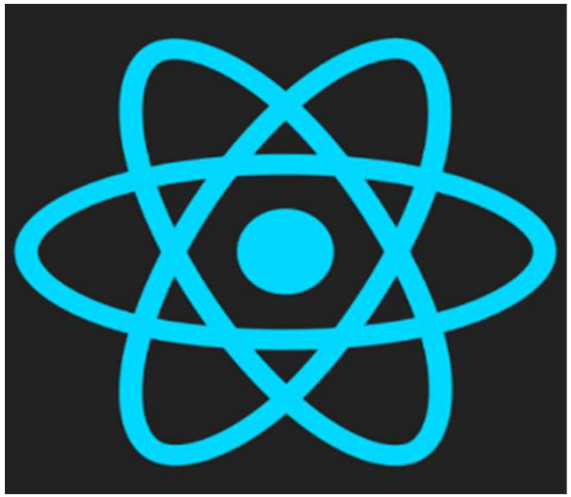
20ITC08

FULL STACK DEVELOPMENT

UNIT-III

React JS

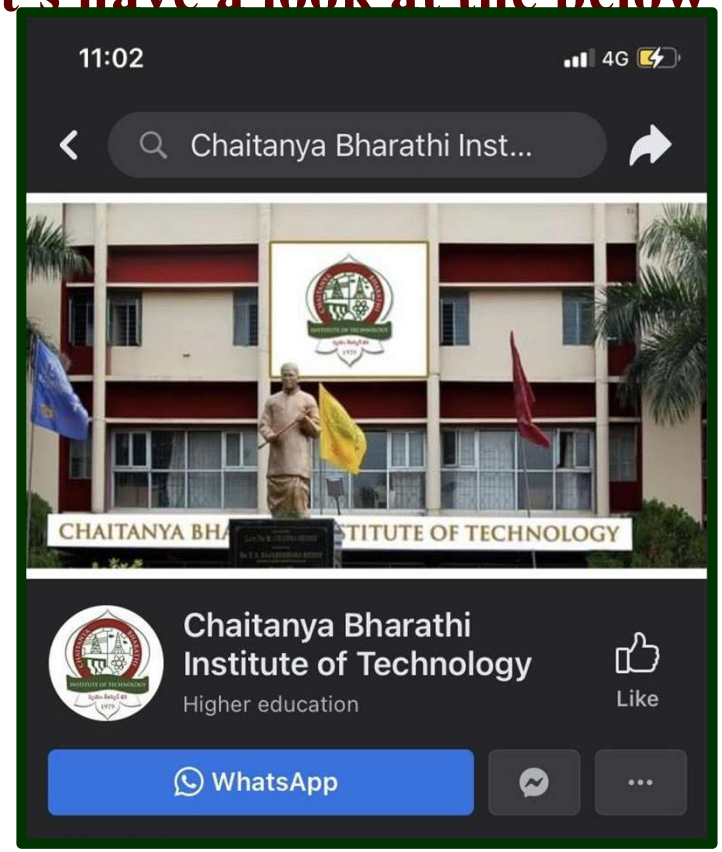


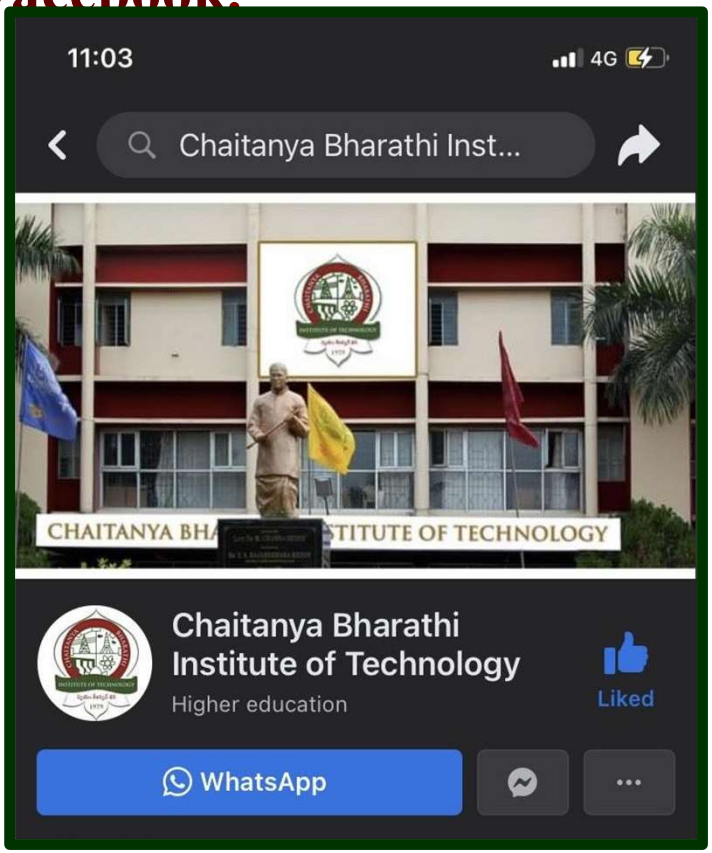
React JS

- **ReactJS, a JavaScript library** for creating user interfaces, makes the development of UI components easy and modular.
- **React JS was created by Jordan Walke**, a software engineer at Facebook and **open sourced** to the world **by Facebook and Instagram**.

Why React JS?

Let's have a look at the below post on Facebook.





Why React JS?

- The Facebook application has to do the changes by **updating DOM tree** of the page and **re-rendering the page** in browser.
- Considering the application **size of Facebook** which has more than **1.6 billion daily users and 500,000 "likes" every minutes**, this was a challenge to the engineers of Facebook.
- **React JS library** which optimizes DOM manipulation by writing very simple code.

React VS Angular

React	Angular
Facebook Community-March 2013	Google-October 2010
React is a small view library	Angular is a full framework
React covers only the rendering and event handling part	Angular provides the complete solution for front-end development
Presentation code in JavaScript powered by JSX	Presentation code in HTML embedded with JavaScript expressions
React's core size is smaller than Angular, so bit fast	Angular being a framework contains a lot of code, resulting in longer load time
React is very flexible	Angular has less flexibility
Great performer, since it uses Virtual DOM	Angular uses actual DOM which affects its performance

ReactJS VS React Native

ReactJS	React Native
The ReactJS initial release was in 2013.	The React Native initial release was in 2015.
It is used for developing web applications.	It is used for developing mobile applications.
It uses a JavaScript library and CSS for animations.	It comes with built-in animation libraries.
It uses React-router for navigating web pages.	It has built-in Navigator library for navigating mobile applications.
It uses HTML tags.	It does not use HTML tags.
It can use code components, which saves a lot of valuable time.	It can reuse React Native UI components & modules which allow hybrid apps to render natively.
In this, the Virtual DOM renders the browser code.	In this, Native uses its API to render code for mobile applications.

React Features

1. Component based architecture

2. Virtual DOM

3.Unidirectional data flow

4. JSX

5. SEO performance

React Installation

To install create-react-app, follow the below steps:

1. **Install node.js** with version 14+ from Node.js official site or from Software House.

2. Install create-react-app by running the following command:

D:\>npm install -g create-react-app

3. Once the installation is done, create a React app using the below command:

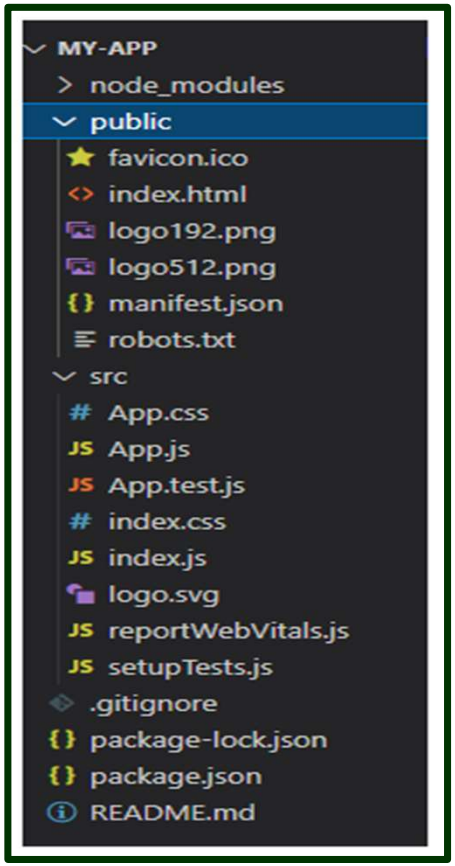
D:\>create-react-app my-app

React Installation

- If you do not want to install create-react-app then you can use the below command to create a React application.

D:\>npx create-react-app my-app

Files	Purpose
node_modules	All the node module dependencies are created in this folder.
public	This folder contains the public static assets of the application.
public/index.html	This is the first page that gets loaded when you run the application.
src	All application related files/folders are created in this folder.
src/index.js	This is the entry point of the application.
package.json	Contains the dependencies of the React application.



- To run the application: **D:/>my-app>npm start**

React System Configuration - Internal

Observe the file src/index.js :

ReactDOM.render(<h1>HelloReact!<h1/>, document.getElementById('root'));

ReactDOM.render is used to render an element or component to the virtual DOM.

- The **first argument** specifies **what needs to be rendered.**
- The **second argument** specifies **where to render.**

The root element is present inside index.html:

```
<body>
   <noscript>You need to enable JavaScript to run this app.</noscript>
   <div id="root"></div>
</body>
```

DOM [Document Object Model]s

- 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**.

// Simple getElementById() method

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

DOM [Document Object Model]

When writing the above code in the console or in the JavaScript file:

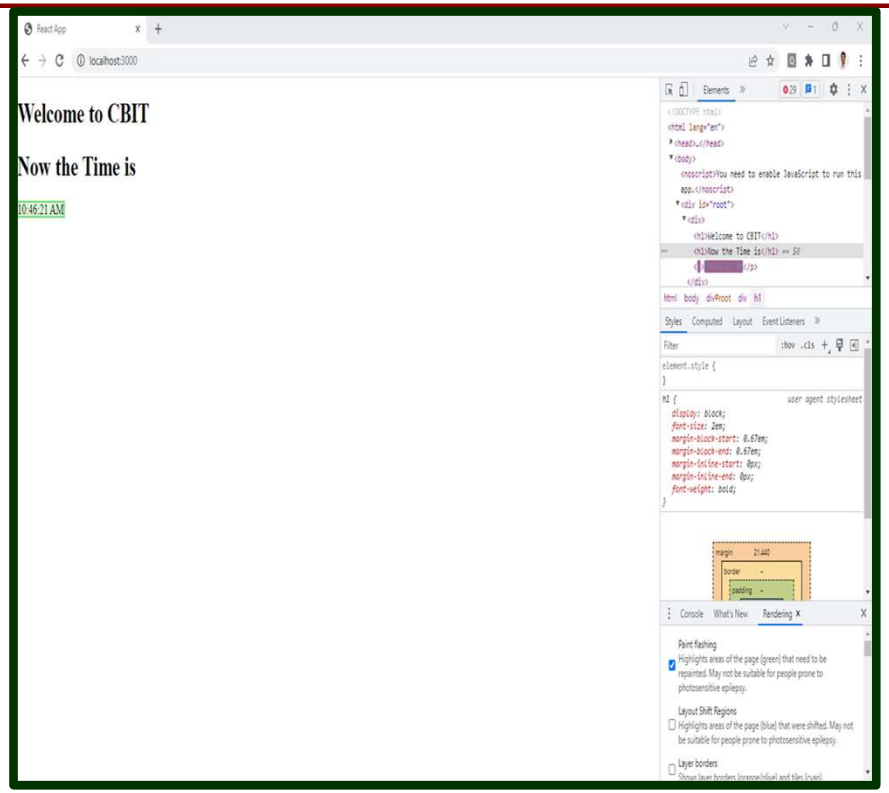
- The browser parses the HTML to find the node with this id.
- It removes the child element of this specific element.
- Updates the element(DOM) with the ‘updated value’.
- Recalculates the CSS for the parent and child nodes.
- Update the layout.
- Finally, traverse the tree and paint it on the screen(browser) display.

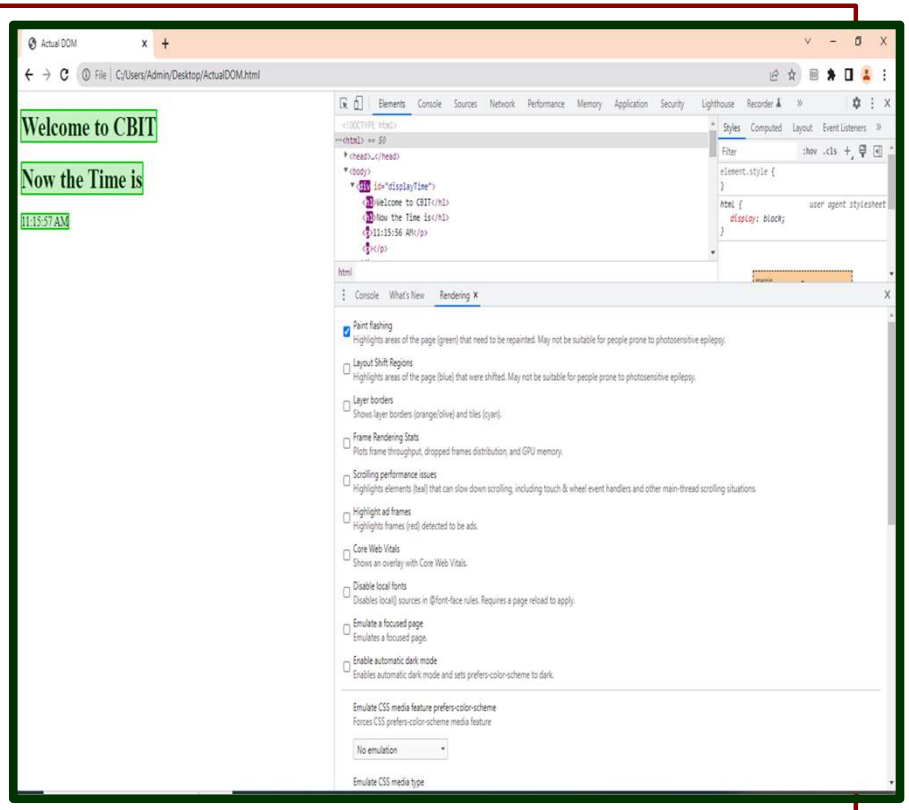
Actual DOM VS React VDOM

```
<!DOCTYPE html>
<html>
 <head>
    <title>Actual DOM</title>
 </head>

 <body>
    <div id='displayTime'></div>
 </body>
        <script>
            function updateTime()
            {
            var element  =  '<h1>Welcome to CBIT</h1>';
                element +=  '<h1>Now the Time is</h1>';
                element +=   '<p>'+new Date().toLocaleTimeString()+'<p>';
                document.getElementById('displayTime').innerHTML=element;
            }
            window.setInterval(updateTime, 1000);
</script>

</html>
```

```
import ReactDOM from 'react-dom';

function App()
{
 const element=
 (
    <div >
      <h1>Welcome to CBIT</h1>
      <h1>Now the Time is</h1>
      <p>{new Date().toLocaleTimeString()}</p>
    </div>
  );
  ReactDOM.render(element, document.getElementById('root'));
}
setInterval(App, 1000);
```

Actual DOM VS React VDOM





What is JSX?

JSX stands for JavaScript XML or Extension.

With the help of JSX we can write and add HTML in React.

JSX converts HTML tags into React element.

It's not compulsory to use JSX.

Rules of Writing JSX

- Html code must wrap into one top level element.
- Elements must be closed.
- Attribute class = className
- No if else condition inside jsx but ternary operator is okay.
- JS expression in JSX must be wrapped in {}.

Why JSX in React JS ?

```
function App() {
  return (
    React.createElement('form', {},
    React.createElement("h1", {}, "Login"),
    React.createElement('input', {type: 'text',placeholder:'Name', value: '',}),
    React.createElement('br', {}),React.createElement('br', {}),
    React.createElement('input', {type: 'password', placeholder: 'password',
                          value: '',}),
    React.createElement('br', {}), React.createElement('br', {}),
    React.createElement('button', {type: 'submit'}, "Login"))
   )
}
export default App;
```

```
function App() {
  return (<form><h2>Login</h2>
    <input type="text" placeholder="Name" /><br/><br/>
    <input type="password" placeholder="password" /> <br/><br/>
    <input type="submit" nvalue="log" />
</form>);
}
export default App;
```

Why JSX in React JS ?

- JSX is a special syntax introduced in ReactJS to write elements of components. It is syntactically identical to HTML and hence it can be easily read and written.

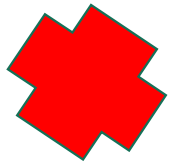
```
1. import React from 'react'
2.
3. function App () {
4.
5.     return <h1>Hello World</h1>
6.
7. }
8.
9. export default App;
```

- As the browser does not understand JSX code, this gets converted to JavaScript using the plugins of the babel

Why JSX in React JS ?

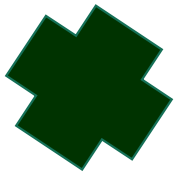
- When working with React components, you may need to render multiple React elements.

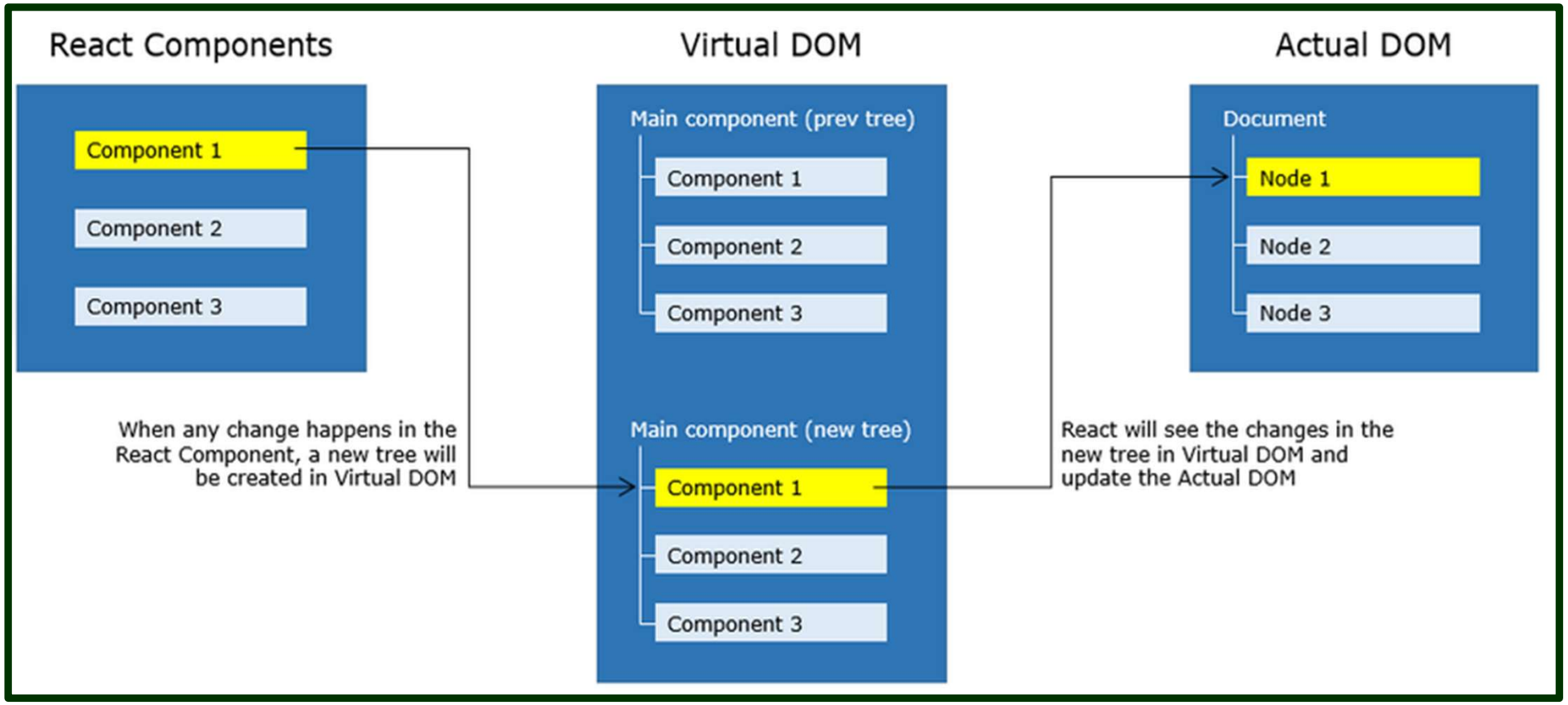
```
1. function App() {
2.     return
3.     <h3>ReactJS:</h3>
4.      <img src="./image/react.PNG" width="120" height="120"/>
5.     <p> React is a JavaScript library for creating User Interfaces.</p>
6. }
7. export default App;
```



- As per the JSX syntax, all the adjacent elements must be wrapped in an enclosing tag .

```
1. function App() {
2.   return (
3.     <div>
4.       <h3>ReactJS:</h3>
5.       <img src="./image/react.PNG" width="120" height="120" />
6.       <p> React is a JavaScript library for creating User Interfaces.</p>
7.     </div>
8.   );
9. }
10. export default App;
```




React Components
Virtual DOM
Actual DOM
Component 1
Component 2
Component 3
Main component (prev tree)
Component 1
Component 2
Component 3
Document
Node 1
Node 2
Node 3
When any change happens in the React Component, a new tree will be created in Virtual DOM
Main component (new tree)
Component 1
Component 2
Component 3
React will see the changes in the new tree in Virtual DOM and update the Actual DOM

React Component

Components are

independent and reusable

bits of code.

Functional Component	Class Component
• Functional Components is a plain JavaScript, you do not have a choice to set the state in functional component. • There is no render function we are using in functional components. • Functional components only accept the props as an argument. • Functional components are sometimes called stateless components.	• Class components we have a feature to set the set state in component. • In class components, we have a render function which is use to return the react elements. • In class components, we have both options use the props and set the state also. • Class components are sometimes called stateful components.

React Component

In ReactJS, we have mainly two types of components.

1. Functional Components **2. Class Components**

- Created as a simple JavaScript function
- It just returns the HTML elements

Functional Components:

```
import React from 'react';
function App()
{
   return React.createElement("h1", {}, "Hello World!!!");
}
export default App;
```

React Component

App - The component name should be in **Pascal Casing**.

React.createElement - helps to create an element in plain JavaScript.

In the code, React.createElement("h1", {}, "Hello World!!!");

h1 - is the HTML element to be used

{ } - Attributes of an element can be mentioned

"Hello World!!!" - Content to be appended to the h1 element

export default App - App component must be exported to use.

React Component

To display elements of a component, the component must be rendered.

Syntax: ReactDOM.render(<parameter 1/>, parameter 2);

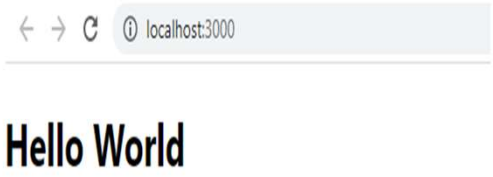
parameter 1' is the **name of the component to be rendered**.

'parameter 2' is the **where the component to be rendered**.

React Component

```
1. import React from 'react';
2.
3. function App() {
4.     return React.createElement("h1", {}, "Hello World!!!");
5. }
6. export default App;
```

```
1. import React from 'react';
2. import ReactDOM from 'react-dom';
3. import './index.css';
4. import App from './App';
5.
6. ReactDOM.render(<App/>, document.getElementById('root'));
7. reportWebVitals();
```



React Class Component

- A **class component** must include the **extends React.Component** statement.
- **The component also requires a render() method, this method returns HTML.**

```
class Car extends React.Component
 {
        render()
                { return <h2>Hi, I am a Car!</h2>; }
}
```

Render the content based on conditions

Using if-else

```
function App() {
  let element = null;
  let isLoggedIn = false;
  if (isLoggedIn) {
    element = <h2>Welcome Admin</h2>;
  } else {
    element = <h2>Please Login</h2>;
  }
  return <>{element}</>;
}
export default App;
```

Render the content based on conditions

Using ternary operator:

```
function App() {

  let isLoggedIn = false;

  return isLoggedIn ? <h2>Welcome Admin</h2> : <h2>Please Login</h2>;

}
```

Render the content based on conditions

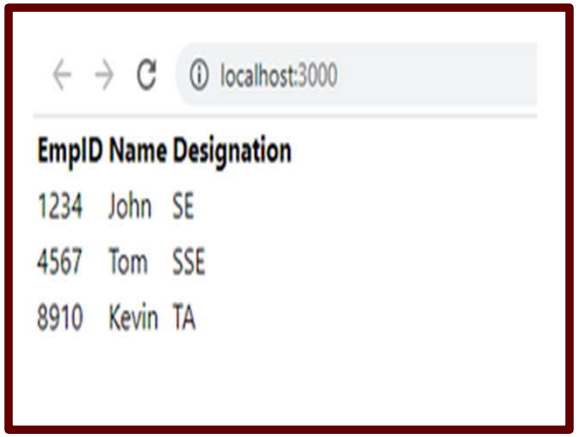
App.js

```
function App() {
  var employees = [
    { empId: 1234, name: "John", designation: "SE" },
    { empId: 4567, name: "Tom", designation: "SSE" },
    { empId: 8910, name: "Kevin", designation: "TA" },
  ];
  return (
    <>
      <table>
        <thead>
          <tr>
            <th>EmpID</th>
            <th>Name</th>
            <th>Designation</th>
          </tr>
        </thead>
        <tbody>
          {employees.map((employee) => {
            return (
              <tr key={employee.empId}>
                <td>{employee.empId}</td>
                <td>{employee.name}</td>
                <td>{employee.designation}</td>
              </tr>
            );
          })}
        </tbody>
      </table>
    </>
  );
}
export default App;
```

Render the content based on conditions

Index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';


ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);


reportWebVitals();
```



React Props

Props are arguments passed into React components.

Props are passed to components via HTML attributes.

Example

Add a "brand" attribute to the Car element:

```
const myElement = <Car brand="Ford" />;
```

Example

Use the brand attribute in the component:

```
function Car(props)
{
    return <h2>I am a { props.brand }!</h2>;
}
```

React Props

```
import React from 'react';

import ReactDOM from 'react-dom';

function Car(props)

{

  return <h2>I am a { props.brand }!</h2>;

}

const myElement = <Car brand="Ford" />;

const root = ReactDOM.createRoot(document.getElementById('root'));

root.render(myElement);
```

React Props

Let's recap what happens in this example:

- **We call root.render() with the <Car brand="Ford" /> element.**
- **React calls the Car component with {brand: 'Ford'} as the props.**
- **Our Car component returns a <h2>I am a, Ford</h2>. element as the result.**
- **React DOM efficiently updates the DOM to match <h2>I am a, Ford</h2>.**

React Props

- Props are also how you pass data from one component to another, as parameters.

```
import React from 'react';
import ReactDOM from 'react-dom/client';
function Car(props)
{          return <h2>I am a { props.brand }!</h2>;     }
function Garage() {
  return (
    <>          <h1>Who lives in my garage?</h1>              <Car brand="Ford" />    </>
  );
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Garage />);
```

Output:

Who lives in my Garage?

I am a Ford!

React Props

- Create a variable named **carName** and send it to the **Car component**

```
import React from 'react';
import ReactDOM from 'react-dom/client';

function Car(props) {
  return <h2>I am a { props.brand }!</h2>;
}

function Garage() {
  const carName = "Ford";
  return (
    <>
      <h1>Who lives in my garage?</h1>
      <Car brand={ carName } />
    </>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Garage />);
```

React Props

- Create an **object named carInfo** and send it to the **Car component**:

```
import React from 'react';
import ReactDOM from 'react-dom/client';

function Car(props) {
  return <h2>I am a { props.brand.model }!</h2>;
}

function Garage() {
  const carInfo = { name: "Ford", model: "Mustang" };
  return (
    <>
      <h1>Who lives in my garage?</h1>
      <Car brand={ carInfo } />
    </>
  );
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Garage />);
```

React Default Props

App.js:

```
function App(props) {

  return(<>

            <h1> {props.element1} </h1>

            <h1> {props.element2} </h1>

        </>);


}

App.defaultProps = {

    element1 : "Hello",

    element2 : "World",

}

export default App;
```

index.js

```
import React from 'react';

import ReactDOM from 'react-dom';

import './index.css';

import App from './App';

import reportWebVitals from './reportWebVitals';


ReactDOM.render(

  <React.StrictMode>

    <App />

  </React.StrictMode>,

  document.getElementById('root')

);

reportWebVitals();
```

Output:

ReactJS Fragments

- when we are trying to render more than one root element we have to put the entire content inside the ‘div’ tag which is not loved by many developers.
- So in React 16.2 version, **Fragments** were introduced, and we use them instead of the extraneous ‘div’ tag.

```
<React.Fragment>
    <h2>Child-1</h2>
    <p> Child-2</p>
</React.Fragment>
```

```
<>
    <h2>Child-1</h2>
    <p> Child-2</p>
</>
```

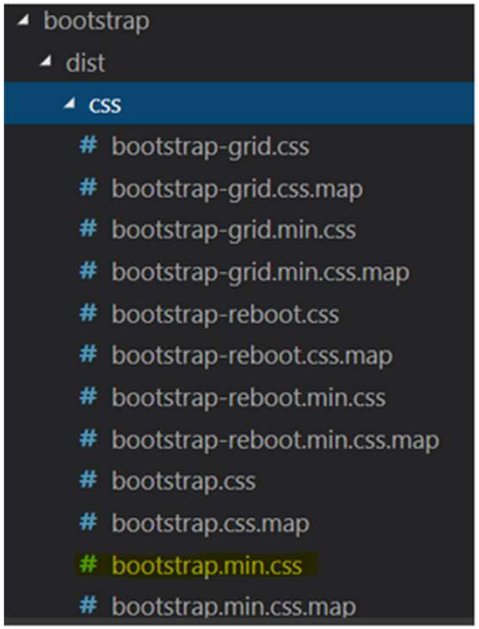
Styling Components

- To style components using the bootstrap library, we need to install the bootstrap library.

npm install bootstrap

npm install react-bootstrap

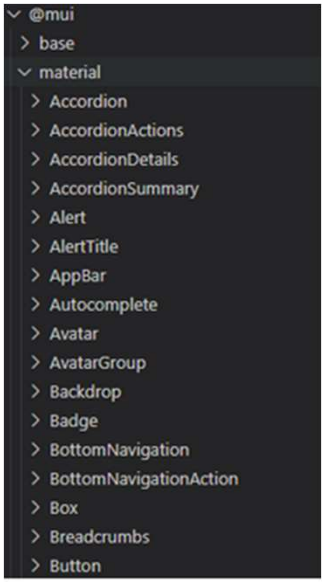
```
import 'bootstrap/dist/css/bootstrap.min.css';

<button className="btn btn-success">Submit</button>
```



Styling Components

- Material-UI is a popular React UI framework which provides various components and themes for styling React components.

npm install @mui/material @emotion/react @emotion/styled



Styling Components

Demo 1: Styling Components

App.js

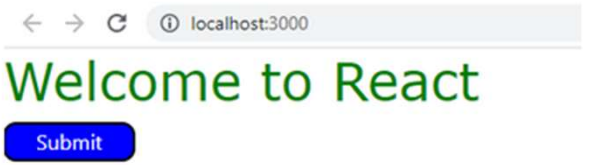
```
import "bootstrap/dist/css/bootstrap.min.css";

function App() {

  return (

    <>

      <h1>Welcome to React</h1>

      <button className="btn btn-success">Submit</button>

    </>

  );

}

export default App;
```

Index.js

```
import React from 'react';

import ReactDOM from 'react-dom';

import './index.css';

import App from './App';

import reportWebVitals from './reportWebVitals';

ReactDOM.render(

  <React.StrictMode>

    <App />

  </React.StrictMode>,

  document.getElementById('root')

);

reportWebVitals();
```

Styling Components

Demo 2: Styling Components



App.js

```
import './App.css'
function App() {
  return (
    <>
      <h1 style={{ color: "green", fontFamily: "verdana" }}>
        Welcome to React
      </h1>
      <button className="button">Submit</button>
    </>
  );
}
export default App;
```

Index.js

```
import React from 'react';
import ReactDOM from 'react-dom';
import './index.css';
import App from './App';
import reportWebVitals from './reportWebVitals';
ReactDOM.render(
  <React.StrictMode>
    <App />
  </React.StrictMode>,
  document.getElementById('root')
);
reportWebVitals();
```

App.css

```
.button {
  background-color:blue;
  color:white;
  border-radius:10px;
  width:100px;
  height:30px;
}
```

React Component

Props	State
Props are read-only.	State changes can be asynchronous.
Props are immutable.	State is mutable.
Props allow you to pass data from one component to other components as an argument.	State holds information about the components.
Props can be accessed by the child component.	State cannot be accessed by child components.
Props are used to communicate between components.	States can be used for rendering dynamic changes with the component.
Props make components reusable.	State cannot make components reusable.

React State

The React useState Hook allows us to track state in a function component.

State generally refers to **data or properties** that need to be tracking in an application.

Import useState

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

Initialize useState

We initialize our state by calling **useState** in our function component.

useState accepts an initial state and returns two values:

- The current state.
- A function that updates the state.

Example:

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

function Counter() {
  // Declare a state variable 'count' with an initial value of 0
  const [count, setCount] = useState(0);

  // Function to increment the count
  const increment = () => {
    setCount(count + 1);
  };

  // Function to decrement the count
  const decrement = () => {
    setCount(count - 1);
  };

  return (
    <div>
      <h1>Counter: {count}</h1>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
    </div>
  );
}
```

Output:
When you click the "Increment" button, the counter increases:

```
Counter: 1
```

When you click the "Decrement" button, the counter decreases:

```
Counter: 0
```

React Lifecycle Methods

- In ReactJS, every component creation process involves various lifecycle methods. These lifecycle methods are termed as component's lifecycle.

The lifecycle of the component is divided into four phases.

Mounting (adding nodes to the DOM)

Updating (altering existing nodes in the DOM)

Unmounting (removing nodes from the DOM)

Error handling (verifying that your code works and is bug-free)

React Lifecycle Methods- Mounting phase

Mounting

Mounting means putting elements into the DOM.

React has four built-in methods .

constructor()

getDerivedStateFromProps()

render()

componentDidMount()

React Lifecycle Methods

constructor

The constructor() method is called before anything else, when the component is initiated.

getDerivedStateFromProps

The getDerivedStateFromProps() method is called right before rendering the element(s) in the DOM.

render()

The render() method is required, and is the method that actually outputs the HTML to the DOM.

The componentDidMount() method is called after the component is rendered. This is where you run statements that requires that the component is already placed in the DOM.

React Lifecycle Methods

getDerivedStateFromProps

```
import React from 'react';
import ReactDOM from 'react-dom/client';

class Header extends React.Component {
  constructor(props) {
    super(props);
    this.state = {favoritecolor: "red"};
  }
  static getDerivedStateFromProps(props, state) {
    return {favoritecolor: props.favcol };
  }
  render() {
    return (
      <h1>My Favorite Color is {this.state.favoritecolor}</h1>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Header favcol="yellow"/>);
```

React Lifecycle Methods-Update Phase

React has five built-in methods that gets called, in this order, when a component is updated:

getDerivedStateFromProps()

shouldComponentUpdate()

render()

getSnapshotBeforeUpdate()

componentDidUpdate()

React Lifecycle Methods

- **getDerivedStateFromProps** is the first method that is called when a component gets updated.
- In the **shouldComponentUpdate()** method you can return a Boolean value that specifies whether React should continue with the rendering or not.
- **The default value is true.**
- The **render()** method is of course called when a component gets *updated*, it has to re-render the HTML to the DOM, with the new changes.

React Lifecycle Methods

- In the **getSnapshotBeforeUpdate()** method you have access to the props and state *before* the update, meaning that even **after the update, you can check what the values were *before* the update**.
- The **componentDidUpdate** method is called after the component is updated in the DOM.

React Lifecycle Methods

React Events

- Handling events with React elements is very similar to handling events on DOM elements.

1. React events are named using **camelCase,** rather than **lowercase** — ***onClick* instead of *onclick*.**

2. We write React event handlers inside **curly braces**.

In the case of React(/JSX), rather than passing a string, we pass a function as the event handler. *onClick={buttonClicked}* instead of *onclick="buttonClicked()"*.

React Events

In HTML

```
<button onclick=”buttonClicked()”>
 Click the Button!
</button>
```

In React JS

```
<button onClick={buttonClicked}>
 Click the Button!
</button>
```

React Events

- Another difference is that in order to prevent default behavior to React, we cannot return false. We have to call *preventDefault* explicitly.

In HTML:

```
// HTML
<button onclick="console.log('Button clicked.'); return false">
 Click Me
</button>
```

In React:

```
// React
function handleClick(e) {
    e.preventDefault();
    console.log('Button clicked.');
}

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

Handling onClick Event in the Functional Component

```
import React from 'react'

function FunctionClick()
{
    function clickHandler()
        {
        console.log('Button clicked')
        }
      return
      (
        <div>
          <button onClick={clickHandler}>Click</button>
        </div>
      )
}

export default FunctionClick;
```

React Events

```
import React, { Component } from 'react';
class App extends React.Component {
    constructor(props) {
        super(props);
        this.state = {
            companyName: ''
        };
    }
    changeText(event) {
        this.setState({
            companyName: event.target.value
        });
    }
    render() {
        return (
            <div>
                <h2>Simple Event Example</h2>
                <label>Enter company name: </label>
                <input type="text" id="companyName" onChange={this.changeText.bind(this)}/>
                <h4>You entered: { this.state.companyName }</h4>
            </div>
        );
    }
}  export default App;
```

React Events Summary List

Clipboard

- onCopy
- onCut
- onPaste

Keyboard

- onKeyDown
- onKeyPress
- onKeyUp

Selection

- onSelect

UI

- onScroll

Mouse

- onMouseDown
- onMouseEnter
- onMouseLeave
- onMouseMove
- onMouseOut
- onMouseOver
- onMouseUp

Animation

- onAnimationStart
- onAnimationEnd
- onAnimationIteration

Image

- onLoad
- onError

Why We Need React Router?

Multi-Page Application :

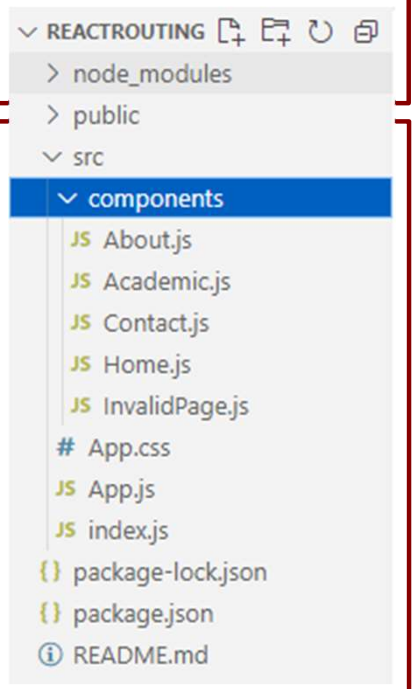
- Every request will be sent to the server from the client.
- The Server responds to the client with new HTML content.
- Every time page reload will happen for every request.
- This would increase the round trips to the server and cause delay in response.

Why We Need React Router?

In single page applications :

- In MPA After the initial page load, no server communication is required for further page updates upon user interaction .
- So here, we have to navigate from one view to another without hitting server.
- **For this React JS provides the react-router-dom library.**

React Router



Step 1 :

npm install react-router-dom

Step 2:

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

React Router

<BrowserRouter>: It is a primary component containing the entire routing configuration. It is a more popular one because it uses the HTML5 History API to keep track of your router history.

<Route>: It is used to define and render component based on the specified path.

path: Maps to the path given in URL

element(component): Contains component name to be rendered when the route is mapped.

exact: This property tells Route to match the exact path.(Removed in V6)

Key Components in React Router:

1. `BrowserRouter`:
   - It keeps the UI in sync with the URL using the HTML5 history API (pushState, replaceState, and popstate event).
2. `Routes`:
   - It replaces the older `<Switch>` component in React Router v6 and is used to render the first route that matches the current location.
3. `Route`:
   - Used to define individual routes. The `path` attribute defines the URL path, and the `element` attribute specifies the component to render.
4. `Link`:
   - A navigation component that allows users to navigate between different routes without a full page reload.

React Router

```
<Routes>
        <Route path = "/"  element={<Home/>}></Route>
        <Route path = "/about" element = {<About/>}></Route>
        <Route path = "/academic" element= {<Academic/>}></Route>
        <Route path = "/contact" element = {<Contact/>}></Route>
        <Route path = "*" element = {<InvalidPage/>}></Route>
</Routes>
```

React Router

<Link> vs <NavLink>

- The Link component allows navigating the different routes on the websites, whereas NavLink component is used to add styles to the active routes.

```
<li><NavLink to='/' activeClassName="active"> Home </NavLink></li>
<li><NavLink to='/about' activeClassName="active">About</NavLink></li>
<li><NavLink to='/academic' activeClassName="active">Academic</NavLink></li>
<li><NavLink to='/contact' activeClassName="active">Contact</NavLink></li>
```

React Router Switch (Removed in V6)

- The <**Switch**> component is used to render components only when the path will be **matched**. Otherwise, it returns to the **not found** component.

App.js

```jsx
import React from 'react';
import { BrowserRouter as Router, Route, Routes, Link } from 'react-router-dom';
import Home from './Home';
import About from './About';
import Contact from './Contact';

function App() {
  return (
    <Router>
      <div>
        <nav>
          <ul>
            <li>
              <Link to="/">Home</Link>
            </li>
            <li>
              <Link to="/about">About</Link>
            </li>
            <li>
              <Link to="/contact">Contact</Link>
            </li>
          </ul>
        </nav>

        <Routes>
          <Route path="/" element={<Home />} />
          <Route path="/about" element={<About />} />
          <Route path="/contact" element={<Contact />} />
        </Routes>
      </div>
    </Router>
  );
}

export default App;
```

Home.js

```
import React from 'react';
function Home()
{
return <h2>Welcome to the Home Page</h2>;
}
export default Home;
```

About.js

```
import React from 'react';
function About()
{
return <h2>This is the About Page</h2>;
}
export default About;
```

Contact.js

```
import React from 'react';
function Contact()
{
return <h2>Contact Us!</h2>;
}
export default Contact;
```

React Form

- Forms are an integral part of any modern web application. It allows the users to interact with the application as well as gather information from the users.
- **In HTML**, form elements such as **<input>, <textarea>, and <select>** typically maintain their own state and update it based on user input.
- **In React**, state of these input elements is typically kept in the state property of components and only updated with setState().

React Form

There are mainly two types of form input in React.

- Uncontrolled component
- Controlled component

In React, the form is usually implemented by using controlled components.

React Form

Uncontrolled component

- The uncontrolled input is similar to the traditional HTML form inputs.
- The DOM itself handles the form data.
- Here, the HTML elements maintain their own state that will be updated when the input value changes.
- To write an uncontrolled component, you need to use a **ref** to get form values from the DOM.

React Form

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

function App() {
const inputRef = useRef(null);

function handleSubmit() {
     alert(`Name: ${inputRef.current.value}`);
}

return (
     <div className="App">
     <h3>Uncontrolled Component</h3>
     <form onSubmit={handleSubmit}>
          <label>Name :</label>
          <input type="text" name="name" ref={inputRef} />
          <button type="submit">Submit</button>
     </form>
     </div>
);
}

export default App;
```

React Form

Controlled Components:

- In React, Controlled Components are those in which form's data is handled by the component's state.
- It takes its current value through props and makes changes through callbacks like onClick, onChange, etc.

Controlled	Uncontrolled
It does not maintain its internal state.	It maintains its internal states.
Here, data is controlled by the parent component.	Here, data is controlled by the DOM itself.
It accepts its current value as a prop.	It uses a ref for their current values.
It allows validation control.	It does not allow validation control.
It has better control over the form elements and data.	It has limited control over the form elements and data.

React Tables

- A table is an arrangement which organizes information into rows and columns. It is used to store and display data in a structured format.
- **Let us create a React app using the following command.**

```
$ npx create-react-app myreactapp

$ npm install react-table-6

import ReactTable from "react-table-6";
```

React Tables

```
import React, { Component } from 'react';
import './App.css';

function App() {
  return (
    <div className="App">
      <table>
        <tr>
          <th>Name</th>
          <th>Age</th>
          <th>Gender</th>
        </tr>
        <tr>
          <td>Anom</td>
          <td>19</td>
          <td>Male</td>
        </tr>
        <tr>
          <td>Megha</td>
          <td>19</td>
          <td>Female</td>
        </tr>
        <tr>
          <td>Subham</td>
          <td>25</td>
          <td>Male</td>
        </tr>
      </table>
    </div>
  );
}
export default App;
```

Name	Age	Gender
Anom	19	Male
Megha	19	Female
Subham	25	Male

React Tables

```
import './App.css';
import React, { Component } from 'react';

const data = [
{ name: "Anom", age: 19, gender: "Male" },
{ name: "Megha", age: 19, gender: "Female" },
{ name: "Subham", age: 25, gender: "Male"},
]

function App() {
return (
  <div className="App">
  <table>
    <tr>
    <th>Name</th>
    <th>Age</th>
    <th>Gender</th>
    </tr>
    {data.map((val, key) => {
    return (
      <tr key={key}>
      <td>{val.name}</td>
      <td>{val.age}</td>
      <td>{val.gender}</td>
      </tr>
    )
    })}
  </table>
  </div>
);
} export default App;
```

```
.App {
  width: 100%;
  height: 100vh;
  display: flex;
  justify-content: center;
  align-items: center;
}

table {
  border: 2px solid rgb(4, 29, 4);
  width: 800px;
  height: 200px;
}

th {
  border-bottom: 1px solid rgb(4, 29, 4);
}

td {
  text-align: center;
}
```

React Tables

react-table-6:

```
import ReactTable from "react-table";

import "react-table/react-table.css";
```

- It is lightweight .
- It is fully customizable
- It has filters.
- Minimal design & easily themeable

React Tables

Let us assume we have data which needs to be rendered using react-table.

```
const data =
        [
            {
            name: 'Virat',
            age: 30
            },
            {
            name: 'Rohit',
            age: 32
            },
            {
            name: 'Dhoni',
            age: 37
            }
        ]
```

```
const columns =
        [
            {
            Header: 'Name',
            accessor: 'name'
            },
            {
             Header: 'Age',
             accessor: 'age'
            }
      ]
```

React Tables

Inside the render method, we need to bind this data with react-table and then returns the react-table.

```
return (
      <div>
         <ReactTable
              data={data}
              columns={columns}
              defaultPageSize = {2}
              pageSizeOptions = {[2,4, 6]}
          />
      </div>
)
```

React Tables

Name	Age
Ayaan	26
Ahana	22

Previous Page 1 of 3 2 rows Next

Name	Age
Ayaan	26
Ahana	22
Peter	40
Virat	30
Rohit	32
Dhoni	37

Previous Page 1 of 1 6 rows Next

React Hooks

- Hooks were added to **React in version 16.8.**
- Hooks allow function components to have access to state and other React features.
- It allows you to use state and other React features without writing a class.
- **Because of this, class components are generally no longer needed.**
- Hooks allow us to "hook" into React features such as state and lifecycle methods.

React Hooks

Rules of Hooks

Only call Hooks at the top level

Do not call Hooks inside loops, conditions, or nested functions. Hooks should always be used at the top level of the React functions.

Only call Hooks from React functions

You cannot call Hooks from regular JavaScript functions. Instead, you can call Hooks from React function components.

React Hooks

The React useState Hook allows us to track state in a function component.

To use the useState Hook, we first need to import it into our component.

import { useState } from "react";

Initialize useState

useState accepts an initial state and returns two values:

The current state.

A function that updates the state.

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

function FavoriteColor() {
  const [color, setColor] = useState("");
}
```

React Hook State

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

class CountApp extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }
  render() {
    return (
      <div>
        <p><b>You clicked {this.state.count} times</b></p>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Click me
        </button>
      </div>
    );
  }
}
export default CountApp;
```

React Hook State

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

function CountApp() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
export default CountApp;
```

React State (Read)

```
import { useState } from "react";
import ReactDOM from "react-dom/client";
function FavoriteColor()
{
  const [color, setColor] = useState("red");
  return <h1>My favorite color is {color}!</h1>
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<FavoriteColor />);
```

React State (Update)

```
import { useState } from "react";
import ReactDOM from "react-dom/client";
function FavoriteColor() {
 const [color, setColor] = useState("red");
 return (
  <>
   <h1>My favorite color is {color}!</h1>
   <button
    type="button"
    onClick={() => setColor("blue")}>Blue</button>
  </>
 )
}
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<FavoriteColor />);
```

React Component

```jsx
import { useState } from "react";
import ReactDOM from "react-dom/client";


function Car() {
  const [brand, setBrand] = useState("Ford");
  const [model, setModel] = useState("Mustang");
  const [year, setYear] = useState("1964");
  const [color, setColor] = useState("red");

  return (
    <>
      <h1>My {brand}</h1>
      <p>
        It is a {color} {model} from {year}.
      </p>
    </>
  )
}


const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Car />);
```

```jsx
import { useState } from "react";
import ReactDOM from "react-dom/client";

function Car() {
  const [car, setCar] = useState({
    brand: "Ford",
    model: "Mustang",
    year: "1964",
    color: "red"
  });

  return (
    <>
      <h1>My {car.brand}</h1>
      <p>
        It is a {car.color} {car.model} from {car.year}.
      </p>
    </>
  )
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Car />);
```

React useEffect Hook

- The **Effect Hook allows us to perform side effects (an action)** in the function components.

  Example: Fetching data, directly updating the DOM, and timers.

- It does not use components lifecycle methods which are available in class components.
- In other words, Effects Hooks are equivalent to componentDidMount(), componentDidUpdate(), and componentWillUnmount() lifecycle methods.

React useEffect Hook

useEffect accepts two arguments.

The second argument is optional.

useEffect(<function>, <dependency>)

I've rendered 55 times!

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


function Timer() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    setTimeout(() => {
      setCount((count) => count + 1);
    }, 1000);
  });

  return <h1>I've rendered {count} times!</h1>;
}

export default Timer;
```

React useffect Hook

No dependency passed:

```
useEffect(() => {
  //Runs on every render
});
```

An empty array:

```
useEffect(() => {
  //Runs only on the first render
}, []);
```

Props or state values:

```
useEffect(() => {
  //Runs on the first render   //And any time any dependency value changes
}, [prop, state]);
```

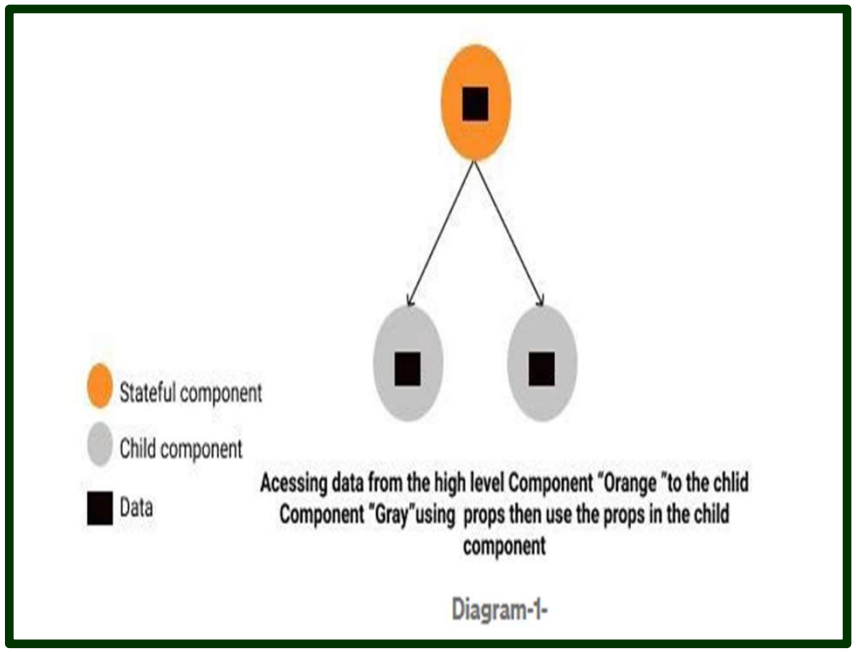
React useContext Hook

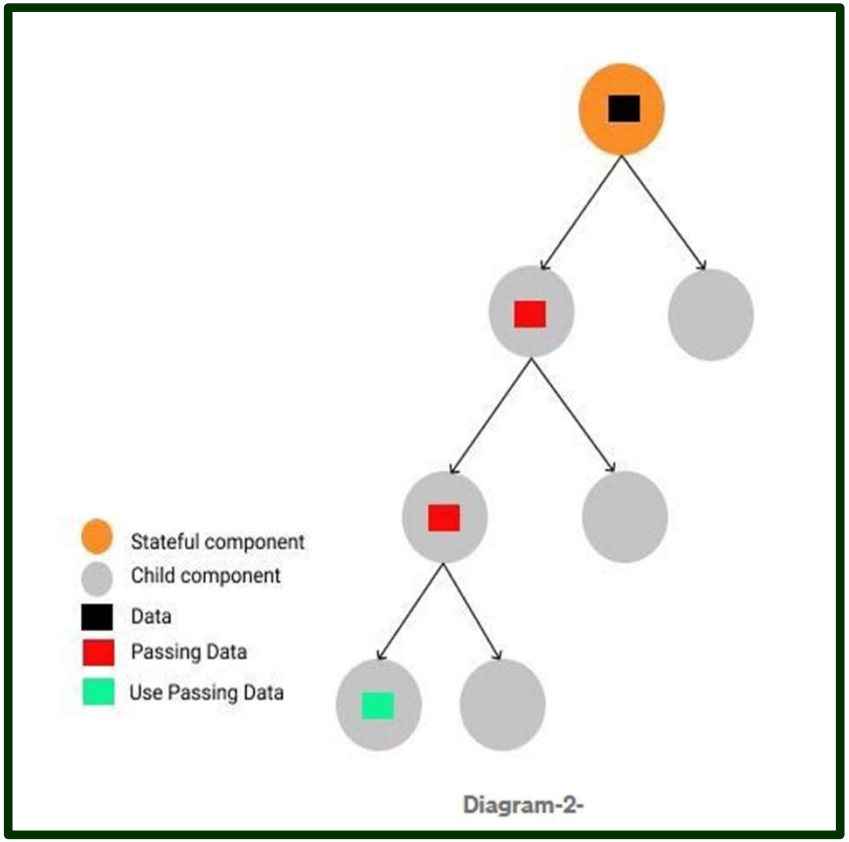
- **React Context is a way to manage state globally**.
- It can be **used together with the useState Hook to share state between deeply nested components** more easily than with useState alone.
- **Prop drilling :**

  Prop drilling refers to **transporting this data/state as props to the intended destination** through intermediate components.

React useContext Hook

Prop drilling:





React useContext Hook

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

function Component1() {
  const [user, setUser] = useState("CBIT-IT3");

  return (
    <>
      <h1>{`Hello ${user}!`}</h1>
      <Component2 user={user} />
    </>
  );
}
function Component2({ user }) {
  return (
    <>
      <h1>Component 2</h1>
      <Component3 user={user} />
    </>
  );
}
function Component3({ user }) {
  return (
    <>
      <h1>Component 3</h1>
      <h2>{`Hello ${user} again!`}</h2>
    </>
  );
}
export default Component1;
```

Hello CBIT-IT3!

Component 2

Component 3

Hello CBIT-IT3 again!

React useContext Hook

Create Context

To create context, you must Import createContext and initialize it:

```
import { useState, createContext } from "react";
import ReactDOM from "react-dom/client";

const UserContext = createContext()
```

Context Provider

Wrap child components in the Context Provider and supply the state value.

```
function Component1() {
    const [user, setUser] = useState("CBIT-IT3");

    return (
      <UserContext.Provider value={user}>
        <h1>{`Hello ${user}!`}</h1>
        <Component2 />
      </UserContext.Provider>
    );
  }
```

React useContext Hook

Use the useContext Hook

- In order to use the Context in a child component, we need to access it using the **useContext** Hook.

```
import { useState, createContext, useContext } from "react";
function Component3() {
  const user = useContext(UserContext);

  return (
    <>
      <h1>Component 3</h1>
      <h2>{`Hello ${user} again!`}</h2>
    </>
  );
}
```

React useContext Hook

```
import React,{ useState } from "react";
import { useState, createContext, useContext } from "react";

const UserContext = createContext();

function Component1() {
 const [user, setUser] = useState("CBIT-IT3");
 return (
  <UserContext.Provider value={user}>
   <h1>{`Hello ${user}!`}</h1>
   <Component2 />
  </UserContext.Provider>
 );
}
function Component2({ user }) {
 return (
  <>
   <h1>Component 2</h1>
   <Component3 user={user} />
  </>
 );
}
function Component3() {
 const user = useContext(UserContext);
 return (
  <>
   <h1>Component 3</h1>
   <h2>{`Hello ${user} again!`}</h2>
  </>
 );
} export default Component1;
```

Hello CBIT-IT3!

Component 2

Component 3

Hello CBIT-IT3 again!

React Custom Hooks

- **Hooks are reusable functions.**
- When you have component logic that needs to be used by multiple components, we can extract that logic to a custom Hook.
- The main reason for which you should be using **Custom hooks is to maintain the concept of DRY(Don't Repeat Yourself) in your React apps**.
- **Custom Hooks start with "use". Example: useFetchCBIT.**

React Custom Hooks

```
import { useState, useEffect } from "react";
import ReactDOM from "react-dom/client";

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

  useEffect(() => {
    fetch("https://jsonplaceholder.typicode.com/todos")
      .then((res) => res.json())
      .then((data) => setData(data));
 }, []);

  return (
    <>
      {data &&
        data.map((item) => {
          return <p key={item.id}>{item.title}</p>;
        })}
    </>
  );
};
```

delectus aut autem

quis ut nam facilis et officia qui

fugiat veniam minus

et porro tempora

laboriosam mollitia et enim quasi adipisci quia provident illum

qui ullam ratione quibusdam voluptatem quia omnis

illo expedita consequatur quia in

quo adipisci enim quam ut ab

molestiae perspiciatis ipsa

illo est ratione doloremque quia maiores aut

vero rerum temporibus dolor

ipsa repellendus fugit nisi

et doloremque nulla

repellendus sunt dolores architecto voluptatum

ab voluptatum amet voluptas

accusamus eos facilis sint et aut voluptatem

React Custom Hooks

```jsx
import React,{ useState, useEffect } from "react";
import useFetchCBIT from "./useFetchCBIT";

const Home = () => {
  const [data] = useFetchCBIT("https://jsonplaceholder.typicode.com/todos");

  return (
    <>
      {data &&
        data.map((item) => {
          return <p key={item.id}>{item.title}</p>;
        })}
    </>
  );
};

export default Home;
```

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

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

  useEffect(() => {
    fetch(url)
      .then((res) => res.json())
      .then((data) => setData(data));
  }, [url]);

  return [data];
};

export default useFetchCBIT;
```

React Portals ★

- Portals provide a first-class way to render children into a DOM node that exists outside the DOM hierarchy of the parent component.
- So far we were having one DOM element in the HTML into which we were mounting our react application, i.e., the root element of our **index.html** in the public folder.
- If you take a look at the browser in the DOM tree **every single React component in our application falls under the root element**, i.e., inside this statement.

** **

React Portals

The **React 16.0** version introduced React portals in **September 2017**.

ReactDOM.createPortal(child, container)

Index.html

```
<body>
  <noscript>You need to enable JavaScript to run this app.</noscript>
  <div id="root"></div>
  <div id="cbit-root"></div>
  <!--
    This HTML file is a template.
    If you open it directly in the browser, you will see an empty page.

    You can add webfonts, meta tags, or analytics to this file.
    The build step will place the bundled scripts into the <body> tag.

    To begin the development, run `npm start` or `yarn start`.
    To create a production bundle, use `npm run build` or `yarn build`.
  -->
</body>
```

App.js

```
import React from 'react'
import Portaldemo from './Portaldemo'

function App() {
  return (
    <>
    <div>Welcome to CBIT</div>
    <Portaldemo />
    </>
  )
}

export default App
```

Portaldemo.js

```
import React from 'react'
import ReactDOM  from 'react-dom'


function Portaldemo() {
  return (
    <h1>Welcome to IT3</h1>

  )
}
export default Portaldemo
```

React Portals

```
<!DOCTYPE html>
<html lang="en"> event
▶ <head> ··· </head>
▼ <body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
  ▼ <div id="root"> event
      <div>Welcome to CBIT</div>
      <h1>Welcome to IT3</h1>
    </div>
    <div id="cbit-root"></div>
    <!--
    This HTML file is a template. If you open it directly in the browser, you will see an empty page. You can add webfonts, meta tags, or analytics
    to this file. The build step will place the bundled scripts into the <body> tag. To begin the development, run `npm start` or `yarn start`. To
    create a production bundle, use `npm run build` or `yarn build`.
    -->
  </body>
</html>
```

React Portals

Index.html

```
<body>
  <noscript>You need to enable JavaScript to run this app.</noscript>
  <div id="root"></div>
  <div id="cbit-root"></div>
  <!--
    This HTML file is a template.
    If you open it directly in the browser, you will see an empty page.

    You can add webfonts, meta tags, or analytics to this file.
    The build step will place the bundled scripts into the <body> tag.

    To begin the development, run `npm start` or `yarn start`.
    To create a production bundle, use `npm run build` or `yarn build`.
  -->
</body>
```

App.js

```
import React from 'react'
import Portaldemo from './Portaldemo'

function App() {
  return (
    <>
    <div>Welcome to CBIT</div>
    <Portaldemo />
    </>
  )
}

export default App
```

Portaldemo.js

```
import React from 'react'
import ReactDOM  from 'react-dom'


function Portaldemo() {
  return ReactDOM.createPortal(
    <h1>Welcome to IT3</h1>,
    document.getElementById('cbit-root')
  )
}

export default Portaldemo
```

React Portals

```
<!DOCTYPE html>
<html lang="en"> event
▶ <head> ··· </head>
▼ <body>
    <noscript>You need to enable JavaScript to run this app.</noscript>
  ▼ <div id="root"> event
      <div>Welcome to CBIT</div>
    </div>
  ▼ <div id="cbit-root"> event
      <h1>Welcome to IT3</h1>
    </div>
    <!--
    This HTML file is a template. If you open it directly in the browser, you will see an empty page. You can add webfonts, meta tags, or analytics
    to this file. The build step will place the bundled scripts into the <body> tag. To begin the development, run `npm start` or `yarn start`. To
    create a production bundle, use `npm run build` or `yarn build`.
    -->
  </body>
</html>
```

React Portals

When to use?

- Modals
- Tooltips
- Floating menus
- Widgets
- **Event bubbling** is a method of event propagation in the HTML DOM API when an event is in an element inside another element, and both elements have registered a handle to that event. **if we fire an event from inside a portal it will propagate to the Parent component in the containing React tree.**

Back End Integration in React

- Axios is an HTTP client library that allows you to make requests to a given endpoint.

```
router.get('/', async(req,res) =>
{
    try
   {
            const aliens = await Alien.find()
            res.json(aliens)
    }catch(err)
   {
        res.send('Error ' + err)
    }
})
```

npm install axios

Why Use Axios in React

- It has good defaults to work with JSON data.
- Axios has function names that match any HTTP methods.
- Axios has better error handling.
- Axios does more with less code.
- Axios can be used on the server as well as the client.

Note : Complete Crud Operation Needs to be Written.

22ITC08

FULL STACK DEVELOPMENT

UNIT-III

Redux

What is Redux

- **Redux is a pattern and library for managing and updating global application state, where the UI triggers events called "actions" to describe what happened, and separate update logic called "reducers" updates the state in response.**
- It serves as a centralized store for state that needs to be used across your entire application, with rules ensuring that the state can only be updated in a predictable fashion.

Why Should I Use Redux?

- The patterns and tools provided by Redux make it easier to understand when, where, why, and how the state in your application is being updated, and how your application logic will behave when those changes occur.

When Should I Use Redux?

- You have large amounts of application state that are needed in many places in the app
- The app state is updated frequently over time
- The logic to update that state may be complex

Redux Libraries and Tools

- Redux is a small standalone JS library. However, it is commonly used with several other packages:

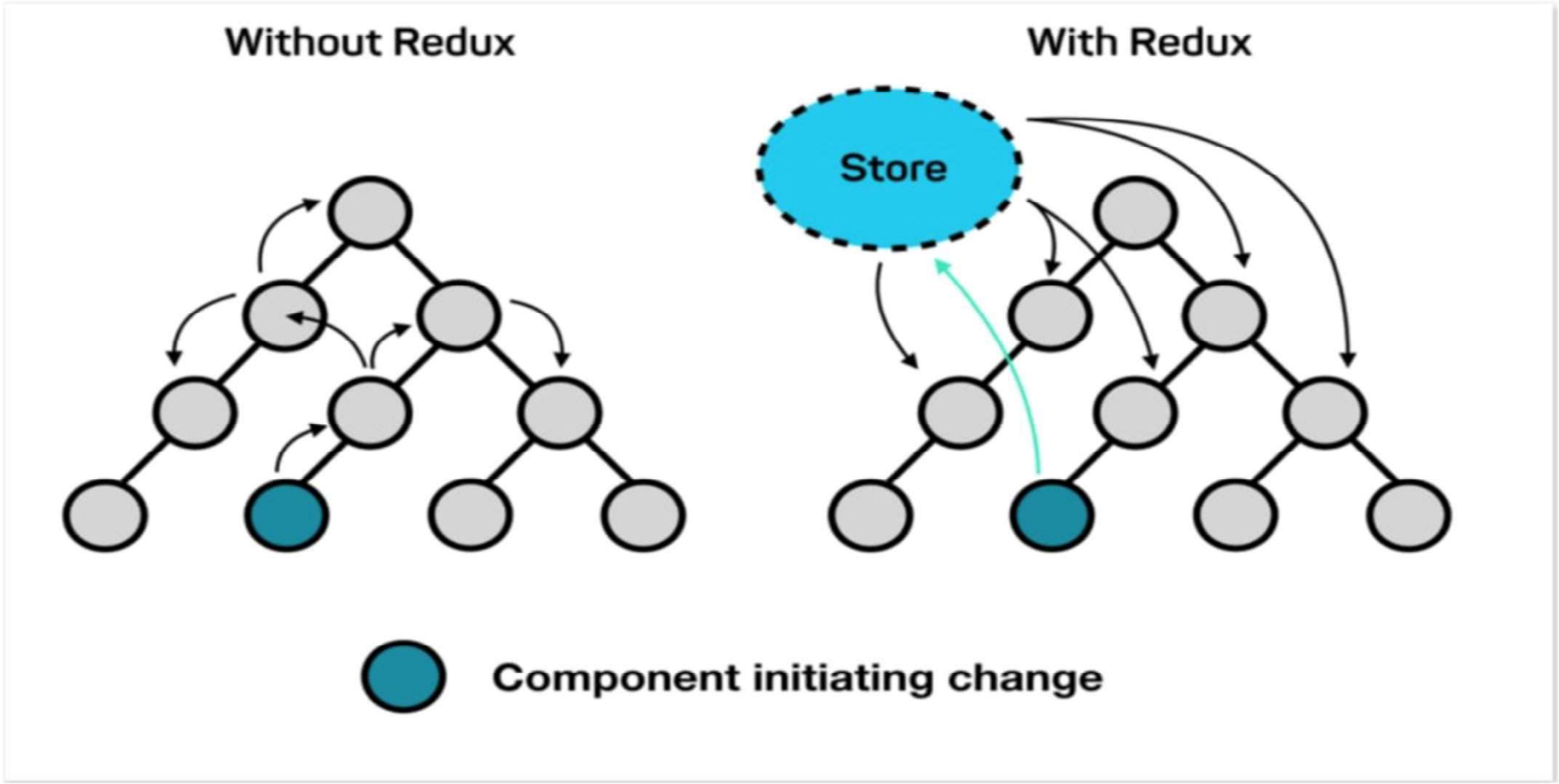
Redux Toolkit **npm install @reduxjs/toolkit react-redux**

- Redux Toolkit is our recommended approach for writing Redux logic.

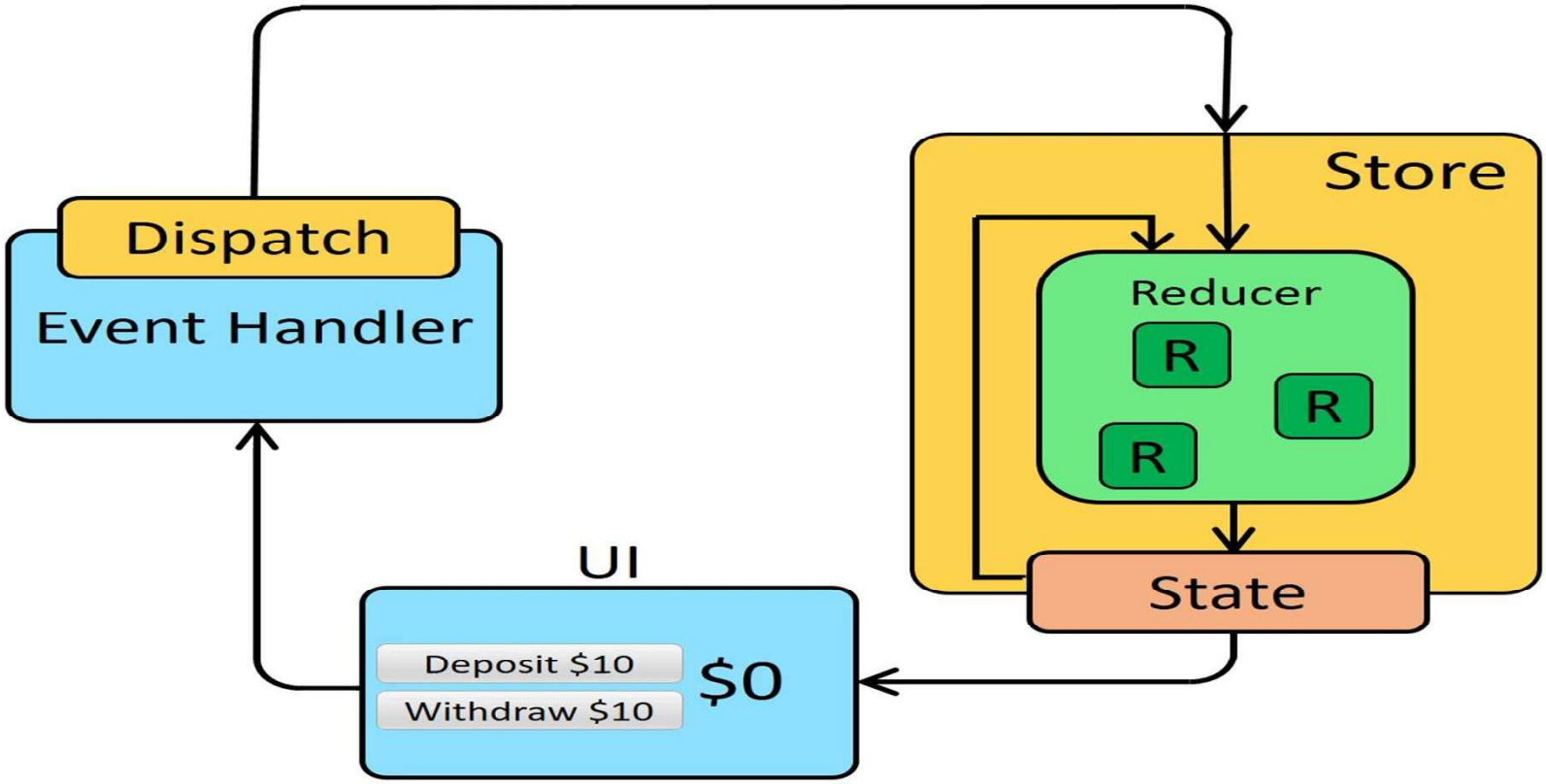
React-Redux

- Redux can integrate with any UI framework, and is most frequently used with React. React-Redux is our official package that lets your React components interact with a Redux store by reading pieces of state and dispatching actions to update the store.

Redux Dataflow



Redux Dataflow



Redux Dataflow

Redux Store:

- The global state of an application is stored in an object tree within a single store.
- The Redux store is the main, central bucket which stores all the states of an application.
- It should be considered and maintained as a **single source of truth** for the state of the application.

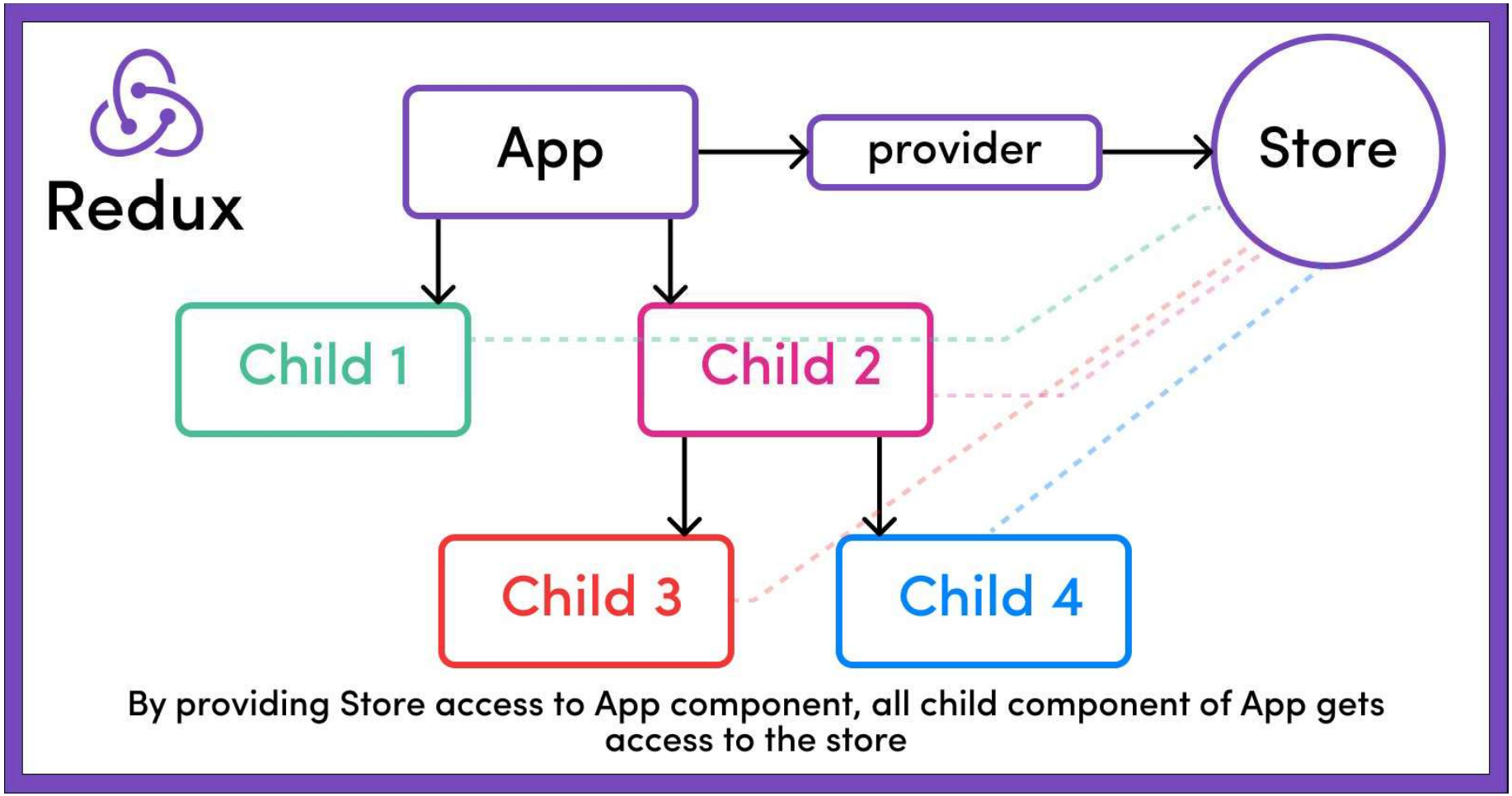
```
import { createStore } from 'redux';
import counterReducer from './counterReducer';

const store = createStore(counterReducer);

export default store;
```

Redux Dataflow

- If the store is provided to the App.js by wrapping the App component within the **<Provider> </Provider> tag** as shown in the code snippet below, then all children components of App.js can also access the state of the application from the store.
- **This makes it act as a global state.**

```
// src/index.js
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import store from './store';
import App from './App';

ReactDOM.render(
    <Provider store={store}>
        <App />
    </Provider>,
    document.getElementById('root')
);
```

Redux Dataflow



Redux Dataflow

Actions in Redux:

- **Actions are plain JavaScript objects that have a type field.** The only way to change the state is to emit an action, which is an object describing what happened**.**
- **An action as an event that describes something that happened in the application.**
- if anyone wants to change the state of the application, then they'll need to express their intention of doing so by **emitting or dispatching an action**.

```
// src/actions/index.js
import { INCREMENT, DECREMENT } from './actionTypes';

export const increment = () => ({
   type: INCREMENT,
   payload: 5
});

export const decrement = () => ({
   type: DECREMENT,
   payload :5
});
```

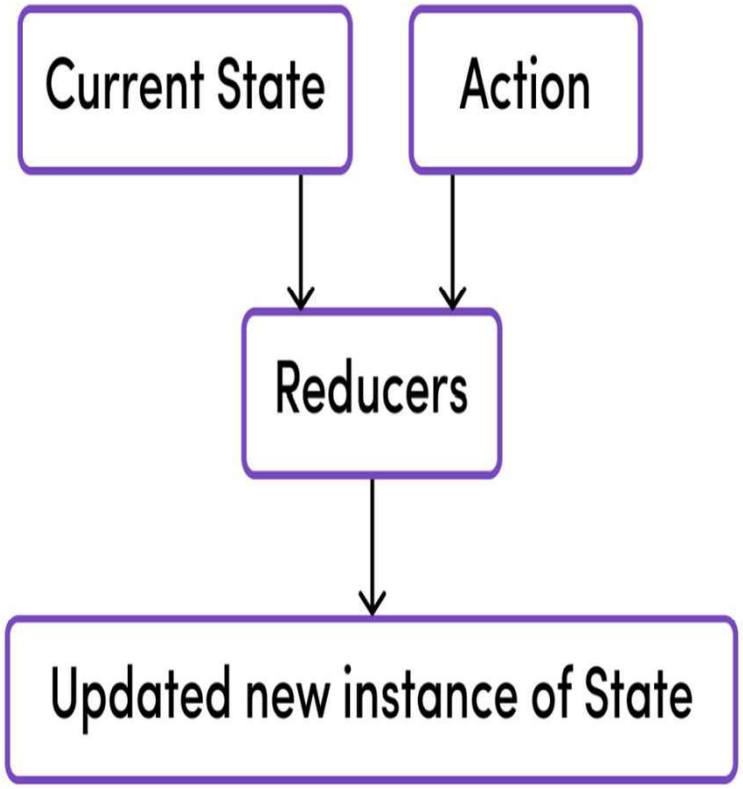
Redux Dataflow

Reducer in Redux:

- Reducers are functions that take the current state and an action as arguments, and return a new state result. **In other words, (state, action) => newState.**
- Reducers, as the name suggests, take in two things: previous state and an action. Then they reduce it to one entity: the new updated instance of state.
- **Whenever an action is dispatched, all reducers are activated.**
- Each reducer checks the action type with a switch statement. If a match is found, the reducer updates the state and returns a new instance of the global state.

Redux Dataflow

```
// src/reducers/counterReducer.js
import { INCREMENT, DECREMENT } from './actionTypes';

const initialState = 0;

const counterReducer = (state = initialState, action) => {
    switch (action.type) {
        case INCREMENT:
            return state + 1;
        case DECREMENT:
            return state - 1;
        default:
            return state;
    }
};

export default counterReducer;
```



Redux Dataflow

Action Creators in Redux:

```
<div>
    <h1>Counter: {count}</h1>
    <button onClick={() => dispatch(increment())}>Increment</button>
    <button onClick={() => dispatch(decrement())}>Decrement</button>
</div>
```

- Dispatch an action on click of the button. Or rather, to be more specific, Dispatch something known as an action creator – that is, the function increment()/decrement().
- This in turn returns an action which is a plain JS object describing the purpose of the action denoted by the type key along with payload data required for the state change.

Redux Dataflow

Principles of Redux :

1. Single source of truth

2. State is read-only.

3. Changes are made with pure functions.

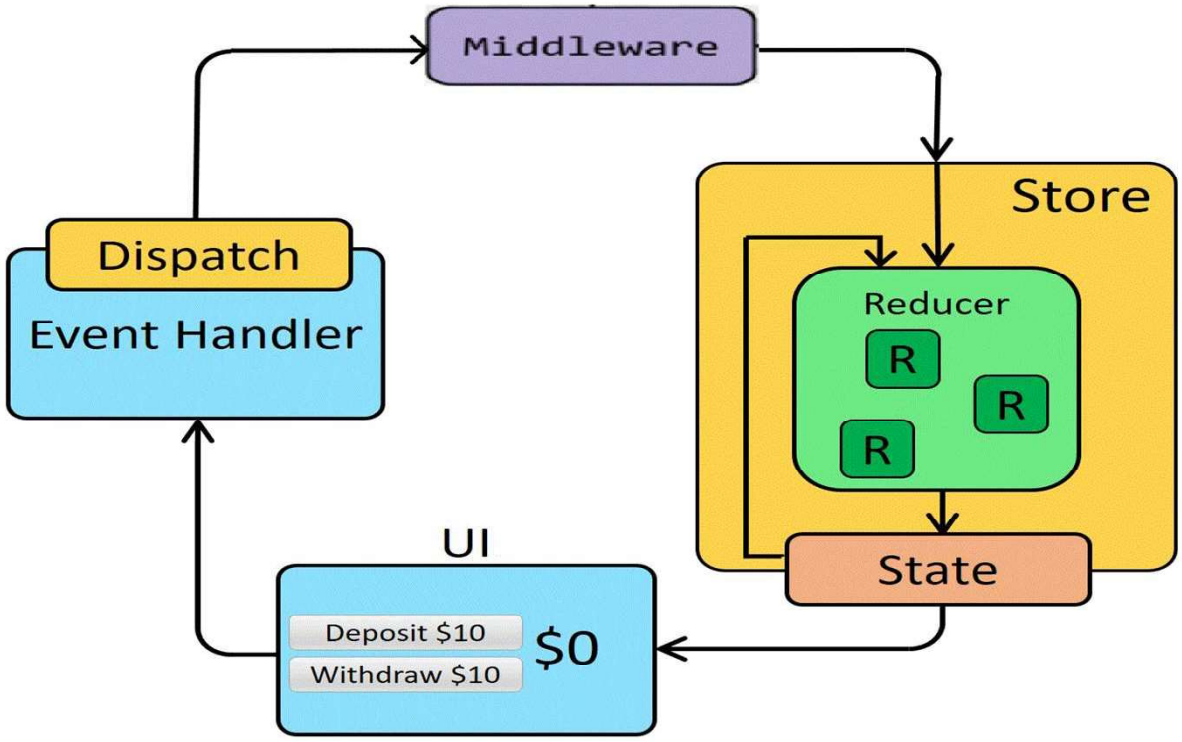
Note: ***Click Here to Access the Redux implementation in GitHub.***

Note: ***Click Here to Access the Redux Color Reducer in GitHub***

Note: ***Click Here to Access the Redux Sort Reducer in GitHub***

Redux Middleware

- Redux middleware provides a third-party extension point between dispatching an action, and the moment it reaches the reducer.



Redux Middleware

- Redux middleware are actually implemented on top of a very special store enhancer that comes built in with Redux, called **applyMiddleware**.
- In particular, middleware are intended to contain logic with side effects.
- In addition, middleware can modify dispatch to accept things that are not plain action objects.

```
// Middleware from the previous response
const myLogger = (store) => (next) => (action) => {
  console.log('I am in First Middleware');
  return next(action);
};

const secondMiddleware = (store) => (next) => (action) => {
  console.log('I am in Second Middleware');
  return next(action);
};


const stopAtTen = (store) => (next) => (action) => {
  if (store.getState() >= 10) {console.log('I am in Third Middleware');
    return next({ type: 'DECREMENT' });
  }
  return next(action);
};

const store = createStore(counterReducer, applyMiddleware(myLogger, secondMiddleware, stopAtTen));
```

Redux Middleware

Middleware Use Cases:

A middleware can do anything it wants when it sees a dispatched action:

1. Log something to the console.
2. Set timeouts.
3. Make asynchronous API calls.
4. Modify the action.
5. Pause the action or even stop it entirely.

Note: Click Here to Access the Redux Middleware in GitHub

Material UI(MUI)

- Material UI is an open-source React component library that implements **Google's Material Design.**

npm install @mui/material @emotion/react @emotion/styled

@mui/material: Provides Material-UI components such as Button, Typography, Container, etc.

@emotion/react: Provides CSS-in-JS utilities, enabling the use of the css prop and theme management in React components.

@emotion/styled: Allows you to create styled components using JavaScript with the styled API, enabling more structured component styling.

Material UI(MUI)

Advantages of Material UI

1. Ship Faster.
2. Beautiful by Default.
3. Customizability.
4. Cross-Team Collaboration.
5. Trusted by Thousands of Organizations.

```
import React from 'react';
import { useSelector, useDispatch } from 'react-redux';
import { increment, decrement } from './actions';
import { Button, Typography, Container } from '@mui/material';  // MUI components

const App = () => {
  const count = useSelector((state) => state);
  const dispatch = useDispatch();

  return (
    <Container maxWidth="sm">
      <Typography variant="h4" gutterBottom>
        Count: {count}
      </Typography>
      <Button
        variant="contained"
        color="primary"
        onClick={() => dispatch(increment())}
        style={{ marginRight: '10px' }}
      >
        Increment
      </Button>
      <Button
        variant="text"
        color="secondary"
        onClick={() => dispatch(decrement())}
      >
        Decrement
      </Button>
    </Container>
  );
};

export default App;
```

22ITC08

FULL STACK DEVELOPMENT

UNIT-III

Integration of Google MAP API and GPS Location Tracking

Google Maps (API)

- Integrating Google Maps API and GPS location tracking provides a powerful combination for creating location-based services.
- This integration allows applications to provide users with dynamic, real-time location data and mapping capabilities.

Google Maps (API)

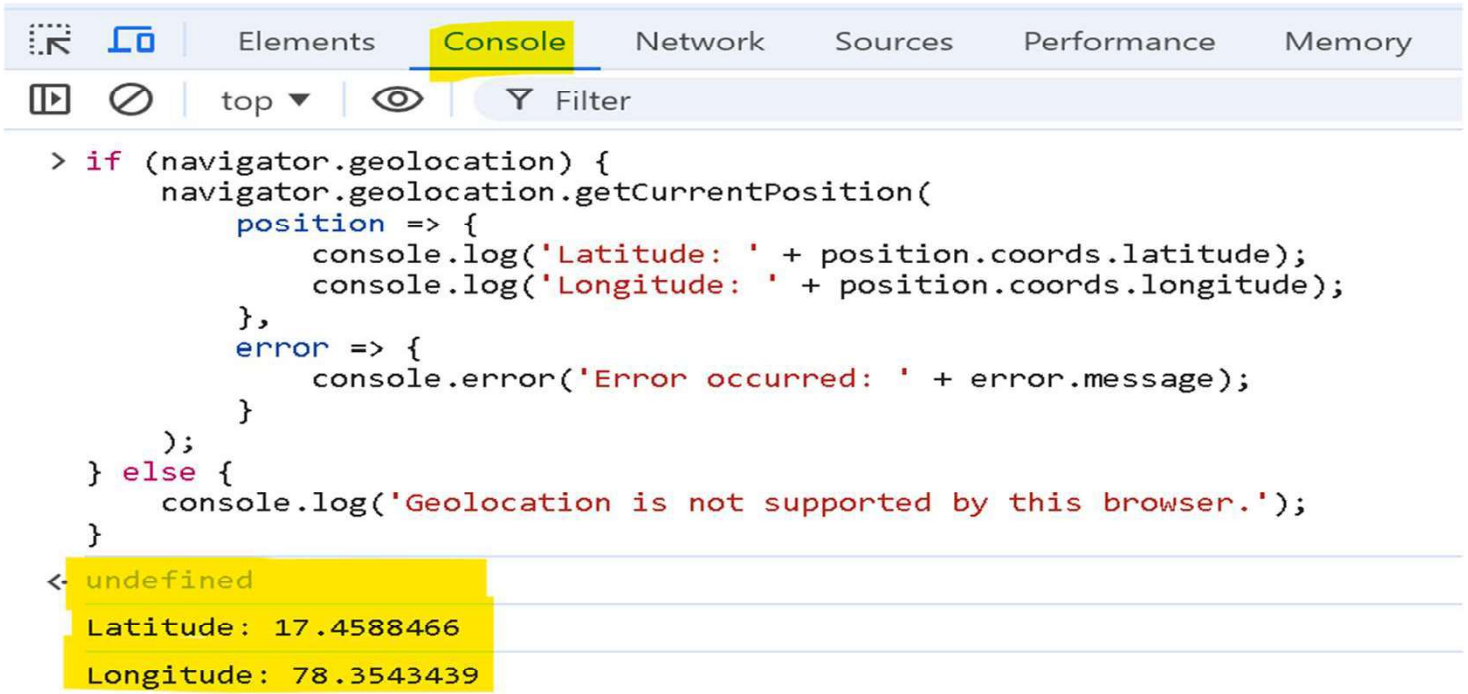
- Modern web browsers are equipped with powerful APIs that enable developers to access and utilize geographical location data.
- One of the most significant features is the Geolocation API, which allows web applications to retrieve the geographical position of a device.
- This API is built into most modern browsers and can be used to obtain latitude and longitude coordinates, making it easier to implement location-based features.

Google Maps (API)

Test the Geolocation API

1.Open the Console Tab: Click on the **Console** tab in the Developer Tools.

2.Run the Geolocation Test:



Google Maps (API)

Step 3: Grant Permissions

- When run the code, browser may prompt to allow or deny location access. You need to **allow** the access to get the coordinates.
- If you deny permission, you'll see an error message in the console.

Step 4: Check the Output

- If you granted permission, the console will display your **latitude** and **longitude**.
- If there was an error (like permission denied), you will see an appropriate error message.

Google Maps (API)

How the Geolocation API Works in Browsers?

When you call **navigator.geolocation.getCurrentPosition(),** the browser performs the following:

- **User Permission:** It prompts the user to grant permission to access their location.
- **Determination of Location:** If the user agrees, the browser determines the location using available methods:

1. **GPS:** For devices with GPS capabilities, it uses satellite signals for accurate location.
2. **Wi-Fi:** For devices without GPS, it can use nearby Wi-Fi networks to estimate the location.
3. **Cell Tower Triangulation:** For mobile devices, it may use cell towers to improve location accuracy.
4. **IP Address:** If all else fails, it may use the IP address to provide a rough estimate.

Google Maps (API)

Why an API Key is Required Despite Built-In Browser Geolocation Capabilities?

- While the Geolocation API in browsers provides basic latitude and longitude data, the Geocoding API and other Google Maps services offer enhanced functionality that requires the management of resources, security, and usage through an API key.
- Obtaining an API key from the Google Cloud Console is essential for accessing these services, ensuring secure, controlled, and scalable use of Google's powerful location-based APIs.

Google Maps (API)

How to Obtain and Use an API Key?

Note: *Click Here to Access the implementation in GitHub.*

1.Create a Project in Google Cloud Console:

1. Sign in to the Google Cloud Console.
2. Create a new project or select an existing project.

2. Enable the Geocoding API:

1. Navigate to the API Library in the Cloud Console.
2. Search for "Geocoding API" and enable it for your project.

3. Generate an API Key:

1. Go to the "Credentials" page in the Cloud Console.
2. Click on "Create credentials" and select "API key".
3. Configure the API key by setting restrictions based on IP addresses, referrer URLs, or apps to enhance security.

4. Integrate the API Key in Your Application

Google Maps (API)

```
const express = require('express');
const axios = require('axios');
const cors = require('cors');
const app = express(); const port = 3000;

const GOOGLE_MAPS_API_KEY = 'Here You have to Pass Your API Key';
app.post('/location', async (req, res) => {
  const { latitude, longitude } = req.body;

  if (!latitude || !longitude) {
    return res.status(400).json({ error: 'Latitude and Longitude are required' });
  }

  try {
    const response = await axios.get(
      `https://maps.googleapis.com/maps/api/geocode/json?latlng=${latitude},${longitude}&key=${GOOGLE_MAPS_API_KEY}`
    );

    if (response.data.status !== 'OK') {
      return res.status(500).json({ error: 'Failed to fetch address' });
    }

    const address = response.data.results[0].formatted_address;
    res.json({ address });
  } catch (error) {
    res.status(500).json({ error: 'An error occurred while fetching the address' });
  }
});

app.listen(port, () => {
  console.log(`Server is running on http://localhost:${port}`);
});
```

Google Maps (API)

Real Time Applications:

1. **Navigation Systems:** Real-time turn-by-turn navigation using Google Maps and GPS data.
2. **Ride-Sharing Services:** Tracking the location of drivers and riders.
3. **Delivery Services:** Monitoring delivery vehicles and optimizing routes.
4. **Fitness Apps:** Tracking runs, bike rides, and other outdoor activities.
5. **Geofencing:** Triggering notifications or actions when entering or leaving predefined areas.