InnerComponent />
    </div>
  );
}

export default OuterComponent;
```

InnerComponent.js

```
import React from 'react';

function InnerComponent() {
  return (
    <div>
      <p>This is the Inner Component</p>
```

```
    </div>
  );
}

export default InnerComponent;
```

From above example you can get this conclusion:

App Component: The `App` component is the root component of our application. It renders the `OuterComponent`.

OuterComponent: The `OuterComponent` is a child of the `App` component. It renders the `InnerComponent`.

InnerComponent: The `InnerComponent` is a child of the `OuterComponent`. It simply displays a static message.

Benefits of Nested Components

1. **Reusability**: Components can be reused in different parts of the application.
2. **Modularity**: Breaking down the UI into smaller components makes the code more modular and easier to manage.
3. **Separation of Concerns**: Each component can focus on a specific piece of functionality or UI, improving readability and maintainability.
4. **Encapsulation**: Components encapsulate their logic and styles, reducing the likelihood of unintended side effects.

This approach makes the UI scalable and maintainable by clearly defining the role of each component and how they interact with each other.

05. What are props in React? How are they used to pass data between components?

Props in React are inputs that you pass into components. The props enable the component to access customized data, values, and pieces of information that the inputs hold.

The term 'props' is an abbreviation for 'properties' which refers to the properties of an object.

1. Passing data from Parent to Child in React

For passing data from parent to child component, we use props. Props data is sent by the parent component and cannot be changed by the child component as they are read-only.

Example: The following example covers how to pass data from Parent to Child Component in ReactJS.

App.js

```
// App.js
import React from "react";
import "./index.css";
import Parent from "./Parent";
import "./App.css";

const App = () => {
    return (
        <div className="App">
            <h1 className="geeks">Univeristy</h1>
            <h3>This is App.js Component</h3>
            <Parent />
        </div>
    );
};

export default App;
```

Parent.js

```
// Parent.js
import React from "react";
import Child from "./Child";

const Parent = () => {
    const data = "Data from Parent to Child";
    return (
        <div>
            <h4>This is Parent component</h4>
            <Child data={data} />
        </div>
    );
};

export default Parent;
```

Child.js

```
// Child.js
import React from "react";

const Child = (props) => {
    return <h3> {props.data} </h3>;
};

export default Child;
```

What is Prop Drilling?

Prop drilling, also known as "threading props" or "component chaining," refers to the process of passing data from a parent component down to nested child components through props.

Prop drilling occurs when a prop needs to be passed through several layers of nested components to reach a deeply nested child component that actually needs the prop. Each intermediary component in the hierarchy has to pass the prop down, even if it doesn't use the prop itself.

Consider a scenario where you have a top-level component that fetches data from an API and needs to pass this data down to multiple nested child components.

Instead of directly passing the data to each child component, you pass it through each intermediary component in the hierarchy until it reaches the desired child component. This passing of props through multiple levels of components is what prop drilling entails.

Example on Prop Drilling:

ParentComponent.js

```
// ParentComponent.js
import React from 'react';
import ChildComponent from './ChildComponent';

function ParentComponent() {
  const data = 'Hello from Parent';

  return (
    <div>
      <ChildComponent data={data} />
    </div>
  );
}

export default ParentComponent;
```

Child Component

```
// ChildComponent.js
import React from 'react';
import GrandchildComponent from './GrandchildComponent';

function ChildComponent(props) {
  return (
    <div>
      <GrandchildComponent data={props.data} />
    </div>
```

```
  );
}

export default ChildComponent;
```

Grandchild

```
// GrandchildComponent.js
import React from 'react';

function GrandchildComponent(props) {
  return <div>{props.data}</div>;
}

export default GrandchildComponent;
```

In this example, **GrandchildComponent** needs to access the data prop, but **ParentComponent** and **ChildComponent** do not use it. However, the data prop must still be passed through them.

06 Describe how the state is initialized and updated in a React component

07 How can props be passed from parent to child components? Provide examples.

In React, props (short for properties) are a way of passing data from parent to child components. This mechanism allows you to customize and configure child components according to their parent's state or data. Here's how you can pass props from parent to child components with examples.

ParentComponent renders **ChildComponent** and passes name and age as props.

ChildComponent receives these props and uses them to render content.

```
// ParentComponent.js
import React from 'react';
import ChildComponent from './ChildComponent';

const ParentComponent = () => {
  // Define prop values
  const name = 'John';
  const age = 30;

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

export default ParentComponent;
```

ChildComponent.js

```
// ChildComponent.js
import React from 'react';

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

```
    </div>
  );
};

export default ChildComponent;
```

Passing Props with Destructuring:

```
// ChildComponent.js
import React from 'react';

const ChildComponent = ({ name, age }) => {
  return (
    <div>
      <p>Name: {name}</p>
      <p>Age: {age}</p>
    </div>
  );
};

export default ChildComponent;
```

In this version, we're using destructuring in the function parameter of `ChildComponent` to directly extract `name` and `age` props. This makes the code cleaner and more concise.

```
// ParentComponent.js
import React from 'react';
import ChildComponent from './ChildComponent';

const ParentComponent = () => {
  const person = { name: 'John', age: 30 };

  return (
    <div>
      {/* Pass props using spread operator */}
      <ChildComponent {...person} />
    </div>
  );
};

export default ParentComponent;
```

```
// ChildComponent.js
import React from 'react';

const ChildComponent = (props) => {
  return (
    <div>
      <p>Name: {props.name}</p>
      <p>Age: {props.age}</p>
    </div>
  );
};

export default ChildComponent;
```

Here, we define a person object in ParentComponent and spread its properties as props to ChildComponent. This approach is useful when you have multiple props to pass.

These are some common ways of passing props from parent to child components in React. Each method offers flexibility and can be chosen based on your specific use case and coding style preferences.

08. What are React hooks? Provide examples of commonly used hooks and their purposes

Introduced in React version 16.8, hooks are functions that let you "hook into" and use state and other React features without having to write class components. This allows for cleaner, more concise, and easier to understand code.

Here are some commonly used hooks and their purposes:

- **`useState`**: This is the most fundamental hook, allowing you to add state to functional components. It returns an array with the current state value and a function to update it.
- **`useEffect`**: This hook lets you perform side effects in your components, such as data fetching, subscriptions, or manual DOM manipulation. It runs after the component renders.
- **`useContext`**: This hook provides a way to access React Context from functional components. Context is a way to share data across your application without having to pass props down every level of the component tree.
- **`useRef`**: This hook creates a mutable ref object that persists across re-renders. Refs are used to store imperative values that don't participate in the component's state, such as DOM node references or timers.

These are just a few of the many hooks available in React. Hooks provide a powerful and flexible way to build React applications, making them more maintainable and easier to reason about.

Example on Each hook:

1. useState

Creates an “state variable” which updates the component on change.

useState is the most common hook in React. The hook needs 3 inputs to create a state.

1. **Current state (count):** The name of the variable, which is equal to the current state.
2. **Function (setCount)**: A function which gets called to change the state.
3. **Initial value (0)**: The default value when page initialised.

```
import { useState } from "react";

const App = () => {
 const [count, setCount] = useState(0);

 return (
   <div>
     <h1>{count}</h1>
     <button onClick={() => setCount(count + 1)}>Add 1 to count</button>
   </div>
 );
};
```

2. useEffect:

A function which gets called every time the view gets mounted, or when the state inside [] changes.

useEffect is very often used in React. One of the key things to notice when using **useEffect**, is the second param: []. Their is a big difference between leaving the squared brackets empty, or entering a state(s).

- **Empty []**: This means that the useEffect() only gets called once upon mounting.
- **Filled [count]**: This means that the useEffect() gets called on mounting, and when the state (count) changes.

```
import { useEffect, useState } from "react";

const App = () => {
 const [count, setCount] = useState(0);

 useEffect(() => {
   console.log("View Mounted");
 }, []);

 useEffect(() => {
   console.log("View Mounted or Count updated");
 }, [count]);

 // .. //
};
```

3, useContext

Creates a state which is accessible from all components.

In this example we are looking at how to share a state variable among two components, which declares if the application is dark themed.
Normally you would have to pass this information through props, but this would be very painfull when working with multiple components.

Therefore we can use useContext, which creates an state variable, which can be accessed from all components, without the need for props.

```
// App.js //

import { useState, createContext } from "react";

export const ThemeContext = createContext(null);

const App = () => {
 const [isDarkThemed, setIsDarkThemed] = useState(false);

 return (
   <ThemeContext.Provider value={isDarkThemed}>
     <button onClick={() => setIsDarkThemed((prev) => !prev)}>
       Change Theme
     </button>
     <AboutPage />
   </ThemeContext.Provider>
 );
};

export default App;
```

We can then fetch the isDarkThemed value in AboutPage.js

```
// AboutPage.jsx //

import { useContext } from "react";
import { ThemeContext } from "./App";
```

```
const AboutPage = () => {
 const isDarkTheme = useContext(ThemeContext);

 return (
   <div style={{ backgroundColor: isDarkTheme ? "black" : "white" }}>
     // .. //
   </div>
 );
};
```

4. useRef

A hook which can store a mutable value, which does not re-render the view on update. Also, it can be used to store DOM elements.

The most common use of useRef, is to store an element, which then can be accessed within the component. This can be useful when creating a input field, where you e.g. need to access the value.

```
import { useRef } from "react";

const App = () => {
 const inputRef = useRef();

 return (
   <input
     ref={inputRef}
     onChange={() => {
       console.log(inputRef.current.value);
     }}
   />
 );
};
```

09. Develop a React component called Greetings that displays a simple greeting message. Render it in the App component. Pass a prop to the Greetings component to customize the greeting message.

To create a React component called `Greetings` that displays a simple greeting message and renders it in the `App` component with a customizable greeting message passed as a prop, follow these steps:

Greetings.js

```
// src/Greetings.js
import React from 'react';
import './Greetings.css'; // Import the CSS file for styling

const Greetings = ({ message }) => {
  return (
    <div className="greetings-container">
      <h1 className="greeting-message">{message}</h1>
    </div>
  );
};

export default Greetings;
```

src/Greetings.css

```
/* src/Greetings.css */
.greetings-container {
  background-color: #f0f0f0;
  padding: 20px;
  border-radius: 8px;
  box-shadow: 0px 4px 8px rgba(0, 0, 0, 0.1);
  text-align: center;
  max-width: 400px;
  margin: 0 auto;
}

.greeting-message {
```

```
  color: #333;
  font-size: 24px;
}
```

src/App.js

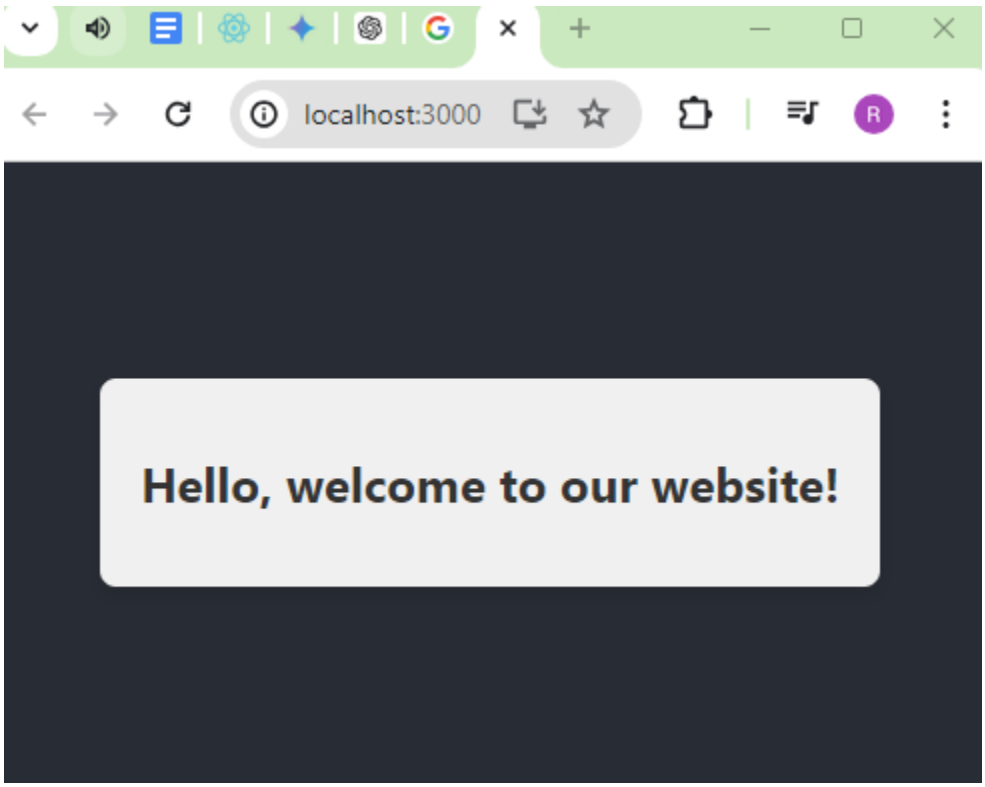
```
// src/App.js
import React from 'react';
import Greetings from './Greetings';
import './App.css';

function App() {
  const greetingMessage = "Hello, welcome to our website!";

  return (
    <div className="App">
      <header className="App-header">
        <Greetings message={greetingMessage} />
      </header>
    </div>
  );
}

export default App;
```

src/App.css

```
/* src/App.css */
.App {
  text-align: center;
}

.App-header {
  background-color: #282c34;
  min-height: 100vh;
  display: flex;
  flex-direction: column;
  align-items: center;
  justify-content: center;
  font-size: calc(10px + 2vmin);
  color: white;
```

```
}

.logo {
  height: 40vmin;
  pointer-events: none;
}

@media (prefers-reduced-motion: no-preference) {
  .logo {
    animation: App-logo-spin infinite 20s linear;
  }
}

@keyframes App-logo-spin {
  from {
    transform: rotate(0deg);
  }
  to {
    transform: rotate(360deg);
  }
}
```

Output:



10.Create a Counter component that displays a count and has buttons to increment and decrement the count. Implement state management within the Counter component to keep track of the count

Step 1: Create a React application using the following command

`npx create-react-app counter`

Step 2: After creating your project folder i.e. counter, move to it using the following command:

`cd counter`

Example: This example demonstrate a simple counter app using useState hook, and the count increase and descrease on click.

App.js:

```
// App.js

import React, { useState } from "react";
import './Counter.css'

const App = () => {
    const [counter, setCounter] = useState(0);

    const handleClick1 = () => {
        setCounter(counter + 1);
    };

    const handleClick2 = () => {
        setCounter(counter - 1);
    };
```

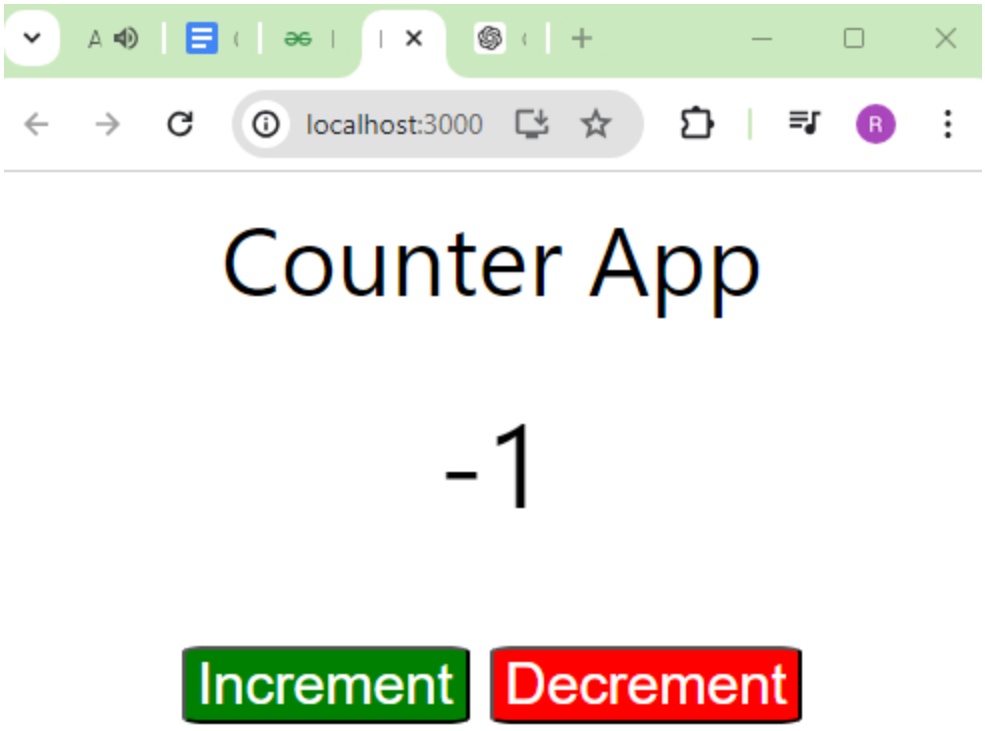
```
    return (
        <div className="app-container">
            Counter App
            <div className="counter-display">{counter}</div>
            <div className="buttons">
                <button className="increment-button"
onClick={handleClick1}>
                    Increment
                </button>
                <button className="decrement-button"
onClick={handleClick2}>
                    Decrement
                </button>
            </div>
        </div>
    );
};

export default App;
```

For styling purposes, we created the below stylesheet with the name Counter.css and ket in the same folder

```
/* counter.css */

.app-container {
    display: flex;
    flex-direction: column;
    align-items: center;
    justify-content: center;
    font-size: 300%;
    position: absolute;
    width: 100%;
    height: 100%;
    top: -15%;
}
```

```
.counter-display {
    font-size: 120%;
    position: relative;
    top: 10vh;
}

.buttons {
    position: relative;
    top: 20vh;
}

.increment-button,
.decrement-button {
    font-size: 60%;
    margin: 0 5px;
    border-radius: 8%;
    color: white;
}

.increment-button {
    background-color: green;
}

.decrement-button {
    background-color: red;
}
```

The output is:


localhost:3000
Counter App
-1
Increment
Decrement